UNIVERSITÄT AUGSBURG

The Model of Computation of CUDA and its Formal Semantics

Based on the Master's Thesis of Axel Habermaier

Axel Habermaier

Report 2011-14 October 2011

Institute for
Software & Systems
Engineering

INSTITUT FÜR INFORMATIK
D-86135 AUGSBURG

Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE

Abstract

We formalize the model of computation of modern graphics cards based on the specification of Nvidia's Compute Unified Device Architecture (CUDA). CUDA programs are executed by thousands of threads concurrently and have access to several different types of memory with unique access patterns and latencies. The underlying hardware uses a single instruction, multiple threads execution model that groups threads into warps. All threads of the same warp execute the program in lockstep. If threads of the same warp execute a data-dependent control flow instruction, control flow might diverge and the different execution paths are executed sequentially. Once all paths complete execution, all threads are executed in parallel again. An operational semantics of a significant subset of CUDA's memory operations and programming instructions is presented, including shared and non-shared memory operations, atomic operations on shared memory, cached memory access, recursive function calls, control flow instructions, and thread synchronization.

Based on this formalization we prove that CUDA's single instruction, multiple threads execution model is safe: For all threads it is provably true that a thread only executes the instructions it is allowed to execute, that it does not continue execution after processing the last instruction of the program, and that it does not skip any instructions it should have executed. On the other hand, we demonstrate that CUDA's inability to handle control flow instructions individually for each thread can cause unexpected program behavior in the sense that a liveness property is violated.

Contents

1 Introduction

Stanford University's Folding@Home[1] is a distributed computing application designed to study protein folding and protein folding diseases. With more than 70 peer-reviewed papers[2], the project's aim is to help form a better understanding of the protein folding process and related diseases including Alzheimer's disease, Parkinson's disease, and many forms of Cancer. Even though some proteins fold in only a millionth of a second, computer simulations of protein folding are extremely slow, taking years in some cases[3]. Being so computationally demanding, the Folding@Home project distributes the necessary calculations to thousands of client computers. People from around the world support the project by downloading the client and donating their idle CPU time to the project. Additionally, in recent years people have been able to run the client on their graphics card as well, resulting in a significant increase in computational power devoted to Folding@Home. All in all, the computational resources available to the project have already crossed the peta FLOPS barrier.

Since the molecular dynamics simulations performed by the Folding@Home client are computationally expensive, running them on GPUs has the potential of drastically reducing computation times. With the advent of general purpose GPU programming supported by both Nvidia and AMD graphics cards, it has become viable to develop new Folding@Home clients that run the computations on the GPU instead of the CPU. The results speak for themselves: The GPU clients outperform their CPU counterparts by at least two orders of magnitude, even though the peak theoretical power of the GPUs has not yet been reached, i.e. further optimizations are still possible. Additionally, many molecules simulated by the CPU client are too small to fully utilize the graphics card because of its massively parallel nature. Therefore, it is expected that the performance gap between CPUs and GPUs will increase even further for simulations of larger molecules [1]. But even today GPUs already play an important role for Folding@Home as illustrated by figure 1.1. Even though the number of active GPUs contributing to the project is way smaller than the amount of CPUs, the actual floating point operations per second exceed those of the CPUs by an order of magnitude. The same applies to the PlayStation 3, which runs a special client optimized for the console's Cell chip.

Platform	TFLOPS	Active Processors
CPUs (Windows)	213	223933
AMD GPUs	697	6481
Nvidia GPUs	2199	8760
PlayStation 3	1671	28090

Figure 1.1: Folding@Home Client Statistics[4]

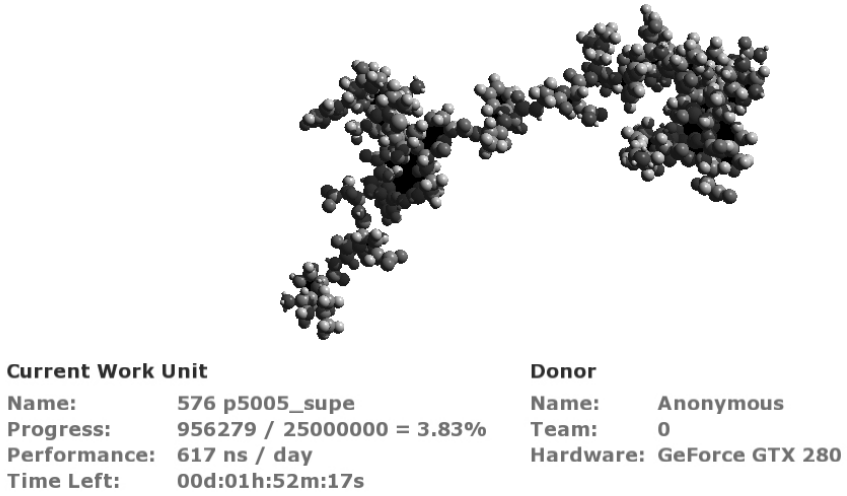
---

[1] `http://folding.stanford.edu/`

[2] `http://folding.stanford.edu/English/Papers`, last access 2010-10-28

[3] `http://folding.stanford.edu/English/Science`, last access 2010-10-28

[4] based on `http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats`, last access 2010-10-28; see also `http://folding.stanford.edu/English/FAQ-flops`



Current Work Unit
Name: 576 p5005_supe
Progress: 956279 / 25000000 = 3.83%
Performance: 617 ns / day
Time Left: 00d:01h:52m:17s

Donor
Name: Anonymous
Team: 0
Hardware: GeForce GTX 280

Figure 1.2: Screenshot of the Folding@Home Client for Nvidia GPUs

There are some issues, however, that the GPU client has to deal with. As general purpose GPU programming has only been recently introduced, development tools and environments are not yet as mature as their CPU counterparts. Additionally, GPU programming is still very close to the hardware, so problems like efficient memory accesses, CPU/GPU synchronization, and divergent control flow must be considered carefully. Precision of floating point operations is an important issue too, as using double precision computations instead of single precision ones is significantly slower on today's GPUs.

Another area where GPUs excel is the reconstruction phase of magnetic resonance imaging. There, the data sampled by a scanner needs to be transformed into an image that is then presented to a human for further analysis. Again, a speedup of two orders of magnitude can be achieved by offloading the required transformations to the GPU. As [2, 8] shows, hardware-specific optimizations are of utmost importance, though: A naive implementation is only about ten times faster then the equivalent CPU program. However, once memory accesses are optimized and the hardware's trigonometry function units are fully utilized, GPUs outperform CPUs by a factor of about 100 in total and one specific sub-problem is even computed around 357 times faster.

Figure 1.3: MRI Scan of a Human Head[5]

As these two examples show, GPUs can significantly speed up certain algorithms and operations — provided that they suit the novel model of computation of GPUs which vastly

---

[5] `http://de.wikipedia.org/wiki/Magnetresonanztomographie`, last access 2010-10-28

differs from the traditional one of x86 CPUs. Because of those tremendous performance improvements, Folding@Home will most likely continue to focus on the development of the GPU clients. But besides Folding@Home, there are many other research projects in physics, chemistry, biology, and other sciences that benefit significantly from GPU-accelerated computations. On the other hand, GPUs are also used outside academia in real-world applications where they might affect people's life or health as the aforementioned MRI example illustrates. Thus, developers of such applications must be able to guarantee that their programs behave correctly in all cases — either by extensive testing or by formally proving certain properties of their programs. However, general purpose GPU programming is a novel field of research and academic interest is mostly focused on finding ways to adapt specific problems and algorithms to the GPU's programming model and on optimizing GPU programs in order to make the computations as efficient as possible. Correctness, on the other hand, is only of secondary interest and is not proven formally in most cases. By contrast, this report is aimed at formalizing the model of computation of CUDA, Nvidia's general purpose GPU programming infrastructure. With some further work that is outside the scope of this report, it should be possible to use the presented formalization to formally prove properties of GPU-accelerated programs.

Based on an informal overview of the hardware architecture and the programming model in chapter 2, CUDA's memory model and program semantics are formalized in chapters 4 and 5, respectively. Although the formalization is not exhaustive, it includes most of the important features and instructions supported by CUDA, like cached memory operations, thread synchronization, thread divergence caused by data-dependant control flow instructions, atomic memory operations, and many more. Due to the complexity of the underlying hardware and CUDA, contrasting GPU and CPU semantics as a whole is outside the scope of this report. However, chapter 6 indeed shows a difference in program behavior caused by the GPU's single instruction, multiple thread (SIMT) architecture. While in real-world scenarios mostly relevant for optimal performance, the SIMT architecture might cause deadlocks that would not occur if the program were executed on the CPU. We show that only terminating programs are provably correct in the general case.

2 Overview of CUDA

Compared to traditional x86 CPUs, CUDA utilizes a different programming model to take advantage of the massively parallel nature of modern GPUs. CUDA programs are concurrently executed by thousands of threads in parallel and have access to a variety of different memory types, all of which have distinct access characteristics, sizes, and uses. Unlike x86 CPUs, GPUs do not execute each thread individually, but rather group threads into warps. These warps are executed individually, but all threads of the same warp execute their instructions in lockstep. If the control flow of threads of the same warp diverges due to a data-dependant conditional branch instruction, execution of the warp is serialized for each unique path, disabling the threads that did not take the path. When all paths complete, the threads converge back to the same path and they execute concurrently in lockstep again.

With the first release of CUDA as recently as November 2006 [3], the programming model is still very close to the hardware. Therefore, CUDA developers must take into account the distinctive traits and feature sets of the hardware for a program to run as efficiently as possible on the GPU. While efficiency and performance are not the primary topic of this report, the formalization of the semantics is indeed affected by the novelty of general purpose GPU programming; we generally define the semantics in a low-level fashion and often refer to the hardware implementation of a specific feature. As GPUs have just recently gained features such as indirect function calls, recursion and dynamic memory allocation, it will probably take several years before high-level abstractions like virtual machines similar to Java will become viable from a performance standpoint. At that point, it should be possible to define a formal semantics that is less closely tied to the underlying hardware.

Since the hardware is such an integral part of CUDA, we first take a brief look at the architecture of modern GPUs before we give an overview of CUDA's programming model. As the primary use of modern GPUs is still the acceleration of graphics rendering, we explain the evolution of CUDA based on the evolution of the underlying hardware from fixed-function graphics co-processors to fully programmable general purpose processors. In this chapter, we also introduce the programming languages and compiler infrastructure used to develop CUDA applications and also outline some of the differences and similarities of CUDA compared to other general purpose GPU programming frameworks. Except for section 2.4, this entire report focuses on Nvidia's GPUs, as CUDA is a Nvidia exclusive technology. We focus on CUDA instead of multi-vendor frameworks such as OpenCL and Direct Compute because CUDA's specification was the most comprehensive one at the time of writing.

2.1 Evolution of GPUs

In the late 1990s, GPUs were used exclusively to accelerate fixed-function graphics processing. Using an API like OpenGL or Direct3D, a developer instructed the GPU to draw triangles on the screen, using a combination of different states to specify how the triangles should be textured, lit, alpha-blended, etc. At that time, developers were only able to use the functionality that was supported by the hardware and exposed by the graphics API; programming the GPU directly was not yet possible.

Having only a limited instruction set and no programmability was the main reason why the GPUs of that time outperformed the CPUs: The chip developers were able to use the chip's transistor budget to implement slow operations like texture filtering and raster operations in hardware, as they did not have to implement transistor-heavy functionality like data caches and branch prediction logic. Furthermore, graphics rendering is an inherently parallel task; each vertex of a geometry set and each rasterized pixel can be processed independently and in parallel. This allowed the GPUs to increase the performance by just putting more of the same functional units on the chip.

Eventually, it became apparent that the fixed-function design severely limited the graphical effects that can be achieved by GPU based renderers. In 2001, Nvidia introduced a new generation of graphics processors which for the first time allowed the GPU to be programmed directly by the developer. The new programmability could be used to specify shader programs that operated on individual vertices and pixels, however, shaders had to be very short and control flow instructions were missing.

Over time, GPUs became faster by allowing more shaders to be executed in parallel, while at the same time increasing the instruction count limit and adding new operations to the instruction set. At the same time, both OpenGL and Direct3D introduced the C-like languages GLSL and HLSL respectively which could be used to develop shaders in a more higher-level fashion than writing assembly code.

With the DirectX 10 generation of graphics card, GPUs gained the ability to be used for general purpose processing, circumventing the traditional graphics APIs thanks to the introduction of CUDA (Nvidia) and CAL (AMD). Previously, some developers had already attempted to use GPUs for non-graphics related tasks, but had to fit their algorithms into the limits imposed by the graphics APIs. As those were not designed for such usage, the developers had to work around the limitations of both the API and the hardware, reducing the possible performance gains. Another important change of the DirectX 10 generation of GPUs was the introduction of unified shading. Previous generations of graphics card had distinct hardware units for vertex and pixel shaders. That was problematic in some cases; for example if a draw operation was heavily pixel shader bound, the vertex shader parts of the chip were running idle. As DirectX 10 added yet another type of shader, the geometry shader, the traditional design of the shader units would have become too wasteful. Therefore, all DirectX 10 chips consist of unified shading units, allowing the same hardware units to be used to execute vertex, geometry, and pixel shaders. For general purpose programming, this meant that a program was now able to use all hardware units available on the graphics card. Still not usable are some of the fixed-function units that are still needed for performance reasons; those remain idle when the chip executes a non-graphics program. Future generations of graphics cards will probably replace more and more of the current fixed-function functionality by programmable units that then might be usable by non-graphics related programs as well.

With CUDA, developers were now able to write general purpose applications without having to pay attention to the graphics heritage of the GPUs. Additionally, Nvidia added dedicated functionality to the chips which — until now — are only relevant to general purpose programming, like shared and non-shared read- and writable memory, atomic memory operations, and explicit synchronization points. The DirectX 11 generation added even more features like recursion, a cache hierarchy for improved latency, indirect function calls, and more. All of these features are exposed directly to the developers who can use, for example, an extended version of C to program the GPU [2, 2]. Some of these features, especially indirect function calls, are likely to be supported by future versions of the shader programming languages as well. DirectX 11, for instance, has already added support for interfaces and classes to HLSL, although currently calls to an interface function are not virtual but resolved at shader link time [4].

For the future, it is expected that the importance of graphics APIs will vanish whereas the general programmability APIs will gain more traction. Tim Sweeney, one of the developers of the successful Unreal Engine 3, is already predicting that they will write their next generation renderer in a programming language such as C++ or CUDA, abandoning OpenGL and Direct3D entirely[1]. Another possibility is to take a hybrid approach, where the graphics APIs are used for the basic rendering work and the greater flexibility of general purpose APIs is used to accelerate post-processing effects. For instance, there is already a game that use a compute shader, the DirectX 11 equivalent of a CUDA program, to calculate the screen space ambient occlusion effect for scenes rendered with a traditional renderer[2].

2.2 Architecture of a Modern GPU

GPUs have both strengths and weaknesses compared to traditional (multi-core) CPUs. Therefore, it depends on the program whether it can take advantage of the GPU's superior computational power or whether it is actually faster to run it on the CPU. Because of their graphics acceleration heritage, GPUs specialize on highly parallel as well as compute and memory bandwidth intensive tasks, whereas CPUs excel at sequential, control flow intensive ones. Hence, GPUs have more computational power and bandwidth than the CPUs at the cost of an increased operation latency. This latency, however, can be completely hidden if the program being run is sufficiently parallel, i.e. a massive number of threads can be run concurrently, as explained later on.

While there are many different chips capable of executing CUDA applications, we focus on the latest generation of graphics cards called Fermi or GF100. The remainder of this section discusses those parts of Fermi's architecture that influence the semantics of CUDA programs, leaving out all the complexities that are not relevant for this report. The figures below are simplified versions of figures found in [5] and [6]. Additional information about Fermi's architecture can be found in [5], [7], [3], [8], [9], [6], and [2][3].

Fermi is Nvidia's current high-end chip and has — like all recent graphics chips — a modular design: With relatively low engineering effort, performance, mainstream and low-end chips can be derived from the top model, offering less features and performance at a lower price point. All of these GPUs use the same basic architecture and only differ in memory sizes and the number of functional units still left in the chip. Thus, when we speak of the "Fermi architecture", we actually mean the basic architecture described below, parameterized over the memory sizes and the number of functional units. For a better understanding, the following sections and figures use the concrete values of the high-end chip GF100 instead of abstract placeholders. Later, we do indeed use global constants to

---

[1] `http://arstechnica.com/gaming/news/2008/09/gpu-sweeney-interview.ars`, last access 2010-10-10

[2] `http://www.anandtech.com/show/2848/2`, last access 2010-10-10

[3] Unless noted otherwise, all referenced Nvidia documents can be found at `http://developer.nvidia.com/object/gpucomputing.html`, last accessed 2010-10-31.

parameterize the domains and functions, as we want the semantics to be applicable to all Fermi-based GPUs.

2.2.1 Chip Layout

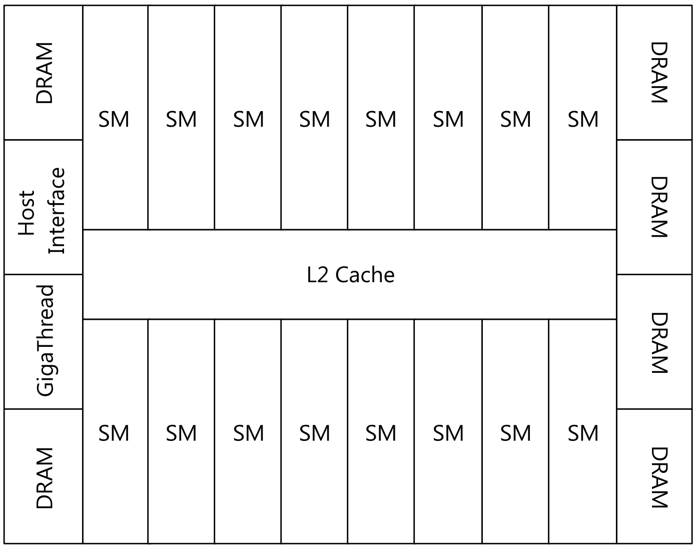


Figure 2.1: Fermi Chip Layout

Figure 2.1 gives an overview of Fermi's basic chip layout. The GPU has six memory partitions positioned around a central L2 cache. Fermi supports cached read and write access to shared and non-shared memory as well as atomic and volatile memory operations. Section 2.2.3 takes a closer look at the different kinds of memory found on a Fermi chip.

The GPU uses the host interface for all communications with the CPU. The CPU has read and write access to the GPU's memory, loads CUDA program code onto the GPU and launches programs on the GPU with a given set of input parameters. Once the program has been completed, the CPU is informed that the output produced by the program is ready for consumption by the host program. The communication between the CPU and GPU can be complex as both processors operate independently and in parallel. It is the graphics driver's responsibility to synchronize both processors when specific events occur. We do not go into further details concerning the interaction of CPUs and GPUs and also do not consider multi-GPU setups.

The Giga Thread scheduler is responsible for distributing waiting threads to the 16 streaming multiprocessors (SM). The scheduling algorithm used by the scheduler is unknown. A SM executes many threads in parallel and issues read and write requests to the L2 cache or directly to the DRAM. The architecture of SMs is explained in further detail in section 2.2.2.

As mentioned above, Fermi's architecture was designed to be flexible, such that parts of the chip can be easily removed to produce cheaper and smaller chips. This is achieved by reducing the L2 cache size, the amount of DRAM partitions, and the number of SMs on the chip, as well as by removing other fixed-function, graphics-related functionality not shown in figure 2.1. In fact, for the first derivation of GF100, GF104, Nvidia also modified the SMs in an attempt to improve graphics performance at the expense of reducing the performance of certain CUDA-exclusive features. The GF104 changes, however, only affect the number and capabilities of the computation units within the SMs and have therefore no direct effect on the semantics of CUDA programs.

2.2.2 Processing Units

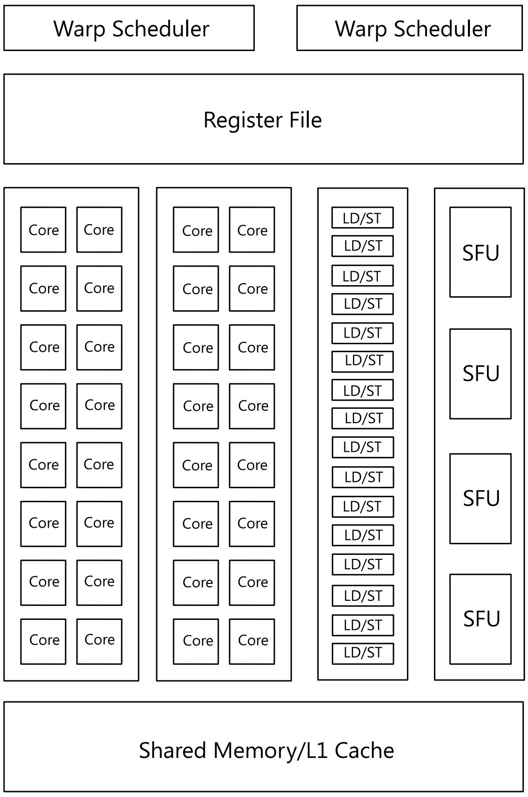


Figure 2.2: Fermi Streaming Multiprocessor

Each streaming multiprocessor can have up to a total of 1536 threads scheduled for execution. The architecture used to manage such a large amount of threads is shown in figure 2.2.

Each SM consists of 32 CUDA cores capable of performing floating point and integer calculations. Additionally, there are 16 load-store units for memory operations and four special-function units for transcendental operations and other special functions like sin, cos, and exp. The units read from and write to the register file consisting of 2^{15} 32-bit registers. Memory operations can optionally use the L1 cache when accessing the DRAM. For efficient thread communication, the SM also provides up to 48 KByte of shared memory.

Nvidia uses a single instruction, multiple threads (SIMT) architecture to maximize the utilization of the SM's functional units. When the Giga Thread scheduler dispatches threads to the SM, the SM groups 32 threads into a warp. All threads of a warp execute the same instruction in parallel. The two warp schedulers simultaneously schedule two warps ready to execute their next instruction. Usually, after a warp has executed an instruction, it is not scheduled again to execute its next instruction. To hide the latency caused by memory operations or even register reads and writes, the warp schedulers always try to schedule a warp that can be executed immediately without waiting for any data to become available. Whereas context switches for threads are an expensive operation on CPUs, there is no overhead associated with warp switching on the GPU; all thread state like the current program counter and registers is stored by the SM for the lifetime of the warp. This allows GPUs to minimize the impact of high-latency operations like DRAM access; instead of stalling the warp or the entire SM, the warp schedulers are free to schedule any other warp, without any overhead, for maximum efficiency.

The scheduling algorithm used by the warp schedulers is unknown. It is known, however, that one scheduler manages all warps with odd ids, whereas the other scheduler manages the warps with even ids. Each scheduler can only utilize 16 of the 32 cores — except when scheduling double precision instructions, in which case all 32 cores are used by one of the schedulers and the other scheduler cannot issue any instruction to the cores. Each scheduler can issue an instruction to one of the four execution blocks of an SM; execution blocks are the CUDA cores divided into two blocks with 16 cores each, the group of 16 load-store units, and the four special-function units. Depending on the type of the operation, an issued instruction takes several clock cycle to complete: operations executed on the cores or load-store units take two clock cycles to complete, whereas operations run on the four special-function units take eight cycles to complete. In any case, it takes only one clock cycle to issue the instructions and the warp schedulers can already issue the next instructions to idle units without waiting for previous instructions to complete, as illustrated by figure 2.3.

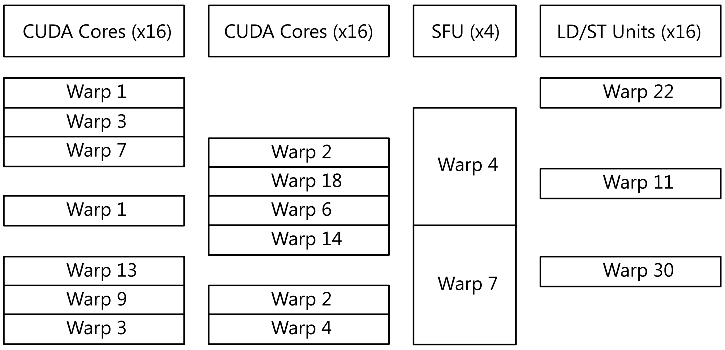


Figure 2.3: Utilization of Execution Blocks

Giving up individual thread execution in favor of the SIMT model allows the GPU to be designed with less transistors and enables some memory operation optimizations otherwise impossible. A smaller chip usually allows higher clock speeds, and coalescing memory operations performed by many threads into fewer, but larger memory transactions has the potential of significant performance improvements. Moreover, the hardware must fetch and process an instruction only once per warp and not once per thread, allowing these costs to be amortized over many threads [2, 6.1]. However, warps become problematic when threads are allowed to use data-dependent control flow instructions that cause threads to take different execution paths. In graphics programming, shaders used to avoid control flow instructions for performance reasons, and often the decision of which path to take could already be made at compile-time. With shaders getting more complex and the introduction of general purpose programming, supporting branching efficiently became increasingly important. Compared to older generations of graphics cards, the SIMT architecture employs a sophisticated algorithm to support arbitrary branching of individual threads. Basically, divergent control flow is handled by executing all paths serially with all threads not on the current path deactivated. After all paths have completed, all threads execute the same path again in parallel. Thus, the algorithm's worst case performance reduction is proportional to the warp size. For this reason, CUDA developers try to avoid control flow instructions which cause threads of the same warp to take different paths. In contrast, different warps are executed independently, so there is no performance gain or penalty when they are executing common or disjoint code paths. Since the handling of divergent control flow at the warp level can cause unexpected program behavior, we examine the branching algorithm used by Fermi thoroughly in later chapters and prove its correctness for terminating programs in section 6.

2.2.3 Memories and Caches

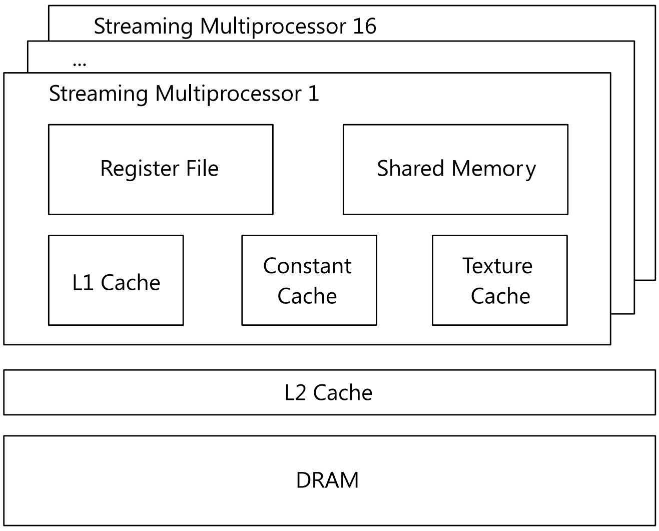


Figure 2.4: Fermi's Memories and Caches

As shown in figure 2.4, there are many different types of memories and caches placed at various locations on the chip. Each one of the 16 streaming multiprocessors has its own register file and shared memory, as well as L1, constant, and texture caches. Threads cannot access the memories and caches of any SM other than the one they are executed on.

A number of registers is assigned to each thread running on a SM, the exact amount depending on the program executed by the threads. Once assigned to a thread, a register is exclusively accessible by its assigned thread and becomes available for reassignment only when the thread terminates. If a CUDA program requires many registers per thread to execute, the maximum number of threads that can be concurrently scheduled on a SM decreases when the SM runs out of register file space. Having less warps available for scheduling might degrade performance, as it becomes more likely that all warps are blocked waiting for the completion of a high-latency operation like uncached DRAM access.

The SM's shared memory and L1 cache are actually the same on-chip memory that can be configured to provide either 16 KByte of shared memory and a 48 KByte L1 cache or vice versa. Accessing shared memory is almost as fast as accessing a register, so using shared memory is significantly faster than accessing the DRAM. However, Fermi can use the L1 cache to reduce DRAM access times. Furthermore, the 768 KByte L2 cache shared by all SMs also speeds up DRAM accesses, whereas the constant cache is used to speed up access to special constant data in the DRAM. In graphics mode, many pixel shaders sample values from textures stored in DRAM. Although not shown in figure 2.2, all SMs have special hardware units for texture sampling operations that perform address calculations and filtering operations efficiently in hardware. We do not consider texture operations in the following chapters, despite the fact that CUDA programs can indeed use the streaming multiprocessor's texturing hardware.

The DRAM is shared by all streaming multiprocessors. With up to several gigabytes it is the largest type of memory on the GPU. In graphics mode, the DRAM is used to hold all texture and vertex data. In general purpose computing mode, any data can be stored in the DRAM and all threads have full read and write access.

Based on this hardware-centric overview of the available memory types, section 2.2.3 revisits this topic from the software point of view.

2.2.4 Compute Capability

Since the release of CUDA in 2006, new features have been added to the hardware and subsequently to CUDA as well. The features and instructions supported by a GPU are defined by the GPU's compute capability, which also specifies some hardware constants like the maximum number of concurrently resident threads on a SM or the number of registers per SM. Fermi's compute capability is 2.0. The first digit, the major revision number, denotes the core architecture of the GPU. The second digit, the minor revision number, reflects incremental improvements to the core architecture. A device of higher compute capability is backward compatible to a device of lower compute capability, as devices of higher compute capability support a superset of the features of older devices [8, 3, 1.2.1 and 2.5 respectively]. During the writing of this report, Nvidia released the GF104 chip that supports compute capability version 2.1. However, no new instructions were added with version 2.1. The most significant changes concern the architecture of streaming multiprocessors and the fact that the warp schedulers now support the scheduling of two successive, independent instructions of the same warp in parallel [10, 5.2.3 and G.4.1]. The formal semantics defined in the following chapters reflects compute capability version 2.0.

An overview of supported instructions and device parameters for all current GPUs can be found in [3, chapter G.1]. For example, atomic memory operations were introduced with compute capability version 1.1 and are therefore unsupported by hardware of compute capability 1.0. The number of registers per SM has increased from 8192 for devices of compute capability 1.0 and 1.1 to 32768 for Fermi. On the other hand, the warp size remained constant at 32 threads per warp for all currently available GPUs. There are also device parameters, such as the amount of streaming multiprocessors on the chip, that can vary even within the same compute capability version.

As this report focuses solely on Fermi, we consider all CUDA features to be supported by the underlying hardware but still abstract from specific numbers such as the amount of registers per SM, as mentioned above. Since all lower compute capability versions are subsets of compute capability 2.0, it should be possible to construct a formal semantics for older CUDA versions by removing the unsupported features from the semantic domains and rules defined in the following chapters. However, this report does not explore the feasibility of this idea.

2.3 CUDA Programs

At least for the time being, operating systems such as Windows and Linux launch all programs on the CPU. Therefore, a CUDA application is started on the CPU and must use the CUDA runtime to launch a calculation on the GPU. As the CPU and GPU are usually fully independent chips, they operate in parallel, making explicit synchronization necessary. The graphics driver is responsible for handling the synchronization details and hence hides these complexities from the CUDA developer. The part of the CUDA application that is executed on the CPU is called the host program, whereas the part that is executed on the GPU is called the device program or kernel.

Program 2.1 is a simple CUDA program that squares the values of an array[4]. It is written in CUDA-C, an extension of the C programming language developed by Nvidia to simplify the development of general purpose applications for the GPU. Even though we use CUDA-C for the introductory example, we eventually switch to using PTX, an assembly level language for GPU programming. We use PTX to define the formal semantics of CUDA, because PTX makes it more explicit what operations are actually performed on the hardware. For example, in CUDA-C we don't know if `a = 1;` is a cached or uncached write, whereas in PTX the statement `st.global.ca [a], 1;` states the cache operation `ca` explicitly. By using PTX, we avoid having to "guess" which cache operation to use in this case. PTX is introduced in section 2.3.3.

We explore the basic structure of CUDA programs based on how the host (the CPU) and the device (the GPU) execute program 2.1. When the operation system launches program 2.1, the CPU starts executing the function `main` at line 9. `main` is a standard C function which is entirely executed by the CPU. It calls CUDA runtime functions to interface with the GPU, but all interactions with the GPU are completely handled by the CUDA runtime and the graphics driver.

In line 9, two pointer variables are declared that are used to hold the input array which is to be squared by the GPU. In this example, we only start 32 threads on the GPU, therefore we have to allocate enough memory to hold 32 floating point values. The host memory for the array is allocated in line 11 and the pointer to the starting address of the allocated memory is stored in `a_h`. The next line uses the CUDA function `cudaMalloc` to allocate an array of equal size on the GPU's DRAM. The pointer to the address of the array in device memory is stored in `a_d`; it has no meaning for the host and dereferencing it would very likely result in either an access violation or reading garbage.

The CPU then fills the host array with some values. Afterwards, these values are copied into device memory using CUDA's `cudaMemcpy` function. Both `a_h` and `a_d` now hold the same data, once in host memory and once in device memory. This is necessary because the GPU usually has no access to host memory and therefore cannot read `a_h` directly. Now that the GPU has access to all required input data, the CPU launches the `squareArray` function on the GPU in line 17. The <<< `...` >>> syntax specifies the execution configuration, in this

---

[4] Program 2.1 is a simplified version of the program found at `http://llpanorama.wordpress.com/2008/05/21/my-first-cuda-program/`.

```
 1 __global__ void squareArray(float *a)
 2 {
 3   int idx = threadIdx.x;
 4   a[idx] = a[idx] * a[idx];
 5 }
 6
 7 int main(void)
 8 {
 9   float *a_h, *a_d;
10   int size = 32 * sizeof(float);
11   a_h = (float *)malloc(size);
12   cudaMalloc((void **) &a_d, size);
13
14   // Initialize host array...
15
16   cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
17   squareArray <<< 1, 32 >>> (a_d);
18   cudaMemcpy(a_h, a_d, size, cudaMemcpyDeviceToHost);
19
20   // Use the results...
21
22   free(a_h);
23   cudaFree(a_d);
24 }
```

Listing 2.1: A simple program written in CUDA-C that squares the values of an array

case it means that the GPU should use 32 threads to execute the function. Also, the pointer to the array in device memory is passed to the GPU.

The GPU runs the `squareArray` function defined at line 1, generally called a kernel, once for each of the 32 threads. The Giga Thread scheduler assigns the threads to one streaming multiprocessor. Since the warp size of all current GPUs is 32, the SM puts all threads into one single warp. The warp then executes all 32 threads concurrently. `threadIdx.x` is a predefined variable accessible by all functions running on the GPU and returns the index of the thread; here a unique value between 0 and 31 is returned for each thread. At line 4, the thread index is used to read the input data from global memory using the pointer passed to the kernel. Subsequently, the squared value is written back to the array. Since thread indices are unique, this means that all 32 elements of the array are squared and updated at the same time.

After the GPU has completed the execution of the kernel, the CPU resumes execution at line 18 where it copies back the result values from device memory into host memory. The CPU can then use the results and eventually frees the memory allocated on both the device and the host before the program exits.

CUDA supports some more complex forms of CPU and GPU synchronization like streams and events described in [3, 3.2.7.5 and 3.2.7.6], whose primary purpose is to improve the level of parallelism between the CPU and the GPU. However, we do not consider any form of CPU/GPU interaction for the remainder of this report and focus solely on the GPU semantics.

Now that we have some basic understanding of the structure of CUDA programs, we take a closer look at how CUDA organizes threads on the GPU and the characteristics of the different types of memory available to CUDA programs. Thereafter we give a brief introduction to the PTX programming language and the compilation process of CUDA programs.

2.3.1 Thread Organization

When a kernel function is launched on the GPU it is executed by many threads in parallel. The hardware automatically assigns a thread index to each thread that the program can use to identify the thread. For example, program 2.1 uses the thread index to read from and write to a specific memory address by using the thread index to access a value of an array. Thread indices are three-dimensional, because CUDA programs often deal with two- or three-dimensional programs. For instance, a GPU accelerated raytracer might use one thread to calculate the final color of one pixel. The location of a pixel is stored in a two-dimensional vector which can thus be trivially mapped to the thread's index. While three-dimensional indices are convenient, sometimes a program might also require a thread's one-dimensional index; the CUDA documentation specifies how a one-dimensional index can be obtained from a three-dimensional one [3, 2.2].

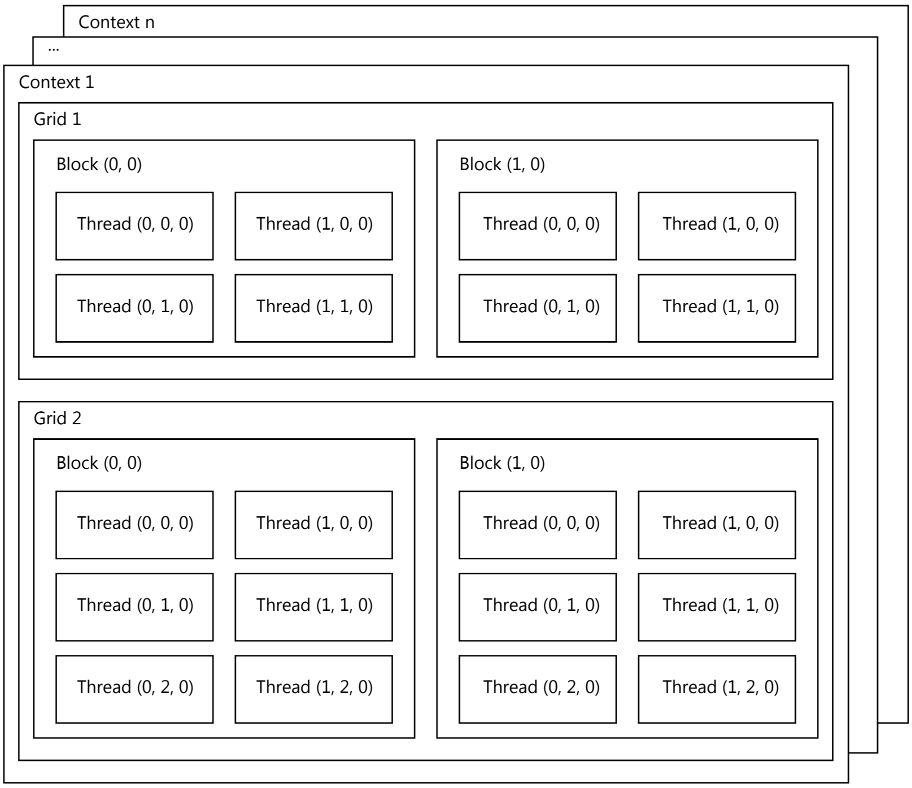


Figure 2.5: CUDA Thread Hierarchy

Figure 2.5 shows how CUDA organizes threads on the GPU. Threads are grouped into thread blocks, also called cooperative thread arrays, and within a thread block each thread index is unique. All threads belonging to the same thread block are executed on the same streaming multiprocessor. When a streaming multiprocessor has free capacity, the Giga Thread scheduler assigns an unscheduled thread block to the SM. The SM creates all threads and allocates the necessary resources to execute them. Threads with consecutive indices are grouped into a warp, i.e. threads 0 to 31 belong to warp 0, threads 32 to 63 belong to warp 1, and so on. If the amount of threads within a thread block is not a multiple of the warp size, the last warp is not fully populated. Warps are not shown in figure 2.5, because they are not part of the CUDA specification [2, 4.5]. A streaming multiprocessor must have enough resources available to create all threads of a block, otherwise the Giga Thread scheduler cannot issue a pending thread block to the SM. Therefore, the number of thread blocks and threads that can concurrently be allocated on a processor depends on the number of registers each thread requires to execute the program, the number of threads within a thread block, the SM's register file size, the maximum number of resident warps and blocks allowed on a processor, and the available and required amount of shared memory. Threads of the same thread block can communicate and share data through shared memory and can also synchronize their execution to avoid race conditions when accessing shared memory. Threads not belonging to the same thread block can only cooperate using global memory. However, no synchronization mechanism exists between threads of different blocks, so reading global data written by other threads is generally unsafe.

The streaming multiprocessors assign a unique block index to each block. Blocks are organized two-dimensionally within grids and are distributed among the streaming multiprocessors. All blocks of a grid contain the same amount of threads and execute the same program concurrently. When a kernel function is launched, the GPU creates a grid and the required amount of thread blocks and begins distributing thread blocks to available processing cores. For device of compute capability 2.0, each thread block can consist of up to 1024 threads if not restricted by any other hardware constraints and each grid can contain thousands of blocks. CUDA programs can also access the block index as well as the block and grid dimensions in addition to the thread index, which can be used to uniquely identify a thread within the grid. Referring back to the raytracer example, 1024 threads might not be enough to raytrace an image of acceptable resolution if each thread corresponds to one pixel. Therefore, the calculation must be spread across several thread blocks and processors. If the raytraced image is to be stored in a two-dimensional array in global memory, the memory address where each thread stores its result can be calculated using the thread and block index as well as the block dimension.

Grids belong to a context. A context can consist of many grids with different dimensions that execute distinct programs. Up to 16 grids of the same context can execute concurrently. All grids of the same context share a common virtual address space, thus threads belonging to different contexts cannot access each others memory locations. While there can be several contexts created on the device, only one context can be active at a time. Even though it is not explicitly stated by the CUDA documentation, it is safe to assume that grid execution and therefore also context execution cannot be preempted; Nvidia recently announced that preemption will be a feature supported by future GPUs[5].

2.3.2 Memory Types

CUDA programs have access to all of the different memory types mentioned in section 2.2.3. We now revisit the topic of memories and caches from the software point of view and look at the restrictions and performance traits of each type of memory.

Figure 2.6 gives an overview of some of the properties of the available CUDA memory types. Registers, special registers, and shared memory are located within the streaming

---

[5] http://www.hexus.net/content/item.php?item=26552

Type	Access	Scope	Location	Cached
Registers	R/W	Thread	SM	No
Special Registers	Read	Thread	SM	No
Shared Memory	R/W	CTA	SM	No
Local Memory	R/W	Thread	DRAM	Yes
Global Memory	R/W	Context	DRAM	Yes
Constant Memory	Read	Context	DRAM	Yes
Texture Memory	Read	Context	DRAM	Yes

Figure 2.6: Properties of CUDA Memory Types

multiprocessors and are thus only accessible by the SM they belong to. These low-latency memories are not cached, because access times are generally low. On the other hand, the local, global, constant, and texture memories are all distinct sections of DRAM memory located off the chip. Accessing DRAM is at least an order of magnitude slower than accessing on-chip memory, therefore caches are used to amortize the costs.

Whenever threads of the same warp write to the same memory location, the results are unpredictable; the behavior even depends on the compute capabilities of the device the application is executed on. For devices based on the Fermi architecture, only one thread performs the write, but which one is undefined [3, G.4.2]. If more than one thread of the same warp performs an atomic read-modify-write on the same memory location, all operations are serially executed, but the order in which the operations are performed is undefined [3, 4.1].

Registers. When a thread is allocated on a streaming multiprocessor, the SM assigns the required amount of registers to the thread, which are subsequently accessible by this thread only. Each thread belonging to the same grid requires the same amount of registers to execute the program. Register access is fast, but registers might be blocked after being used as an argument for a high-latency operation such as DRAM memory access. For optimal performance, a thread may execute subsequent instructions that do not depend on any of the blocked registers. If a register used by the next instruction is blocked, however, the thread and consequently the thread's warp cannot be scheduled for execution. In this case, the warp scheduler checks if there is any other warp ready to execute its next instruction and schedules this one instead. That way, latencies can be hidden if only there are enough other warps to execute in the meantime.

Special Registers. Special registers are predefined by the hardware. They convey parameters like the thread index, the block index, the grid size, and so on to a thread. Obviously, the thread index register returns a different value for each thread of the same thread block, whereas other registers like the grid size register hold the same value for all threads of the same grid. Still, even grid-scoped special registers are duplicated, because threads of the same grid are usually executed on many different streaming multiprocessors, each of which provides its own copy of the register.

Shared Memory. A streaming multiprocessor dynamically partitions its shared memory and assigns one of those partitions to each of its thread blocks. Threads of the same thread block can then share data through their assigned shared memory partition but are unable to access any other partition on the same or another streaming multiprocessor. CUDA does not attempt to protect the developer from the problems that can occur due to conflicting access to shared memory. Developers must manually take care of race conditions, read-after-write hazards and the like using atomic operations, thread barriers, or memory fences.

Access to shared memory is almost as fast as accessing a register. Therefore, it might be beneficial to copy data located in global memory to shared memory if it is accessed several times by several different threads of the same block.

Shared memory has 32 banks on current GPUs. Banks are interleaved, more precisely, successive 32 bit words are assigned to successive banks. If all threads of a warp access locations in shared memory with locations residing in different banks, all 32 reads or writes can be performed in parallel. However, if threads access locations where two or more locations reside in the same bank, there is a bank conflict and access has to be serialized. For example, if all threads of a warp read data from successive indices of a 64 bit double precision floating point array in shared memory, the reads of threads belonging to the first half-warp are in conflict with the reads of the threads belonging to the second half-warp. Consequently, the reads are serialized and serviced by two independent shared memory accesses. Avoiding bank conflicts has the potential of significant performance improvements.

Local Memory. Local memory resides in DRAM and is slow to access. To improve latency, threads can use the streaming multiprocessors' L1 caches and the global L2 cache to speed up reads and writes to local memory. A read of a memory address already cached in L1 has a latency that is almost as low as reading a register.

Each thread has its own local memory space and has no access to the local memory of other threads. The local memory is used by the compiler to spill registers to memory when the thread requires more registers than can be reasonably assigned to it. Moreover, the function call stack resides in local memory. Additionally, a thread may use local memory to store other private data that does not fit into registers such as arrays.

With thousands of threads in-flight, the amount of local memory allocated to each thread consumes a significant amount of DRAM. In the case of the GeForce GTX 480, if all streaming multiprocessors were fully utilized and all threads had the maximum amount of local memory allocated, the amount of memory required would be 11.25 GByte[6], which is an order of magnitude more than the available amount of DRAM on the graphics card. The specification does not state what the CUDA runtime does to remedy this situation, so we have to assume that less threads are launched to prevent the program from running out of memory. All in all, the amount of global memory available to a CUDA program is the amount of DRAM memory sans the amount of constant memory and reserved local memory. Consequently, the amount of global memory available to a CUDA program is unknown statically; we only know it is less than the amount of DRAM on the board.

Global Memory. Global memory is shared by all threads of the same context and is not sequentially consistent [9, 5.1.4]. Like for local memory, the streaming multiprocessors' L1 caches and the global L2 cache speed up accesses to global memory. Conflicting writes to the same address by different threads result in undefined behavior and CUDA does not supply any mechanism for global thread synchronization. The only way to synchronize memory access is thread block synchronization or launching dependent grids within the same stream on the CPU-side. Other than that, atomic operations or memory fences might be helpful in avoiding some of the common shared data issues.

---

[6] 15 streaming multiprocessors times 1536 threads per SM times 512 KByte of local memory per thread

The L1 caches of different streaming multiprocessors are not kept coherent for global memory locations. This can result in a thread reading a stale value from a memory location that has long been updated by another thread [9, table 80]: Suppose thread 1 on SM 1 writes the value a to global memory location l. On SM 2, thread 2 reads global memory location l after the effect of thread 1's write is visible to all other threads. If a thread on SM 2 has previously accessed location l and the value stored at l is still cached in the L1 cache, thread 2's read operation returns the stale value from the cache. Then again, this coherency problem only makes an unpredictable situation even more unpredictable, i.e. even without the stale data problem it cannot be statically known whether thread 2 reads the old or the new value at location l. Thread 2 might read location l before thread 1 even executes the write command, or before thread 1's write command is fully processed by the memory subsystem. In that case, thread 2 reads the old value. If thread 2 executes the read command after thread 1's write command is fully processed, thread 2 might get the new value or the old value depending on whether the L1 cache contains a valid cache line for address l. All in all, reading and writing the same global memory address from different threads should be avoided.

Constant Memory. Constant memory is also located in DRAM and cached by several constant caches [11, IV.J]. Threads can only read constant memory locations; the CPU can also write to constant memory. Before Fermi introduced L1 and L2 caches for local and global memories, access to constant memory was generally faster because of the constant caches that already exited on previous generation devices. Even with cached global and local memory operations, there are still some hardware optimizations that might make reads of constant memory faster than global or local reads.

Constant memory is partitioned into eleven 64 KByte banks on current GPUs. Bank zero is used for all statically sized and allocated constant variables, whereas the other banks are used to support the usage of constant arrays whose size is not known at compile-time. We neglect the concept of constant banks entirely, because allocations of constant memory occur on the host side of a program and are therefore outside the scope of the semantics. At runtime, constant banks introduce no semantically interesting features, as the only complications that arise are some advanced address calculations that are necessary to access different arrays within the same bank. The semantics fully supports any kind of address calculations for all types of addressable memory, so constant banks add nothing new.

Texture Memory. In graphics mode, the GPU typically samples thousands of texels each frame. What makes texture memory special is the layout of the texture data in the DRAM and in the texture caches. Since threads of the same warp typically sample texels that are close together in 2D, the texture memory and caches are optimized for 2D spatial locality [3, 5.3.2.5]. A CUDA application might benefit from this optimization if its data is organized appropriately.

2.3.3 Introduction to PTX

The parallel thread execution virtual machine and instruction set architecture, from now on referred to as PTX, is designed to allow efficient execution of general purpose parallel programs on Nvidia GPUs. PTX provides a stable instruction set spanning several compute capability versions and abstracts from the hardware instruction set of the target device. As already mentioned in section 2.2.4, devices of different major compute capability versions have a different core architectures, whose hardware instruction set might differ. PTX makes low-level programming possible without tying the program to a specific compute capability version. Section 2.3.4 explains in greater detail how PTX programs are compiled down to the hardware instruction set.

The `squareArray` kernel of program 2.1 is written in CUDA-C, an extension to ASNI C developed by Nvidia. As the formal semantics developed in this report is close to the hardware, C is already too high-level a language to base the semantics on. We therefore define the formal semantics on an extended subset of PTX. This section focuses on the basic features and characteristics of PTX; more detailed information about PTX can be found in the PTX specification [9].

```
1  .version 2.1
2  .target sm_20
3
4  .entry squareArray (.param .u32 a)
5  {
6    .reg .u32 arrayIndex;
7    .reg .u32 address;
8    .reg .f32 value;
9
10   ld .param .u32 address, [a];
11   mul .u32 arrayIndex, %tid.x, 4;
12   add .u32 address, address, arrayIndex;
13   ld .global .f32 value, [address];
14   mul .f32 value, value, value;
15   st .global .f32 [address], value;
16   exit;
17 }
```

Listing 2.2: PTX version of program 2.1's `squareArray` kernel

Program 2.2 is the PTX version of the `squareArray` kernel of program 2.1 already discussed earlier. The `.version` and `.target` directives inform the PTX compiler about the PTX language version and the target architecture the program is intended to run on. All 1.x PTX programs can be compiled for all `sm_2x` targets because of the backward compatibility, the reverse is not possible.

In line 4, the entry point `squareArray` is defined. Each PTX program can define multiple entry points, so the same PTX program can be used to launch different kernels on the GPU. Like its C counterpart, the kernel has one parameter, `a`. In the C version, `a` is a pointer to a floating point array in global memory. In PTX, however, `a` is an unsigned integer in the parameter state space where the value of the pointer to global memory is stored. Function input and return parameters are stored in registers or in local memory depending on hardware constraints, whether the function is or might be called recursively during the execution of the program, and what is most optimal in terms of performance. It is up to the PTX compiler to decide where the parameters are actually located. In the case of kernel parameters, though, the actual storage location of input parameters is always constant memory [3, B.1.4]. In line 10, the `ld` instruction is used to load the value at address `a` into register `address`. In this case, the compiler replaces the `.param` parameter of the `ld` operation with `.const`, because the `.param` value `a` loaded by this operation lies in constant memory. The address stored in register `address` is the location of the floating point array in global memory. In order to access the array at the index corresponding to the thread's thread index, the thread index stored in special register `%tid.x` is first multiplied by 4 in line 11 and then added to the starting address of the array in line 12. This mimics the semantics of the C statement `a[idx]`: To the location of `a`, add `idx` times the size of one element stored in the array. The multiplication with the element size is implicit in C, but explicit in PTX. The size of a floating point value is 4 bytes.

In line 13, the program loads the array's value stored at the computed address in global memory and squares the loaded value in line 14. The computed value is subsequently written back to the same address in global memory in line 15. The `exit` statement in the next line signals the termination of the threads to the warp.

PTX supports five basic types: signed integers, unsigned integers, floating point values, bit (untyped) values, and predicates. Except for predicates, each type can be used with different sizes. For instance, `.f32` denotes a 32 bit floating point value and `.b64` specifies a 64 bit untyped value. There are complex conversion rules between the different types, defined in [9, Table 13]. Furthermore, registers are virtual in PTX, so even though we defined two registers `arrayIndex` and `address` in lines 6 and 7, the compiler might decide to use only one physical register to improve resource utilization on the hardware.

In the following chapters, the formal semantics is defined on an extended subset of PTX. From the features discussed in this section, we ignore data types, the parameter state space, and virtual registers and assume that type checks, parameter state space resolution, and virtual register to physical register mapping have already been performed by a compiler. This allows us to focus on the core semantical issues without being overwhelmed by PTX' convenience features. The instructions and features supported by the semantics can be found in section 5.1.

2.3.4 Compilation Infrastructure

Nvidia supplies a compiler that separates host and device code of a CUDA-C file and compiles each of those either for the CPU or for the GPU. The device code can either be compiled into PTX or into binary GPU code. Binary code is architecture specific, meaning that code compiled for a device of compute capability $x.y$ can only be run on devices of compute capability $x.z$ with $z \geq y$. If a CUDA application has PTX code embedded into its executable file instead, a just-in-time compiler compiles the PTX program into binary code at runtime. This approach has several advantages. First of all, the CUDA application also supports future hardware that was not yet available at compile time, as a PTX program for some specific compute capability can always be compiled for a device of equal or greater compute capability. Moreover, future compiler updates might generate more efficient code, so the application's performance might increase without the need to recompile and redeploy the application. On the other hand, just-in-time compilation increases application load time [3, 3.1]. Hybrid approaches can be used as well, i.e. code can be compiled ahead-of-time for some devices and just-in-time for others.

Nvidia has recently released a first beta version of cuobjdump disassembler, a tool that decompiles binary code into a PTX-like assembly language. At the time of this writing, cuobjdump disassembler can only be downloaded by registered Nvidia developers, and decompilation of programs compiled for compute capability 2.x targets is not yet supported. Still, some valuable insights can be gained by inspecting the binary code produced by the PTX compiler for 1.x targets. Some of the assumptions that we make in later chapters are based on observations made using the cuobjdump disassembler.

2.4 Alternatives to CUDA

CUDA is a Nvidia exclusive technology, so only Nvidia GPUs are capable of executing CUDA applications. While this exclusivity is beneficial in that the application programming model can be fully optimized for the hardware, it makes it impossible to support other parallel processors such as AMD GPUs or IBM's Cell chip. Basically, an application supporting both AMD and Nvidia GPUs would have to incorporate two versions of any kernel it executes: One written in CUDA, and one written for the Compute Abstraction Layer (CAL), AMD's CUDA equivalent [12][7]. Both the Khronos Group and Microsoft released cross vendor parallel computing frameworks to remedy this situation. The remainder of this section briefly compares CUDA to those two competing technologies.

2.4.1 OpenCL

Apple initiated the development of OpenCL. Like the OpenGL graphics API, OpenCL is an open standard developed by the Khronos Group. The specification [13] currently specifies OpenCL version 1.1, which has many similarities to CUDA for devices of compute capability 1.x. Newer CUDA features such as recursion are not yet supported by OpenCL.

CUDA	OpenCL
Grid	NDRange
Thread Block	Work Group
Thread	Work Item
Streaming Multiprocessor	Compute Unit
CUDA Core	Processing Element
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory

Figure 2.7: Mapping Between CUDA and OpenCL Terminology

Figure 2.7 shows how several CUDA concepts can be mapped directly to their OpenCL counterparts [2, chapter 11]. OpenCL's thread and memory hierarchies as well as the underlying hardware model are closely related to CUDA's ones. This is to be expected, however, as CUDA's and OpenCL's main goals are to achieve the highest possible performance. Therefore, the specification must respect the characteristics of the underlying hardware. In contrast to CUDA, the underlying hardware can also be ATI GPUs, x86 CPUs, or IBM's Cell chip [13, 12].

OpenCL defines a programming language similar to C that is used to develop OpenCL kernels. There is no standalone compiler to create binary code for an OpenCL kernel. The code of some OpenCL kernel is stored within the host program's executable as a string. Before a kernel can be executed on an OpenCL device, it has to be compiled at runtime. Nvidia's OpenCL drivers compile a OpenCL program into PTX [14, 2.2.1], but are currently unable to cache the compiled program to emulate CUDA's precompilation support [15, 10].

---

[7] All referenced AMD documents can be found at `http://developer.amd.com/gpu/ATIStreamSDK/pages/Documentation.aspx`, last accessed 2010-10-15.

In a way, the semantics defined in this report also define a semantics for OpenCL, but we do not check if Nvidia's compilation process fully preserves OpenCL's semantics. This is actually unlikely, as OpenCL does not expose the SIMT execution model of Nvidia GPUs [16, 3] which can lead to unexpected program behavior as shown in chapter 6. Current ATI GPUs also organize threads in warps, called wavefronts, so the same problem should also exist for ATI GPUs [17, 1.3.2]. Whereas the warp size is 32 for all current Nvidia GPUs, the wavefront size varies between 16, 32, and 64 for current ATI GPUs [12, Glossary-A-9]. Hence, OpenCL kernels might exhibit vastly different performance characteristics depending on the hardware they are executed on. For a wavefront or warp size of n threads, performance is reduced by a factor proportional to n in the worst case. Additionally, OpenCL programs have to deal with different feature sets supported by the hardware.

2.4.2 Direct Compute

Microsoft introduced a new shader type with DirectX 11: compute shaders. Written in HLSL, a high-level C-like programming language for DirectX shaders, compute shaders are similar to CUDA and OpenCL kernels.

CUDA	Direct Compute
Grid	Dispatch
Thread Block	Thread Group
Thread	Thread
Streaming Multiprocessor	Multiprocessor
CUDA Core	Scalar Processor
Shared Memory	Group-shared Memory

Figure 2.8: Mapping Between CUDA and Direct Compute Terminology

Due to the similarities between CUDA and Direct Compute, it is again possible to map some of the CUDA concepts to their Direct Compute counterparts as shown in figure 2.8 [18]. However, there are some differences concerning memory. In Direct Compute, memory is partitioned into resources, which are either textures or buffers for arbitrary data formats. The application uses resource views for these resources, which define properties like access patterns or implicit format conversions and subsequently allow the graphics driver to optimize memory access. It is also possible to create several views for the same resource. Compute shaders operate on resource views and can use group-shared memory for thread communication within the same thread group [19].

CUDA, OpenCL and Direct Compute can be used in conjunction with graphics rendering. For computer games, Direct3D is the dominant graphics API and compute shaders can be tightly integrated into a Direct3D 11-based rendering engine. For instance, [20] shows the advantages a deferred rendering implementation using compute shaders has over a traditional pixel shader based implementation; the compute shaders' greater flexibility makes it possible to render many more lights in the scene at the same performance level. Additionally, compute shaders are already used in recent games to accelerate texture compression algorithms[8] and post processing effects such as screen space ambient occlusion[9].

---

[8] `http://www.pcgameshardware.com/aid,776086`, last access 2010-10-10

[9] `http://www.anandtech.com/show/2848/2`, last access 2010-10-10

The feature set that must be supported by compute shader compatible hardware is more strictly defined compared to the OpenCL specification. Currently there are two feature levels, one for DirectX 10 compatible graphics cards and one for DirectX 11 compatible GPUs. Once an application has verified that the hardware supports one or both of the feature sets, it can be sure that all of the corresponding compute shader features are supported by the hardware. Still, the specific implementation of a feature might expose different performance characteristics for different hardware devices. Consequently, it might be necessary to develop different implementations of the same algorithm not only because of features unsupported by some hardware, but also because of features running too slow on some hardware.

3 Conventions, Global Constants, and Rules

Due to the low-level nature of PTX and the complexity of CUDA's programming model, we have to define the domains and rules of the formal semantics in extensive detail. To help reduce the clutter, we introduce some conventions in section 3.1 that at least allow us to express the concepts as concisely as possible. Furthermore, many of the domains and functions defined in chapters 4 and 5 depend on certain parameters like the number of registers per SM or the amount of global memory available on the graphics card. We avoid having to explicitly state the required parameters for each domain and function by declaring these parameters as global constants in section 3.2 and implicitly parameterize all definitions over this set of constants.

3.1 Conventions

Throughout the remainder of this report, we use the following conventions in all formal definitions whereever appropriate:

Optional Elements and Failure Elements. We use ε and $\perp$ to denote the missing element and the failure element respectively. For some domain A, it is always true that $\varepsilon \notin A$ and $\perp \notin A$. We define the liftings $A_\varepsilon := A \cup \varepsilon$, $A_\perp := A \cup \perp$, and $A_{\varepsilon,\perp} := A \cup \varepsilon \cup \perp$. For some $a \in A$, we know that $a \neq \varepsilon$ and $a \neq \perp$, whereas for $a_\varepsilon \in A_\varepsilon$, we only know that $a_\varepsilon \neq \perp$, but a_ε might be ε or any other element of A. When we write $a \in A_\varepsilon$ or $a = f(a')$ for some function $f : A \to A_\varepsilon$, a cannot be ε, so the selection or assignment is only possible if the selected or returned value is not ε. For example, this allows us to shorten some side conditions in function definitions with case distinctions. The following two definitions of function $g : A \to A_\perp$ illustrate the usage of this convention. With function f defined as above, the following two definitions of g are equivalent:

$$g(a') = \begin{cases} a & \text{if} \quad a = f(a') \\ \perp & \text{if} \quad \varepsilon = f(a') \end{cases} \qquad\qquad g(a') = \begin{cases} a_\varepsilon & \text{if} \quad a_\varepsilon = f(a') \wedge a_\varepsilon \neq \varepsilon \\ \perp & \text{if} \quad \varepsilon = f(a') \end{cases}$$

Functions. Oftentimes, the codomain of some function includes the failure element. Usually the definitions of those functions contain case distinctions. Whenever we do not specify all cases for such a function, the remaining cases are implicitly assumed to return $\perp$. Alternatively, we use the word "otherwise" to denote a side condition that is true if the side conditions of all other cases are not true. We do not use partial functions.

It is sometimes necessary to change the value a function returns for a specific input. We establish the notation $f[a \mapsto b]$ for some function $f : A \to B$ that serves this purpose and is defined as follows:

$$f[a \mapsto b](a') = \begin{cases} b & \text{if} \quad a' = a \\ f(a') & \text{otherwise} \end{cases}$$

For convenience, we use the following notations to update several values at the same time: $f[a_1\ a_2 \mapsto b] = f[a_1 \mapsto b][a_2 \mapsto b]$, $f[a_1\ a_2 \mapsto b_1\ b_2] = f[a_1 \mapsto b_1][a_2 \mapsto b_2]$, and _ represents all input values, such that $f[\_ \mapsto b](a)$ returns b for all input values a. This is only well-defined if all updated input values are mutually distinct.

Lists. Some domains defined in the following chapters comprise sets or lists of elements of other domains. In all of these cases, we consistently use lists rather than a combination of sets and lists for reasons of clarity, even if the ordering of elements inside the list is not important. The domain A^* denotes the set of all lists with elements in A whose length is arbitrary but finite. The domain A^n denotes the set of all lists with elements in A whose length is exactly $n \in \mathbb{N}$. ε denotes the empty list. The operations on lists that we use throughout the next chapters are briefly and informally introduced in figure 3.1.

Operation	Description
$a :: \vec{a}$	Concatenates an element and a list. For convenience, we also use the concatenation operator to concatenate two lists.
$\lvert\vec{a}\rvert$	Returns the length of a list; obviously, $\lvert\vec{a}\rvert = n$ for some list $\vec{a} \in A^n$ with $n \in \mathbb{N}$.
$a \in \vec{a}$	True, if the list contains element a somewhere. This is well-defined as lists are always of finite length.
$\vec{a}_k$	Returns the element at position $k \in \mathbb{N}$ in the list. Undefined if $\lvert\vec{a}\rvert < k$.
$\vec{a}[k \mapsto a]$	Returns a new list where the element at position $k \in \mathbb{N}$ is replaced with a. This notational convention is equivalent to that of function updates, so we use the same convenience notations for multiple updates at the same time. Undefined if $\lvert\vec{a}\rvert < k$.
$a \circ \vec{a}$	Inserts a anywhere in $\vec{a}$. For convenience, we also use this operator to insert one list into another. In that case, the elements of the first list are inserted in any order and any location into the second list; in other words, the first list is not continuously inserted into the second one.
$a \odot \vec{a}$	Same as above, however, if the list already contains element a, it is not inserted again.
$\vec{a} = \bigcup_{i \in \{1 \dots n\}} a_{\varepsilon,i}$	Similar to writing $\vec{a} = a_{\varepsilon,1} \circ \dots \circ a_{\varepsilon,n}$, however, if an $a_{\varepsilon,i} = \varepsilon$, it is not inserted into the list, hence $\varepsilon \notin \vec{a}$.
$\vec{a} \setminus a$	Removes all occurrences of a from the list, thus $a \notin \vec{a} \setminus a$.
$a_1 \dots a_n$	An alternative to writing $a_1 \circ \dots \circ a_n$.

Figure 3.1: Brief overview of list operations for elements a and lists $\vec{a}$

Meta-Variables. Words starting with a capital letter denote a semantic domain. For instance, the set of two-dimensional vectors might be defined as $Vector2 = \mathbb{R} \times \mathbb{R}$. To denote a single element of a domain, we use meta-variables that are identical to the domain's name starting with a lower-case letter; in this example, *vector2*. The default meta-variable for a domain can be overridden by explicitly specifying the meta-variable when defining the domain, such as $v \in Vector2 = \mathbb{R} \times \mathbb{R}$. In this case, v denotes an element of domain *Vector2*. We also use meta-variables in variously decorated forms such as v', v'', v_0, v_1, and so on. We adorn a domain's meta-variable with an arrow at the top to denote a list of elements of the domain, for example $\vec{v} \in Vector2^*$.

Handling of Tuples. Most of the semantic domains we deal with in the following chapters consist of several sub-domains. Let $X = A \times B \times C \times D \times E^*$ be a domain and $x \in X$ be an element of that domain. To get the projection of x to one of its sub-domains, we subscript x with the meta-variable of the domain we want to project to. For instance, to get x's projection to domain D, we write $x_d = \pi_4 x$. We use the same convention if the projected sub-domain is a list, so $x_{\vec{e}} = \pi_5 x$.

It often happens that we have a complex tuple type instance like $(a, b, c, d, \vec{e}) \in X$, but in a rule or function we actually only need a few of the variables. For example, suppose we only need to use c and $\vec{e}$ in a rule. In that case, we write $x \triangleright c, \vec{e}$. This way, we can reference the entire tuple with x and do not need any projections to access x's sub-domains. When using this projection syntax, we list the meta-variables of the sub-domains in the order they were specified in the definition of X. If this notational convention causes any ambiguities, we either resort to the standard tuple notation or list all sub-domains in our projection notation.

To update only some of a tuple's values, we use a notation similar to the projection syntax, replacing $\triangleright$ with $\triangleleft$. Thus, if $x \triangleright a, b, c, d, \vec{e}$, we write $x' = x \triangleleft a', \vec{e}'$ as an alternative to $x' = (a', b, c, d, \vec{e}')$.

The projection and update syntax for tuples is used frequently throughout the rules and functions defined in the following chapters.

3.2 Global Constants

We implicitly parameterize all domains, functions, and rules defined in the following chapters over the set of global constants listed in figure 3.2. Unless the name of a constant is self-explanatory, we give a short description or further information for the constant. We also list the values of the constants based on the specification of the Fermi-based GeForce GTX 480 in order to give an idea about the typical magnitude of the constants' values. The values are collected from [3], [9], and the graphics card's product page on the Nvidia website[1].

3.3 Rules

Except for the semantics of memory programs presented in section 4.4.4, we formalize CUDA's model of computation using a structured operational semantics, i.e. a small-step semantics. We define the rules of the semantics over transition systems implicitly defined by specifying their configurations $\langle c \rangle$ and their transitions $\langle c \rangle \rightarrow^x \langle c' \rangle$. We identify each transition system defined throughout this report by its transition symbol; in the example above, $\rightarrow^x$ is the transition system's name. Transitions can optionally carry input and output values in which case we write $\xrightarrow[out]{in}{}^x$. Therefore, the general structure of some rule (r) for some

---

[1] `http://www.nvidia.com/object/product_geforce_gtx_480_us.html`, last access 2010-10-09

transition system $\to$, some configuration Γ, some Boolean side condition b, and some context function $C(\Gamma)$ is:

$$\text{(r)}\quad \frac{\Gamma \to \Gamma'}{C(\Gamma) \to C(\Gamma')} \qquad \text{if} \quad b$$

For instance, consider rule (seq$_1$) of transition system $\to^{\mathrm{Op}}$ defined in section 4.4.4. The configurations and transitions are defined as follows; there are two possible configurations and two possible transitions that might take place. Each transition outputs a yield action *ya*. Transition system $\to^{\mathrm{Op}}$ is special in the sense that it does not define a true small-step semantics; rather, it defines micro steps which resemble more of a big-step semantics and it then uses a small-step semantics to connect sequences of micro steps. A micro step ends once a statement yields execution or terminates the program.

$$\begin{aligned} \mathit{Configurations} &: \langle \mathit{stm}, \eta, \sigma \rangle \in \mathit{MemStm} \times \mathit{MemEnv} \times \Sigma, \qquad \langle \eta, \sigma \rangle \in \mathit{MemEnv} \times \Sigma \\ \mathit{Transitions} &: \langle \mathit{stm}, \eta, \sigma \rangle \xrightarrow[\mathit{ya}]{}{}^{\mathrm{Op}} \langle \mathit{stm}', \eta', \sigma' \rangle, \qquad \langle \mathit{stm}, \eta, \sigma \rangle \xrightarrow[\mathit{ya}]{}{}^{\mathrm{Op}} \langle \eta', \sigma' \rangle \end{aligned}$$

Rule (seq$_1$) executes the first statement of a sequential composition which yields program execution. There are two possibilities: Either, the first statement is completed or its remaining statements have to be executed during the next micro step.

$$\text{(seq}_{1a}\text{)}\quad \frac{\langle \mathit{stm}_1, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}{}^{\mathrm{Op}} \langle \mathit{stm}_1', \eta', \sigma' \rangle}{\langle \mathit{stm}_1\ \mathit{stm}_2, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}{}^{\mathrm{Op}} \langle \mathit{stm}_1'\ \mathit{stm}_2, \eta', \sigma' \rangle}$$

$$\text{(seq}_{1b}\text{)}\quad \frac{\langle \mathit{stm}_1, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}{}^{\mathrm{Op}} \langle \eta', \sigma' \rangle}{\langle \mathit{stm}_1\ \mathit{stm}_2, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}{}^{\mathrm{Op}} \langle \mathit{stm}_2, \eta', \sigma' \rangle}$$

We combine these two rules into one rule that handles both cases by using a notational convention: $\langle [\mathit{stm}_1']\ \mathit{stm}_2, \eta', \sigma' \rangle$ represents a configuration of both configuration types depending on whether the first statement is fully processed during the current micro step. Thus, rules (seq$_{1a}$) and (seq$_{1b}$) can be combined into:

$$\text{(seq}_{1}\text{)}\quad \frac{\langle \mathit{stm}_1, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}{}^{\mathrm{Op}} \langle [\mathit{stm}_1',]\ \eta', \sigma' \rangle}{\langle \mathit{stm}_1\ \mathit{stm}_2, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}{}^{\mathrm{Op}} \langle [\mathit{stm}_1']\ \mathit{stm}_2, \eta', \sigma' \rangle}$$

We define Γ_x to be the set of infinite sequences of execution steps of transition system $\to^x$. For example, $\langle \mathit{stm}_1, \eta_1, \sigma_1 \rangle \xrightarrow[\mathit{ya}_1]{}{}^{\mathrm{Op}} \langle \mathit{stm}_2, \eta_2, \sigma_2 \rangle \xrightarrow[\mathit{ya}_2]{}{}^{\mathrm{Op}} \ldots \in \Gamma_{\mathrm{Op}}$. Obviously, runs of terminating memory programs are not elements of Γ_{Op}.

Constant	GTX 480	Remark
$MaxBlocksPerGrid$	$< 2^{32}$	The maximum number of blocks per grid. The actual number might be lower due to other hardware constraints. The specification does not state a specific value.
$MaxThreadsPerBlock$	1024	The maximum number of threads per block. The actual number might be lower due to other hardware constraints.
$MaxGridSize_{x,y,z}$	65535, 65535, 1	Currently, the value of z must be 1 [3, 2.2].
$MaxBlockSize_{x,y,z}$	1024, 1024, 64	
$WarpSize$	32	The number of threads per warp.
$MaxResidentBlocks$	8	The maximum number of resident blocks per SM.
$MaxResidentThreads$	1536	The maximum number of resident threads per SM; the maximum number of resident warps per SM is therefore $\frac{MaxResidentThreads}{WarpSize} = \frac{1536}{32} = 48$.
$NumBarrierIndices$	16	The number of barriers that can be used by each thread block.
$NumProcessors$	15	The number of streaming multiprocessors on the chip.
$NumCores$	32	The number of CUDA cores per SM.
$NumConWarps$	2	The maximum number of warps that can be executed concurrently on each SM.
$NumConKernels$	16	The maximum number of kernels that can be executed concurrently on the GPU.
$MaxProgSize$	2 million	The maximum number of instructions per kernel.
$MaxRegSize$	4 Byte	The maximum register size in bytes.
$MaxMemOpSize$	128 Byte	The maximum memory request size in bytes.
$SharedMemBanks$	32	
$GlobalMemSize$	< 1536 MByte	
$LocalMemSize$	512 KByte	The amount of local memory per thread.
$ConstMemSize$	64 KByte	
$SharedMemSize$	16 or 48 KByte	The amount of shared memory per SM.
$RegFileSize$	32768	The number of registers per SM.
$Level2CacheSize$	768 KByte	
$Level1CacheSize$	16 or 48 KByte	The size of the L1 cache per SM.

Figure 3.2: Global Constants Used Throughout the Semantics

4 Formal Memory Model

CUDA features a complex memory model whose semantics are only implicitly and partially documented. We base the following discussions on what is said about memory in [3], [9], [7], and [5]. We explicitly state whenever we have to interpret or knowingly deviate from the informal specification of the memory model in those documents. Additionally, we incorporate the information found in patents [21], [22], and [23]. These patents have been released quite recently and fit our general understanding of the memory model gained from the CUDA specification. We assume that the patents describe the actual hardware implementation, although we cannot be certain about that. But as it is unlikely that Nvidia releases patents so early with no hardware yet or at all that puts the patented techniques to use, we feel it is safe to use the information provided by these patents whenever the other documents remain too vague.

The memory model is complex but plays an important part in developing and optimizing CUDA applications. Furthermore, it is unlike the x86 memory model developers have grown accustomed to. Therefore, we intend to formalize the memory model as precisely as possible given the limited amount of information available. With the exception of textures, all types of memories introduced in chapters 2.2.3 and 2.3.2 are discussed in the subsequent sections. We ignore textures because the GPU performs several optimizations to the layout of the data in memory which are entirely unspecified. Additionally, texture address calculations and filtering operations would complicate the formalization but add little to the deeper understanding of the semantics of CUDA programs. For a similar reason, we do not formalize the constant cache. In contrast to what [3] suggests, there seem to be three constant caches on the GPU [11, IV.J], some of which appear to be used as instruction caches as well. Moreover, as data in constant memory does not change during the execution of a program, cached reads of constant memory are fully transparent to the program; the program cannot even specify whether it wants to use the caches when fetching a constant value. Consequently, we do not formalize the constant caches for reasons of brevity. On the other hand, the streaming multiprocessors' L1 caches do have an impact on program semantics and are therefore included in the formalization. The L2 cache is formalized to fill the gap between the L1 caches and DRAM. Furthermore, PTX programs can specify a cache operation that should be used to read or write a value from or to global and local memory. The operations currently supported by the hardware and PTX are listed in [9, Table 80] and [9, Table 81]. The cache operation specified by a load or store instruction determines the eviction policy used to decide if the cache line containing the requested address can be evicted, whether the data should be retrieved from or written to the cache or memory directly, and whether the data is evicted immediately after the current request completes. Embedding all of this behavior into a single transition system would result in fairly verbose and complex rules; it is impossible to define self-contained rule systems for each cache and memory type as the dependencies and interactions between them are complex. Instead, we opt for another approach where we develop a programming language that we in turn use to define the semantics of the memory model. We give a brief introduction to the features of the language in section 4.1 and formally define its syntax and semantics throughout the rest of this chapter.

We have to define some basic domains first before we are able to concentrate on the formalization of the semantics. To uniquely identify in-flight memory operations, we define an abstract domain of memory operation indices and assign a unique element of this domain to each memory operation.

$$mid \in MemOpIdx$$

Furthermore, we allow byte-by-byte access to all memory types. The amount of bytes read or written by one request ranges between one byte and *MaxMemOpSize* bytes. The global constant *MaxMemOpSize* determines the maximum amount of bytes that can be read from or written to memory by a single request.

$$size \in MemOpSize = \{1, \ldots, MaxMemOpSize\}$$

However, this approach is in contrast to the actual hardware implementation of certain memory operations. For instance, global memory accesses are always serviced with either 32 or 128 byte transactions [10, G.4.2]. Still, our formalization of the memory model is indeed capable of simulating the hardware's actual behavior; in the case of global memory accesses by using a 128 byte aligned base address and a memory operation size of 128 byte. We later introduce an abstract translator function that translates memory requests issued by threads into memory operations. One of the function's responsibilities is to ensure that the correct alignments and memory sizes are used to access memory. Additionally, the function coalesces memory requests of threads of the same warp into fewer, but larger ones in accordance with the rules outlined in [3, G.4]. We do not give a concrete implementation of this function as the list of coalescence rules found in the specification is not exhaustive.

In a similar fashion, we define the domain of data words. Data words represent values that are passed around in PTX programs, hence we assume that each register is big enough to hold an entire data word. In reality, registers can only store 32 bits however. For larger data types like double precision floating point values, PTX transparently uses two registers to store the data. Since we already rely on the compiler to correctly handle register assignment, we avoid further complicating the formalization of the memory model by abstracting from data types and sizes and by assuming that all registers are large enough to store all values. Also, a register can be either in a ready or blocked state; blocked registers cannot be accessed by their threads because they are used by some pending memory operation as either source or destination registers.

$$b \in Byte = \{0,1\}^8 \qquad RegState = \{\ \mathrm{ready},\ \mathrm{blocked}\ \}$$

$$d \in DataWord = Byte^{MaxRegSize} \qquad RegValue = DataWord \times RegState$$

Threads perform their address calculations in the virtual address space of their context. When a memory request is translated into a memory operation, the virtual address or addresses requested by the threads are translated into physical addresses. We define the abstract domain *PhysMemAddr* that we use to address a location in any of the different memory types. The smallest addressable unit is one byte for global, local, shared, and constant memory and one register for the register file. From the point of view of PTX programs, virtual and physical addresses are just data, hence the domain of physical memory addresses is a subset of the domain of data words.

$$pAddr \in PhysMemAddr \subseteq DataWord$$

The next section informally introduces the aforementioned memory programs and gives a brief overview of the memory model's basic structure. With this in mind, we formalize the caches and the memory environment in sections 4.2 and 4.3, respectively. These sections include all of the functions that are needed to define the semantics of memory programs in section 4.4. Subsequently, section 4.6 puts all these different pieces together to shape the overall semantics of the CUDA memory model.

4.1 Overview of Memory Programs

As already mentioned above, we define the semantics for a language that we in turn use to define the semantics of the memory model instead of using the traditional rule-based approach. Using memory programs in favor of large rule systems simplifies the formalization and allows us to add support for new memory operations without affecting previously defined programs. This is especially interesting because we do not give program implementations for volatile memory accesses as the specification is severely lacking information about the interplay of volatile accesses and caches. But if Nvidia ever releases more detailed information about this subject, it will only be a matter of writing some memory programs for volatile memory operations, leaving the rest of the memory model unaffected.

The memory programs declaratively define what each memory operations does and in which order caches and memories are accessed. They invoke mathematically defined functions that specify how caches and memories are manipulated and if the state manipulation is actually possible; it might be possible, it might not be possible at all, or it might be possible but not right now. We use a mechanism similar to cooperative multithreading to support the case that manipulating a cache or memory is not possible at the moment but might eventually become possible later on. When threads execute a PTX program, our memory model formalization launches an instance of a corresponding memory program for each memory operation performed by the threads. Only one memory program is executed at a time. Distinct memory operations are not processed concurrently but interleaved; the order in which distinct programs are processed is mostly undefined. Moreover, a program is free to yield execution at any time, allowing another program to continue. All of this is illustrated by program 4.1 that we use to give an informal overview of the semantics of memory programs. The overview is intended to demonstrate the context for the mathematical definitions in the subsequent sections where we begin to formally define the semantics of memory programs and the underlying mathematical framework. The semantics of the memory model is inspired by Esterel's constructive behavioral semantics [24] in the sense that each program executes an uninterruptible series of instructions until it yields execution. The semantics as a whole is defined by arbitrarily choosing and executing some unfinished programs during each step.

The memory program presented in listing 4.1 defines the semantics of a store operation to global memory with the `.wb` cache operation. When some threads of a warp execute a `st.global.wb` statement, one or more instances of program 4.1 are launched to service the request; whether one or more program instances are launched depends on whether the requests issued by the threads can all be coalesced into a single memory transaction. The order in which different memory programs are processed is generally undefined — except for volatile reads and writes as explained later. Each program instance implicitly carries some state that includes the requested program address, the amount of bytes to read or write, the

```
1 discardL1;
2 yield;
3
4 writeL2;
5 classifyNormalL2;
6 releaseRegs;
7 end;
```

Listing 4.1: Memory program for global writes using the `.wb` cache operation

read or written data, the index of the processor the threads that issued the request belong to, and more.

Program execution begins at line 1 where the L1 cache is instructed to discard any matching cache line. Writes to global memory are not cached in L1 because the L1 caches of different SMs are not kept coherent as mentioned in section 2.3.2. The discard operation removes a matching cache line from the cache, even if the cache line contains dirty data that would have to be written back first. However, this is not a problem in this situation: As global writes never write to L1, the cache line cannot be dirty, and even if it were, the data would be overwritten anyway. If the discard is successful or if the address is not cached, the memory program executes the `yield` statement which causes the memory subsystem to stop executing the program for the moment. Other programs can be executed now; we allow this to happen because the specification does not give any guarantees concerning the atomicity of memory operations. At some later point in time, program 4.1 continues execution at line 4. However, it is also possible that the discard operation fails, because the cache line is pinned by at least one other memory operation, meaning that it cannot be evicted right now. In this case, the memory program performs an implicit `yield` and tries to execute line 1 again at a later point in time. The program attempts to execute the discard operation until it eventually succeeds and normal program execution continues. Once the discard and the subsequent `yield` complete, the data is written to the L2 cache in line 4. Again, this operation can succeed immediately or cause an implicit `yield`. If the write eventually succeeds, the cache line's eviction class is set to normal, all registers blocked by the operation are released, and the program terminates. If the write to the L2 cache is instantaneous, the last three lines are executed atomically and cannot be interrupted by any other memory program.

4.2 Formalization of Caches

The CUDA documentation [3] mentions several caches that are present on the GPU: the global L2 cache and the constant, texture, and L1 caches found on each streaming multiprocessor. In reality, however, it appears that there are three levels of constant caches, two levels of texture caches, plus three levels of instruction caches, all of which have varied associativity characteristics and cache line sizes [11]. For the reasons mentioned above, we do not formalize texture memory and therefore we also ignore the texture caches. Neither do we consider the constant and instruction caches, as these only affect execution performance but not program behavior. We only look at the L1 caches found in each streaming multiprocessor and the global L2 cache, all of which can be used to speed up reads from and writes to global and local memory. Since the L1 caches are not kept coherent as mentioned before, they might affect program behavior and are thus included in our formalization.

A cache stores a copy of a small section of memory in a cache line. When a memory read request is issued by some threads and this request is to be cached, the cache checks whether it can service the request by checking if there is a cache line which contains a copy of the requested memory location. If so, there is a cache hit and the request does not have to go to a higher cache or to memory. In this case, the latency of the memory operation is up to an order of magnitude lower than in the case of a cache miss, where another cache or the memory itself must service the request. The value fetched from memory or a higher cache as the result of a cache miss is stored in a cache line so that subsequent requests to the same memory location can be serviced more efficiently. However, after some time all cache lines are occupied, so a cache line has to be cleared before a more recently requested location can be cached. The policy used to evict cache lines can severely affect performance, as it must avoid to evict cache lines that will most likely be requested again later on. For this reason, PTX programs can provide hints as to the likeliness of a subsequent request to the same address. Furthermore, requests can choose to circumvent the L1 cache or both the L1 cache and the L2 cache altogether.

We define a set of domains and operations to formalize the behavior of the caches. We leave some operations abstract, as we do not want to deal with complexities like cache associativity and eviction rules; again, these features only affect performance. We also use the same formalization for both the L1 and the L2 caches because no differences in functionality can be deduced from [3] and [9]. We first explain the structure of the caches before we formalize the supported operations.

4.2.1 Cache Lines

We define all cache operations on lists of cache lines; that way, we can easily describe the L1 and L2 caches as lists of cache lines of distinct fixed sizes and use the same operations for both types of caches.

For each cache line, we have to know whether the cache line is dirty; a set dirty flag indicates that a new value has been written to the cache line that has not yet been written back to a higher cache or memory. Obviously, the cache should not evict a dirty cache line without writing back the new value first, as this would result in a loss of data. Additionally, each cache line has a valid flag that indicates whether the data stored in the cache line is valid. Using the valid flag, we can reserve cache lines. Suppose a memory request comes in, a matching cache line is not found and the value has to be retrieved elsewhere in the hierarchy. Before the cache issues the command to retrieve the value, it first reserves and empty cache line that it can use to store the retrieved value. Otherwise, the cache would have to queue retrieved data and copy it into an empty cache line as soon as one becomes available. This would be problematic, however, because subsequent requests to the same address would all be cache misses too and unnecessary additional request to the same memory location would be propagated up the hierarchy. Hence, the cache reserves a cache line first before it asks the memory subsystem to get the value and sets the cache line's valid flag to false. Subsequent requests to the same address that arrive before the data was retrieved from the memory subsystem are blocked until the cache line's data is valid and are later serviced directly from the cache. Formally, we define the dirty and valid flags as

$$Dirty = \mathbb{B} \qquad\qquad Valid = \mathbb{B}$$

As mentioned above, we do not formalize the eviction policy used by the caches, although [22] does give a rough idea of the algorithm used. However, we do pay attention to a cache line's eviction class, which is either first, normal, or last. Cache lines of class first are evicted before cache lines of class normal are evicted, if there are any clean cache lines of the first class. Some cache operations supported by PTX change a cache line's classification, hence we support it in our formalization; moreover, for some of the supported cache operations, the reclassification of cache lines is the defining feature. In contrast to [22], we do not support the class last because [9, Table 80] and [9, Table 81] do not mention this class being used by any of the cache operations. We assume this class is relevant in graphics mode only.

$$ec \in EvClass ::= \text{first} \mid \text{normal}$$

We assume that each cache line is big enough to store the largest possible memory request. In reality, however, this might not be the case, i.e. even though the largest request might access 128 byte of memory, the cache might only be able to store 64 byte per cache line. So this obviously only works if the function that translates memory accesses of threads into memory operations is aware of this limitations and issues two 64 byte requests instead of one 128 byte request to the cache and memory, for example. Conversely, writing smaller values into the cache line is unproblematic.

$$cw \in CacheWord = Byte^{MaxMemOpSize}$$

Consequently, a cache line stores the state of its dirty and valid flags, its eviction class, optionally some data, the address of the chunk of memory that is cached, and a set of memory operations waiting for the cache line. Whenever a memory operation requests an address that is not cached but is already being retrieved from memory, i.e. there is a cache line with a matching address whose valid flag is set to false, the memory operation's index is stored in the cache line. The cache line cannot be evicted before all pending memory operations waiting for the cache line to become valid have seen the value. In [22], a pinned flag is used to handle this situation. For technical reasons, a Boolean flag does not suffice in our formalization.

$$cl \in CacheLine = PhysMemAddr_{\varepsilon} \times CacheWord_{\varepsilon} \times Dirty \times Valid \times MemOpIdx^{*} \times EvClass$$

Since we do not consider cache associativity — cache line lookup is hidden behind an abstract function —, we can now define the L1 and L2 caches as fixed sized lists of cache lines and can use the same subsequently defined functions to describe their behavior:

$$L1Cache = CacheLine^{Level1CacheSize} \qquad L2Cache = CacheLine^{Level2CacheSize}$$

We use the constant *init* to designate the default state of an unused cache line. Neither data, a program address, nor any memory indices are stored, the eviction class is normal and the cache line is marked to be clean and invalid.

$$init = (\varepsilon, \varepsilon, \text{false}, \text{false}, \varepsilon, \text{normal})$$

4.2.2 Cache Operations

This section presents the operations supported by the caches. We first present some helper functions that we consecutively make use of to define the actual cache operations. We briefly explain the intention behind each function and then give a formal definition and concrete implementation of the function itself — unless the function is abstract, of course.

Helper Functions

The following two functions *unused* and *allUsed* decide whether a single cache line is unused and whether all of the cache lines are in use, respectively. The cache line's program address determines if the cache line is in use; if the program address field is empty, the cache line is unused.

$$unused : CacheLine \to \mathbb{B} \qquad allUsed : CacheLine^{*} \to \mathbb{B}$$

$$unused(cl) \Leftrightarrow cl_{pAddr_{\varepsilon}} = \varepsilon \qquad allUsed(\overrightarrow{cl}) \Leftrightarrow \neg\exists cl \in \overrightarrow{cl}\ .\ unused(cl)$$

A cache line is pinned if its list of memory indices is not empty. The functions *pin* and *unpin* are conveniently defined to add or remove a memory index from the cache line's list of indices. A memory index is only added to the list, however, if it is not already present so the list of memory operation indices is duplicate-free.

$$unpin : CacheLine \times MemOpIdx \to CacheLine \qquad pin : CacheLine \times MemOpIdx \to CacheLine$$

$$unpin(cl \blacktriangleright \overrightarrow{mid}, mid) = cl \blacktriangleleft \overrightarrow{mid} \setminus mid \qquad pin(cl \blacktriangleright \overrightarrow{mid}, mid) = cl \blacktriangleleft mid \odot \overrightarrow{mid}$$

Oftentimes we have to figure out whether there already is a cache line that holds a copy of the requested address. The function *isCached* recursively walks through the list of cache lines and returns true if and only if a matching entry is found. Particularly, the function returns true even if the cache line has only been reserved but does not yet contain valid data.

$$isCached : CacheLine^{*} \times PhysMemAddr \to \mathbb{B}$$

$$isCached(\varepsilon, pAddr) = \text{false}$$

$$isCached(cl \circ \overrightarrow{cl}, pAddr) = \begin{cases} \text{true} & \text{if} \quad cl_{pAddr} = pAddr \\ isCached(\overrightarrow{cl}, pAddr) & \text{otherwise} \end{cases}$$

The CUDA documentation does not give any details about the inner workings of the caches. On the other hand, [11] suggests that most caches found on the GPU are in some way associative. Associativity means that only a subset of all cache lines are eligible to store a certain address. The purpose of the abstract function *eligible* is to abstract from the concrete mechanism used to decide which cache lines can be used to cache a certain address. Given all cache lines and a memory address, it returns the set of cache lines that are eligible to cache the request. If the cache is not associative and does not use any other sophisticated mechanism to allocate cache lines for a request, the function might simply return the original set of cache lines. As functions are deterministic by their very nature, *eligible* always returns the same set of eligible cache lines for the same input, guaranteeing that cached values are found reliably.

$$eligible : CacheLine^{*} \times PhysMemAddr \to CacheLine^{*}$$

canEvict decides whether a given cache line can be evicted. We do not give an implementation for this function; further details about the eviction policy most likely employed by Fermi GPUs can be found in [22]. For convenience, we define a function with the same name that takes a memory address instead of a specific cache line. If a cache line matching the address exists, it uses the abstract *canEvict* function to decide whether the cache line can be evicted.

Otherwise, it returns $\perp$. It is well-defined because there cannot be more than one cache line with a matching address; this is implicitly guaranteed by all of the operations defined in this section.

$$\mathit{canEvict} : \mathit{CacheLine}^* \times \mathit{CacheLine} \to \mathbb{B} \qquad \mathit{canEvict} : \mathit{CacheLine}^* \times \mathit{ProgAddr} \to \mathbb{B}_\perp$$

Even though *canEvict* is abstract, we expect the function to adhere to the following two axioms: Dirty and pinned cache lines are never eligible for eviction.

$$\mathit{canEvict}(\overrightarrow{cl}, cl \triangleright \text{true}, \text{false}) = \text{false} \qquad \mathit{canEvict}(\overrightarrow{cl}, cl \triangleright \overrightarrow{mid}) = \text{false} \quad \text{if} \quad \overrightarrow{mid} \neq \varepsilon$$

The *write* helper function copies the given cache word into the given cache line. The cache line's valid and dirty flags are set to true. We assume there is a hardware optimization for the case that the written value is equal to the value already stored in the cache line. That way, we can keep the old dirty flag and might be able to skip an unnecessary write-back. For instance, the value in the cache line might have been read from memory, thus the data is valid and not dirty. If a program writes the same value to the address, there is no point in copying the same value to memory. However, we do not know if such an optimization really exists in hardware, though it seems likely.

$$\begin{aligned} \mathit{write} &: \mathit{CacheLine} \times \mathit{CacheWord} \to \mathit{CacheLine} \\ \mathit{write}(cl \triangleright cw, cw) &= cl \\ \mathit{write}(cl \triangleright cw_\varepsilon, cw') &= cl \triangleleft cw', \text{true}, \text{true} \quad \text{if} \quad cw_\varepsilon \neq cw' \end{aligned}$$

Operations

The function *reserve* reserves a cache line for a specific memory address. It should not be used if there already is a cache line with a matching address as there may be only one single cache line for each address. On the other hand, if there is no matching cache line, there are three possible cases: If there is an eligible cache line that is currently unused, this cache line is initialized and the requested memory address and the index of the requesting memory operation are copied into the cache line. For convenience, no memory operation index may be given; this allows us to use *reverse* in the *write* function defined below without having to remove the operation index immediately after the call to *reserve*. In the second case, there is no unused cache line in the set of eligible cache lines, but one of the eligible lines can be evicted. Since the function *canEvict* guarantees that it does not allow eviction of dirty cache lines, we can immediately reinitialize the cache line as outlined for the first case, because no write-back of the cached value is required. If no eligible cache line is empty or can be evicted, the reservation fails. However, failure in this case is expected — and the reservation will be retried at some point in the future —, so we do not return $\perp$ in this case, but return empty cache lines instead to designate this situation. The semantics of the memory program interprets this return value and pauses the memory program until the reservation eventually succeeds.

reserve splits the given set of cache lines into the reserved cache line and the rest of the cache lines sans the reserved cache line. A function that uses the results returned from *reserve* therefore has to merge the result to obtain the original amount of cache lines again. While this seems unintuitive, it is unfortunately necessary because oftentimes the reserved cache line needs to be manipulated further, so returning it directly from *reserve* is advantageous.

$$
\begin{aligned}
&\mathit{reserve} : \mathit{CacheLine}^{*} \times \mathit{PhysMemAddr} \times \mathit{MemOpIdx}_{\varepsilon} \rightarrow (\mathit{CacheLine}_{\varepsilon} \times \mathit{CacheLine}^{*})_{\perp}\\
&\mathit{reserve}(\varepsilon, \mathit{pAddr}, \mathit{mid}_{\varepsilon}) = \perp\\
&\mathit{reserve}(\overrightarrow{\mathit{cl}}, \mathit{pAddr}, \mathit{mid}_{\varepsilon}) = \perp \quad \text{if} \quad \mathit{isCached}(\overrightarrow{\mathit{cl}}, \mathit{pAddr})\\
&\mathit{reserve}(\overrightarrow{\mathit{cl}}, \mathit{pAddr}, \mathit{mid}_{\varepsilon}) = \begin{cases}
(\mathit{init} \blacktriangleleft (\mathit{pAddr}, \mathit{mid}_{\varepsilon}), \overrightarrow{\mathit{cl}} \setminus \mathit{cl}) & \text{if} \quad \mathit{unused}(\mathit{cl})\\
(\mathit{init} \blacktriangleleft (\mathit{pAddr}, \mathit{mid}_{\varepsilon}), \overrightarrow{\mathit{cl}} \setminus \mathit{cl}) & \text{if} \quad \mathit{canEvict}(\overrightarrow{\mathit{cl}'}, \mathit{cl})\\
& \wedge\, \mathit{allUsed}(\mathit{cl} \circ \overrightarrow{\mathit{cl}'})\\
(\varepsilon, \varepsilon) & \text{otherwise}
\end{cases}\\
&\qquad\qquad\qquad\qquad \text{where} \quad \mathit{cl} \circ \overrightarrow{\mathit{cl}'} = \mathit{eligible}(\overrightarrow{\mathit{cl}}, \mathit{pAddr}) \quad \text{if} \quad \neg \mathit{isCached}(\overrightarrow{\mathit{cl}}, \mathit{pAddr})
\end{aligned}
$$

Before reading a value from a cache, memory programs must first check whether the requested value is actually cached at the moment. If it is not cached, it is the memory program's responsibility to retrieve the value from a higher level of the memory hierarchy after it has reserved a cache line for the requested memory address; this is not done automatically by the cache itself. Consequently, a call to *read* returns ⊥ if the requested address is not cached. If it is cached, however, there are two possible cases: Either, the data is valid, in which case the data is immediately returned and the cache line is unpinned — the unpinning has no effect if the memory operation has not pinned the cache line before —, or the data is invalid, in which case no data is returned and the cache line is pinned. The intended usage scenario of the *read* function is to first check whether an address is actually cached and if so to read the value by invoking *read*. This way, the value is either read right now and the program continues, or it cannot be read right now and the program yields execution. Later, when it continues execution, it immediately calls *read* again. Again, it either gets the value or it has to wait for a later point in time. If the memory subsystem ever copies the requested value into the cache line, the program is guaranteed to read the value eventually, because the cache line cannot be evicted as long as the program does not unpin the cache line, i.e. as long as the program has not read the value.

$$
\begin{aligned}
&\mathit{read} : \mathit{CacheLine}^{*} \times \mathit{PhysMemAddr} \times \mathit{MemOpIdx} \rightarrow (\mathit{CacheLine}^{*} \times \mathit{CacheWord}_{\varepsilon})_{\perp}\\
&\mathit{read}(\varepsilon, \mathit{pAddr}, \mathit{mid}) = \perp\\
&\mathit{read}(\overrightarrow{\mathit{cl}}, \mathit{pAddr}, \mathit{mid}) = \begin{cases}
(\mathit{unpin}(\mathit{cl}, \mathit{mid}) \circ \overrightarrow{\mathit{cl}'}, \mathit{cl}_{\mathit{cw}}) & \text{if} \quad \overrightarrow{\mathit{cl}} = (\mathit{cl} \blacktriangleright \mathit{pAddr}) \circ \overrightarrow{\mathit{cl}'} \wedge \mathit{cl}_{\mathit{valid}}\\
(\mathit{pin}(\mathit{cl}, \mathit{mid}) \circ \overrightarrow{\mathit{cl}'}, \varepsilon) & \text{if} \quad \overrightarrow{\mathit{cl}} = (\mathit{cl} \blacktriangleright \mathit{pAddr}) \circ \overrightarrow{\mathit{cl}'} \wedge \neg \mathit{cl}_{\mathit{valid}}\\
\perp & \text{if} \quad \neg \mathit{isCached}(\overrightarrow{\mathit{cl}}, \mathit{pAddr})
\end{cases}
\end{aligned}
$$

Memory programs use the function *write* to insert a value into the cache. *write* may only be used if the write request originates from the program; if the cache is to be updated after a cache miss, the function *update* defined below must be used instead. There are three possibilities when writing a value into a cache: If the requested address is not yet cached but it is possible to reserve a cache line right now, the cache line is reserved and the data is copied into the cache. Secondly, the requested address is already cached, so we can just copy the data into the cache line. However, we might write data into a cache line that has only been reserved and contains no valid data yet. For the moment, this is not a problem. But once the value requested from a higher level of the hierarchy due to an earlier cache miss arrives, the value just written might be overwritten by an old value. Therefore, the *update* function defined below must not overwrite cache lines that have been updated in the meantime. The third possibility is that the requested address is not cached and a reservation is currently impossible. In this case, the memory program yields execution and retries at a later point in time, just like outlined above for cached reads.

$$\mathit{write} : \mathit{CacheLine}^* \times \mathit{PhysMemAddr} \times \mathit{CacheWord} \to (\mathit{CacheLine}^*)_\perp$$

$$\mathit{write}(\varepsilon, \mathit{pAddr}, \mathit{cw}) = \perp$$

$$\mathit{write}(\overrightarrow{cl}, \mathit{pAddr}, \mathit{cw}) = \begin{cases} \mathit{write}(cl, cw) \circ \overrightarrow{cl'} & \text{if} \quad \neg \mathit{isCached}(\overrightarrow{cl}, \mathit{pAddr}) \\ & \qquad \wedge\ \mathit{reserve}(\overrightarrow{cl}, \mathit{pAddr}, \varepsilon) = (cl, \overrightarrow{cl'}) \\ \mathit{write}(cl, cw) \circ \overrightarrow{cl'} & \text{if} \quad \overrightarrow{cl} = (cl \triangleright \mathit{pAddr}) \circ \overrightarrow{cl'} \\ \varepsilon & \text{if} \quad \neg \mathit{isCached}(\overrightarrow{cl}, \mathit{pAddr}) \\ & \qquad \wedge\ \mathit{reserve}(\overrightarrow{cl}, \mathit{pAddr}, \varepsilon) = (\varepsilon, \varepsilon) \end{cases}$$

The *update* function updates a cache after a cache miss. The function cannot be called when the cache line has not been reserved previously. If a cache line for the given memory address can be found, the data is written and the cache line is set to valid — unless there was a write to this cache line in the meantime, as outlined above.

$$\mathit{update} : \mathit{CacheLine}^* \times \mathit{PhysMemAddr} \times \mathit{CacheWord} \to (\mathit{CacheLine}^*)_\perp$$

$$\mathit{update}(\varepsilon, \mathit{pAddr}, \mathit{cw}) = \perp$$

$$\mathit{update}(\overrightarrow{cl}, \mathit{pAddr}, \mathit{cw}) = \begin{cases} (cl \triangleleft cw, \text{false}, \text{true}) \circ \overrightarrow{cl'} & \text{if} \quad \overrightarrow{cl} = (cl \triangleright \mathit{pAddr}) \circ \overrightarrow{cl'} \wedge \neg cl_{valid} \\ \overrightarrow{cl} & \text{if} \quad \overrightarrow{cl} = (cl \triangleright \mathit{pAddr}) \circ \overrightarrow{cl'} \wedge cl_{valid} \\ \perp & \text{if} \quad \neg \mathit{isCached}(\overrightarrow{cl}, \mathit{pAddr}) \end{cases}$$

The *evict* function is used to specifically request the eviction of the cache line denoted by the given address. In that case, eviction classes are ignored, so for example a cache line of normal eviction policy is evicted even though a cache line of class first would normally have been evicted first. Still, eviction is only possible if the cache line is not dirty and not pinned, otherwise the memory program has to wait until later before it can try again.

$$\mathit{evict} : \mathit{CacheLine}^* \times \mathit{PhysMemAddr} \to (\mathit{CacheLine}^*)_\perp$$

$$\mathit{evict}(\varepsilon, \mathit{pAddr}) = \perp$$

$$\mathit{evict}(\overrightarrow{cl}, \mathit{pAddr}) = \begin{cases} \mathit{init} \circ \overrightarrow{cl'} & \text{if} \quad \overrightarrow{cl} = (cl \triangleright \mathit{pAddr}) \circ \overrightarrow{cl'} \wedge \neg cl_{dirty} \wedge cl_{\overrightarrow{mid}} = \varepsilon \\ \overrightarrow{cl} & \text{if} \quad \neg \mathit{isCached}(\overrightarrow{cl}, \mathit{pAddr}) \\ \varepsilon & \text{if} \quad \overrightarrow{cl} = (cl \triangleright \mathit{pAddr}) \circ \overrightarrow{cl'} \wedge (cl_{dirty} \vee cl_{\overrightarrow{mid}} \neq \varepsilon) \end{cases}$$

The function *discard* is similar to *evict*. The difference is that the cache line denoted by the given address is evicted in any case if it is not pinned by any pending memory operation. In particular, this includes dirty cache lines. Hence, if used unwisely, data loss can occur. However, some cache operations like `.lu` specifically request a cache line be deleted from the L1 cache without a prior write-back [9, Table 80]. This can be used to avoid needless write-backs of values that are never used again, for example when a register spilled to local memory is restored.

$$
\begin{aligned}
&\mathit{discard} : \mathit{CacheLine}^{*} \times \mathit{PhysMemAddr} \rightarrow (\mathit{CacheLine}^{*})_{\perp}\\
&\mathit{discard}(\varepsilon, \mathit{pAddr}) = \perp\\
&\mathit{discard}(\overrightarrow{cl}, \mathit{pAddr}) = \begin{cases} \mathit{init} \circ \overrightarrow{cl'} & \text{if} \quad \overrightarrow{cl} = (cl \triangleright \mathit{pAddr}) \circ \overrightarrow{cl'} \wedge cl_{\overrightarrow{mid}} = \varepsilon \\ \overrightarrow{cl} & \text{if} \quad \neg \mathit{isCached}(\overrightarrow{cl}, \mathit{pAddr}) \\ \varepsilon & \text{if} \quad \overrightarrow{cl} = (cl \triangleright \mathit{pAddr}) \circ \overrightarrow{cl'} \wedge cl_{\overrightarrow{mid}} \neq \varepsilon \end{cases}
\end{aligned}
$$

4.3 Formalization of the Memory Environment

The types of memory supported by our formalization of CUDA's memory model are constant, global, local, and shared memory. What these four types of memory have in common is that they are addressable by PTX programs; registers, on the other hand, are not. We define the domain *StateSpace* to refer to one of the four addressable memory types. Additionally, we are interested in the two subsets of *StateSpace* that designate the memory types supported by atomic memory operations and the memory types that lie in DRAM.

$$
\begin{aligned}
\mathit{StateSpace} &= \{\texttt{.global}, \texttt{.local}, \texttt{.shared}, \texttt{.const}\}\\
\mathit{ssa} \in \mathit{StateSpace}_{\mathit{atomic}} &= \{\texttt{.global}, \texttt{.shared}\} \subset \mathit{StateSpace}\\
\mathit{ssd} \in \mathit{StateSpace}_{\mathit{device}} &= \{\texttt{.global}, \texttt{.const}, \texttt{.local}\} \subset \mathit{StateSpace}
\end{aligned}
$$

As mentioned above, the domain *PhysMemAddr* is used to address a location in any of the memory types. We define the following subsets of this address space that identify locations in a specific memory type. For the addressable memory types, each address corresponds to one byte of memory. For registers, each address identifies a register in a SM's register file.

$$
\begin{aligned}
\mathit{GlobalAddr} &\subset \mathit{PhysMemAddr} & \mathit{SharedAddr} &\subset \mathit{PhysMemAddr}\\
\mathit{ConstAddr} &\subset \mathit{PhysMemAddr} & \mathit{RegAddr} &\subset \mathit{PhysMemAddr}\\
\mathit{LocalAddr} &\subset \mathit{PhysMemAddr}
\end{aligned}
$$

However, global, local, and constant memory are all mapped to locations in DRAM. We therefore have to assume that global, local, and constant addresses uniquely identify a location in DRAM, meaning that the same address cannot be used by more than one of the DRAM memory types.

$$
\begin{aligned}
\mathit{GlobalAddr} \cap \mathit{LocalAddr} &= \emptyset & \mathit{GlobalAddr} \cap \mathit{ConstAddr} &= \emptyset\\
\mathit{LocalAddr} \cap \mathit{ConstAddr} &= \emptyset
\end{aligned}
$$

Moreover, the number of addresses for each memory type is limited by the amount of available memory on the graphics card:

$$
\begin{aligned}
|\mathit{GlobalAddr}| &= \mathit{GlobalMemSize} & |\mathit{SharedAddr}| &= \mathit{SharedMemSize}\\
|\mathit{ConstAddr}| &= \mathit{ConstMemSize} & \left|\mathit{RegAddr}\right| &= \mathit{RegFileSize}\\
|\mathit{LocalAddr}| &= \mathit{LocalMemSize} \cdot \mathit{NumProcessors} \cdot \mathit{MaxResidentThreads}
\end{aligned}
$$

Shared memory is organized into banks such that successive 32 bit words are assigned to successive banks. Memory operations affecting different shared memory banks can be serviced concurrently; if more than one 32 bit word is read from the same bank, a bank conflict occurs and access is serialized. However, we only define abstract functions that implicitly define the bank layout and resolve bank conflicts. We do not give concrete implementations because bank conflicts merely affect program performance. By contrast, we do have to consider bank locking to correctly define the semantics of shared memory atomic operations, even though it is not officially documented when and how shared memory banks are locked. Shared memory atomic operations are not guaranteed to be atomic — in contrast to atomic operations on global memory [9, Table 105] — because of the way the PTX compiler uses the locking mechanism to implement atomic operations on shared memory. Therefore, we cannot simply omit the concept of shared memory banks, even though they are solely relevant for performance consideration except for this one case. Consequently, a shared memory bank can either be in locked or unlocked state.

$$Lock = \mathbb{B}$$

We identify shared memory banks by their bank index. The maximum number of shared memory banks is determined by the compute capability of the underlying hardware.

$$BankIdx = \{1, \ldots, SharedMemBanks\}$$

The abstract function *banks* returns the indices of the banks that are accessed when a given amount of bytes starting at the specified address are read from or written to shared memory.

$$banks : SharedAddr \times MemOpSize \to BankIdx^*$$

The formalization of a memory is a function that maps an address to a byte or a register value. For shared memory, we also have to keep track of each bank's lock state. Similarly, we keep track of each register's blocked state. We neglect CUDA's concept of constant memory banks for the reasons outlined in section 2.3.2.

$$\begin{aligned} GlobalMem &= GlobalAddr \to Byte & LocalMem &= LocalAddr \to Byte \\ ConstMem &= ConstAddr \to Byte & RegFile &= RegAddr \to RegValue \\ SharedMem &= (SharedAddr \to Byte) \times (BankIdx \to Lock) \end{aligned}$$

With the above formalization of the different memory types, we can finally define the memory environment that we base the semantics of memory programs on. The memory environment comprises global, local, and constant memory as well as the L2 cache. In addition, we define a domain that encapsulates the memories found on each streaming multiprocessor, i.e. shared memory, the register file, and the L1 cache. The memory environment consequently contains a processor memory environment for each streaming multiprocessor.

$$\begin{aligned} ProcMem &= SharedMem \times L1Cache \times RegFile \\ \eta \in MemEnv &= GlobalMem \times ConstMem \times LocalMem \times L2Cache \times ProcMem^{NumProcessors} \end{aligned}$$

The streaming multiprocessors are successively numbered. These numbers represent the indices of the processors and uniquely identify each one. We assume that the location of a processor in the list of processor memories in the memory environment corresponds to its index number.

$$pid \in ProcIdx = \{0, \ldots, NumProcessors - 1\}$$

Similar to caches, we now formalize the operations supported on the memory environment that the memory program semantics later makes use of, namely reading and writing of any type of memory. To do this, we first define the following helper functions that cut down the amount of projections required to define the rules and functions found in the remainder of this chapter. These functions allow us to access a processor's L1 cache, shared memory, or register file.

$$\begin{aligned}
&l1Cache : MemEnv \times ProcIdx \rightarrow L1Cache && regFile : MemEnv \times ProcIdx \rightarrow RegFile\\
&l1Cache(\eta, pid) = \eta_{\overrightarrow{procMem},pid,l1Cache} && regFile(\eta, pid) = \eta_{\overrightarrow{procMem},pid,regFile}
\end{aligned}$$

$$\begin{aligned}
&sharedMem : MemEnv \times ProcIdx \rightarrow SharedMem\\
&sharedMem(\eta, pid) = \eta_{\overrightarrow{procMem},pid,sharedMem}
\end{aligned}$$

The function $read_{device}$ reads a value from device memory, i.e. global, constant, or local memory. It returns a list of bytes, the size of which depends on the size of the memory request. Additionally, we check whether the given address actually matches the specified state space. If it does not, $\perp$ is returned. We use the abbreviated list concatenation syntax $b_1 \ldots b_n$ instead of $b_1 :: \ldots :: b_n$ to make the definition more readable.

$$\begin{aligned}
&read_{device} : MemEnv \times StateSpace_{device} \times PhysMemAddr \times MemOpSize \rightarrow (Byte^*)_\perp\\
&read_{device}(\eta, \texttt{.global}, pAddr, size) = \eta_{globalMem}(pAddr + 0) \ldots \eta_{globalMem}(pAddr + size - 1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if} \quad \{pAddr + 0, \ldots, pAddr + size - 1\} \subseteq GlobalAddr\\
&read_{device}(\eta, \texttt{.local}, pAddr, size) = \eta_{localMem}(pAddr + 0) \ldots \eta_{localMem}(pAddr + size - 1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if} \quad \{pAddr + 0, \ldots, pAddr + size - 1\} \subseteq LocalAddr\\
&read_{device}(\eta, \texttt{.const}, pAddr, size) = \eta_{constMem}(pAddr + 0) \ldots \eta_{constMem}(pAddr + size - 1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if} \quad \{pAddr + 0, \ldots, pAddr + size - 1\} \subseteq ConstAddr\\
&read_{device}(\eta, ssd, pAddr, size) = \perp \quad \text{otherwise}
\end{aligned}$$

We define a similar function for shared memory reads. However, shared memory reads can access several different locations concurrently, thus the function is passed a list of addresses and access sizes and returns a list of read values. We assume that the function that translates the shared memory accesses of threads into memory operations deals with bank conflicts such that requests that are in conflict are split into two or more conflict-free requests. Therefore, $read_{shared}$ can safely ignore the possibility of bank conflicts. Additionally, non-atomic reads of shared memory do not care about a bank being locked, hence $read_{shared}$ succeeds even if an accessed bank is locked.

$$\begin{aligned}
&read_{shared} : MemEnv \times ProcIdx \times PhysMemAddr^* \times MemOpSize^* \rightarrow ((Byte^*)^*)_\perp\\
&read_{shared}(\eta, pid, \varepsilon, \varepsilon) = \varepsilon\\
&read_{shared}(\eta, pid, pAddr :: \overrightarrow{pAddr}, size :: \overrightarrow{size}) =
\end{aligned}$$

$$(s(pAddr + 0) \dots s(pAddr + size - 1)) :: read_{shared}(\eta, pid, \overrightarrow{pAddr}, \overrightarrow{size})$$

$$\text{if} \quad s = \pi_1\, sharedMem(\eta, pid) \land \{pAddr + 0, \dots, pAddr + size - 1\} \subseteq SharedAddr$$

$$read_{shared}(\eta, pid, \overrightarrow{pAddr}, \overrightarrow{size}) = \bot \quad \text{otherwise}$$

Finally, we define two functions that enable memory programs to write to global, local, and shared memory. Obviously, threads are not allowed to write to constant memory. Only the host program can write to constant memory; however, the semantics of the host program are outside the scope of this report.

$$write_{device} : MemEnv \times StateSpace_{device} \times PhysMemAddr \times MemOpSize \times Byte^{*} \rightarrow MemEnv_{\bot}$$

$$write_{device}(\eta \blacktriangleright globalMem, \texttt{.global}, pAddr, size, \vec{b}) =$$

$$\eta \blacktriangleleft globalMem[pAddr + 0 \dots pAddr + size - 1 \mapsto \vec{b}_0 \dots \vec{b}_{size-1}]$$

$$\text{if} \quad \{pAddr + 0, \dots, pAddr + size - 1\} \subseteq GlobalAddr$$

$$write_{device}(\eta \blacktriangleright localMem, \texttt{.local}, pAddr, size, \vec{b}) =$$

$$\eta \blacktriangleleft localMem[pAddr + 0 \dots pAddr + size - 1 \mapsto \vec{b}_0 \dots \vec{b}_{size-1}]$$

$$\text{if} \quad \{pAddr + 0, \dots, pAddr + size - 1\} \subseteq LocalAddr$$

$$write_{device}(\eta, ssd, pAddr, size, \vec{b}) = \bot \quad \text{otherwise}$$

Again, several shared memory locations can be written concurrently, bank conflicts are already accounted for, and locked banks do not prevent a non-atomic write.

$$write_{shared} : MemEnv \times ProcIdx \times PhysMemAddr^{*} \times MemOpSize^{*} \times ((Byte^{*})^{*}) \rightarrow MemEnv_{\bot}$$

$$write_{shared}(\eta, pid, \varepsilon, \varepsilon, \varepsilon) = \eta$$

$$write_{shared}(\eta \blacktriangleright \overrightarrow{procMem}, pid, pAddr :: \overrightarrow{pAddr}, size :: \overrightarrow{size}, \vec{b} :: \vec{\vec{b}}) =$$

$$write_{shared}(\eta \blacktriangleleft \overrightarrow{procMem}[pid \mapsto \overrightarrow{procMem}_{pid} \blacktriangleleft s], pid, \overrightarrow{pAddr}, \overrightarrow{size}, \vec{\vec{b}})$$

$$\text{where} \quad s' = sharedMem(\eta, pid)$$

$$\land\, s = s' \blacktriangleleft \pi_1(s')[pAddr + 0 \dots pAddr + size - 1 \mapsto \vec{b}_0 \dots \vec{b}_{size-1}]$$

$$\text{if} \quad \{pAddr + 0, \dots, pAddr + size - 1\} \subseteq SharedAddr$$

$$write_{shared}(\eta, pid, \overrightarrow{pAddr}, \overrightarrow{size}, \vec{\vec{b}}) = \bot \quad \text{otherwise}$$

4.4 Formalization and Formal Semantics of Memory Programs

We have already informally presented the syntax and semantics of memory programs in section 4.1 that we now formally define based on the formalization of caches and the memory environment found in the preceding sections. There are four basic terms we use throughout the remainder of this chapter: instantaneous execution, deferred execution, micro steps, and macro steps. As the preceding sections have already mentioned several times, a memory programs is able to yield execution and allow other programs to advance before it resumes execution itself. This yielding can either happen explicitly by processing a `yield` statement or implicitly according to the semantics of a cache operation or shared memory bank locking. When an operation does not perform an implicit `yield`, we call the operation instantaneous. Otherwise, it defers execution until a later point in time. There are operations that are guaranteed to be instantaneous in any case and others whose behavior depends on the state of both the memory environment and the program. When a program is allowed to execute, it executes all instructions until either an explicit or implicit `yield` is performed; we call such a series of executed instructions a micro step. Micro steps are uninterruptible and never concurrent, i.e. no two programs ever execute a micro step at the same time. We assign one of the following yield actions to each program statement to decide whether the micro step is not yet completed and the program should continue to execute, the micro step is completed and the program should yield execution, or the program has terminated.

$$ya \in \mathit{YieldAction} ::= \text{continue} \mid \text{yield} \mid \text{end}$$

Macro steps, on the other hand, form the semantics of the memory model as a whole and are defined in section 4.6. Among other things, they consist of an arbitrary amount of serially executed micro steps of different programs. Within a single macro step, each program can perform zero, one, or more micro steps.

Memory programs can manipulate the state of the memory environment as well as their own implicit state. Additionally, they can execute different code paths depending on the state of the memory environment. Therefore, a memory program consists of Boolean expressions and operational expressions as well as program statements that chain up these expressions and define the general program flow. In order to keep things simple, program statements do not contain variable assignment, function calls, or while-loops. Still, the language is sufficiently expressive to define the semantics of the memory operations supported by CUDA; some exemplary memory programs can be found in section 4.5. The next sections formalize the implicit program state as well as the grammar and semantics of Boolean expressions, operational expressions, and program statements.

4.4.1 Implicit Program State

Memory programs cannot use variables to store data and pass information around. Instead, all relevant state is implicit and inaccessible to the program. The semantics of Boolean and operational expression, however, pass parts of the implicit state as input to the cache and memory operations defined in the previous sections. Hence, we have to formalize the implicit state of memory programs before we define the semantics of expressions.

The program state Σ as specified below might seem overly bloated. However, we deliberately do not define it in a more fine-grained way. The coarse structure of the state prevents the need of a multitude of projections in function definitions and the like. However, we do partition the state domain into four sub-domains: Σ_{atomic} for atomic memory operations, Σ_{device} for DRAM memory operations, Σ_{shared} for shared memory operations, and Σ_{reg} for register update operations.

$$\sigma \in \Sigma = \Sigma_{atomic} \cup \Sigma_{device} \cup \Sigma_{shared} \cup \Sigma_{reg}$$

Except for Σ_{reg} which is somewhat of a special case, all of these program states contain one or more byte lists that are used to hold input and output data, i.e. either a value read from a cache or memory or the value provided by the program that should be written to a cache or memory. Reading from and writing to these byte lists is handled by the program semantics implicitly, not by the programs themselves. To highlight the intended usage of the byte lists, we introduce the following three aliases:

$$v_{in} \in Value_{in} = Byte^* \qquad v_{i/o} \in Value_{in/out} = Byte^* \qquad v_{out} \in Value_{out} = Byte^*$$

Typically, read requests from threads of the same warp are coalesced into fewer, but larger memory transactions. When that happens, it is unclear which value should be written to which output register. As we do not know how requests are coalesced — this is done by an abstract function defined later on —, we also do not know how the value retrieved by the memory program should be decomposed and copied into the destination registers of the requesting threads. We solve this issue by assuming that a decomposition function is given to each program that has to decompose a retrieved value. The abstract function that translates thread requests into one or more memory operations creates a decomposition function that reverts the effects of coalescing and stores it in the program's implicit state. The program semantics can then use the decomposition function to successfully write the retrieved values to the destination registers.

$$df \in DecompFunc = RegFile \times \Sigma \to RegFile$$

Suppose two threads execute the statement `ld.global.ca.s32 r, [a]`, where `r` is the destination register that stores the result of the read and `a` is the location of the value to read. Furthermore, suppose that the values of `r` and `a` are r_1 and a_1 for the first thread and r_2 and a_2 for the second one. Moreover, assume that $a_2 = a_1 + 4$. Because both threads read a 4 byte signed integer, the request might be coalesced into one 8 byte request at location a_1. After the completion of the corresponding memory program, the program's state contains the retrieved value in its value list $\vec{b}$. The decomposition function subsequently copies $\vec{b}_0 \ldots \vec{b}_3$ into register r_1 and $\vec{b}_4 \ldots \vec{b}_7$ into register r_2.

When a thread issues a read request to memory, the destination register of the read is blocked until the read is completed, i.e. the value is written to the register. For a write request, the source register is blocked until the memory subsystem has picked up the value. A thread cannot be scheduled if its next instruction accesses a blocked register. CUDA does not specify when exactly registers are unblocked; it is only said that registers used in a write request are unblocked "much more quickly" than those used in a read request [9, 6.6]. Therefore, we make it the responsibility of the memory programs to unblock blocked registers at some point during the lifetime of the program. Consequently, the program's state contains a list of all registers that were blocked when the request was issued. All of these registers are unlocked at the program's command.

All programs are identified by a unique memory program index and their states optionally contain a decomposition function and a list of blocked registers the program should release. Furthermore, all states store the index of the streaming multiprocessor of the threads that issued the request. The SM index is used to find out which register file, L1 cache, or shared memory should be manipulated by the program. Additionally, Σ_{atomic} stores the location and the size of the value that should be manipulated by the program. It also comprises the atomic operation that should be performed as well as the state space — global or shared — that should be accessed. We assume that a single memory program performs only one atomic operation. Several atomic operations on distinct addresses could be executed in parallel just as well. The CUDA specification does not give any hints as to what the hardware really does, but as atomic operations cannot possibly affect one another, this really has no effect on program behavior. On the other hand, the PTX specification clearly states that the memory model does not guarantee atomicity of atomic operations on shared memory with respect to other, non-atomic memory operations on the same address [9, Table 105]; neither does the formalization presented in this chapter. Depending on the type of the atomic operation, an atomic memory request either has one or two input values. An atomic operation writes the original value at the accessed location to the destination register.

$$\begin{aligned}\sigma_a \in \Sigma_{atomic} = {} & \mathit{PhysMemAddr} \times \mathit{MemOpSize} \times \mathit{ProcIdx} \times \mathit{StateSpace}_{atomic} \times \mathit{AtomicOp} \\ & \times \mathit{Value}_{out} \times \mathit{Value}_{in} \times \mathit{Value}_{in} \times \mathit{DecompFunc} \times \mathit{RegAddr}^* \times \mathit{MemOpIdx}\end{aligned}$$

The program state for DRAM memory operations is used by programs for both read and write operations. Consequently, the byte list in the program's state is used either for the input or output value. The state also stores the request's address and the size.

$$\begin{aligned}\sigma_d \in \Sigma_{device} = {} & \mathit{PhysMemAddr} \times \mathit{MemOpSize} \times \mathit{StateSpace}_{device} \times \mathit{CacheOp}_{\varepsilon} \times \mathit{ProcIdx} \\ & \times \mathit{Value}_{in/out} \times \mathit{DecompFunc}_{\varepsilon} \times \mathit{RegAddr}^* \times \mathit{MemOpIdx}\end{aligned}$$

As memory operations on shared memory can concurrently read or write several different locations, the state of a shared memory program contains several addresses and access sizes. Again, it is assumed that any possible bank conflicts are resolved before memory programs and their states are created by splitting conflicting accesses into several conflict-free ones. Just like the state of DRAM memory operations, the shared memory operation state utilizes the byte list for input and output purposes.

$$\begin{aligned}\sigma_s \in \Sigma_{shared} = {} & \mathit{ProcIdx} \times \mathit{PhysMemAddr}^* \times \mathit{MemOpSize}^* \times \mathit{Value}_{in/out}{}^* \times \mathit{DecompFunc}_{\varepsilon} \\ & \times \mathit{RegAddr}^* \times \mathit{MemOpIdx}\end{aligned}$$

Some memory operations only update registers, but do not touch other types of memory. This is a bit of a special case, because we can use the decomposition function to update the registers, hence the program's state does not have to store the register addresses or the new values. Memory programs with a Σ_{reg} program state are issued when threads perform operations that write to registers. Basically, this includes all arithmetic, logic, and shift instructions supported by PTX. The decomposition function inherently supports coalescing register updates of several threads into one memory operation, assuming the hardware supports that.

$$\sigma_r \in \Sigma_{reg} = \mathit{ProcIdx} \times \mathit{DecompFunc} \times \mathit{RegAddr}^* \times \mathit{MemOpIdx}$$

4.4.2 Boolean Expressions

A memory program can use Boolean expressions in an if-else statement in order to execute different code paths depending on the state of the memory environment. The supported operations are negation, conjunction, and disjunction as well as the two constant values true and false. Additionally, a memory program accessing global or local memory can query whether the requested address is currently cached in L1 or L2.

BExp ::= `true | false`

$$
\begin{aligned}
&\mid \texttt{!}\mathit{BExp} \mid \mathit{BExp}\ \texttt{and}\ \mathit{BExp} \mid \mathit{BExp}\ \texttt{or}\ \mathit{BExp}\\
&\mid \texttt{isCachedL1} \mid \texttt{isCachedL2}
\end{aligned}
$$

Boolean expressions are always instantaneous, thus a program never implicitly yields execution due to the evaluation of a Boolean expression. Furthermore, cache queries are only valid if the memory program actually handles a cached operation. Otherwise, the evaluation of the expression returns $\bot$ and the program is stuck. We generally use $\bot$ as a sanity check to avoid faulty memory programs, i.e. memory programs with undefined behavior.

$$
\begin{aligned}
&\mathfrak{B}[\![-]\!] : \mathit{BExp} \to (\mathit{MemEnv} \times \Sigma \to \mathbb{B}_\bot)\\
\mathfrak{B}[\![\texttt{true}]\!]\eta\sigma &= \text{true}\\
\mathfrak{B}[\![\texttt{false}]\!]\eta\sigma &= \text{false}\\
\mathfrak{B}[\![\texttt{!}\mathit{bExp}]\!]\eta\sigma &= \neg\mathfrak{B}[\![\mathit{bExp}]\!]\eta\sigma\\
\mathfrak{B}[\![\mathit{bExp}_1\ \texttt{and}\ \mathit{bExp}_2]\!]\eta\sigma &= \mathfrak{B}[\![\mathit{bExp}_1]\!]\eta\sigma \wedge \mathfrak{B}[\![\mathit{bExp}_2]\!]\eta\sigma\\
\mathfrak{B}[\![\mathit{bExp}_1\ \texttt{or}\ \mathit{bExp}_2]\!]\eta\sigma &= \mathfrak{B}[\![\mathit{bExp}_1]\!]\eta\sigma \vee \mathfrak{B}[\![\mathit{bExp}_2]\!]\eta\sigma\\
\mathfrak{B}[\![\texttt{isCachedL1}]\!]\eta\sigma_d &= \mathit{isCached}(\mathit{l1Cache}(\eta, \sigma_{pid}), \sigma_{pAddr}) && \text{if} \quad \sigma_{d,cop_\varepsilon} \neq \varepsilon\\
\mathfrak{B}[\![\texttt{isCachedL2}]\!]\eta\sigma_d &= \mathit{isCached}(\eta_{l2Cache}, \sigma_{pAddr}) && \text{if} \quad \sigma_{d,cop_\varepsilon} \neq \varepsilon\\
\mathfrak{B}[\![\mathit{bExp}]\!]\eta\sigma &= \bot && \text{otherwise}
\end{aligned}
$$

4.4.3 Operational Expressions

Operations allow a memory program to change its own implicit state as well as the state of the memory environment. Based on the program's implicit state, the operations access the correct type of memory and select the processor memory environment of the streaming multiprocessor the threads that issued the request belong to. Additionally, a program uses operations to release blocked registers and to write the result of a read operation to the destination registers. Furthermore, shared memory banks are locked and unlocked using the corresponding operational expression.

$$
\begin{aligned}
\mathit{OpExp} ::= {} & \texttt{readMem} \mid \texttt{writeMem} \mid \texttt{atomOp} \mid \texttt{releaseRegs} \mid \texttt{writeToRegs}\\
& \mid \texttt{lock} \mid \texttt{unlock}\\
& \mid \texttt{readL1} \mid \texttt{readL2} \mid \texttt{writeL1} \mid \texttt{writeL2}\\
& \mid \texttt{reserveL1} \mid \texttt{reserveL2} \mid \texttt{updateL1} \mid \texttt{updateL2}\\
& \mid \texttt{evictL1} \mid \texttt{evictL2} \mid \texttt{discardL1} \mid \texttt{discardL2}\\
& \mid \texttt{classifyFirstL1} \mid \texttt{classifyFirstL2}\\
& \mid \texttt{classifyNormalL1} \mid \texttt{classifyNormalL2}
\end{aligned}
$$

In contrast to Boolean expressions, operations are generally not instantaneous; some of them are guaranteed to be instantaneous and others might defer execution depending on the state of the memory environment. The function $\mathfrak{O}[\![-]\!]$ returns a yield action to indicate whether execution should continue. In any case, an operation never terminates a program. Typically, operations that change the state of the memory environment do so using data stored in the program's implicit state, whereas operations that update the program's implicit state do not alter the state of the memory environment.

We define the semantics for operational expressions as follows. To keep the semantics function $\mathfrak{O}\llbracket - \rrbracket$ as concise as possible, it depends on several helper functions, tagged with the subscript *op*, that hook up the current states of the memory environment and the program to the functions that actually perform the operations. The cache and memory operations are defined in the preceding sections, and we define the helper functions that $\mathfrak{O}\llbracket - \rrbracket$ depends on later on. But as can already be seen from the definition of $\mathfrak{O}\llbracket - \rrbracket$, only cache operations and shared memory bank locking potentially defer execution; reads and writes of memory, atomic operations as well as register writes and unblocking are always instantaneous. As for Boolean expressions, the semantics of operational expressions return $\perp$ whenever a program's state does not permit the execution of an operation.

$$
\begin{array}{rcll}
\multicolumn{4}{l}{\mathfrak{O}\llbracket - \rrbracket : \mathit{OpExp} \to (\mathit{MemEnv} \times \Sigma \to (\mathit{MemEnv} \times \Sigma \times \mathit{YieldAction})_\perp)} \\
\mathfrak{O}\llbracket \texttt{readMem} \rrbracket \eta\sigma & = & (\eta, \mathit{readMem}_{op}(\eta, \sigma), \text{continue}) & \\
\mathfrak{O}\llbracket \texttt{writeMem} \rrbracket \eta\sigma & = & (\mathit{writeMem}_{op}(\eta, \sigma), \sigma, \text{continue}) & \\
\mathfrak{O}\llbracket \texttt{atomOp} \rrbracket \eta\sigma_a & = & (\eta, \mathit{atom}_{op}(\sigma_a), \text{continue}) & \\
\mathfrak{O}\llbracket \texttt{releaseRegs} \rrbracket \eta\sigma & = & (\mathit{releaseRegs}_{op}(\eta, \sigma), \sigma, \text{continue}) & \\
\mathfrak{O}\llbracket \texttt{writeToRegs} \rrbracket \eta\sigma & = & (\mathit{writeToRegs}_{op}(\eta, \sigma), \sigma, \text{continue}) & \\
\mathfrak{O}\llbracket \texttt{lock} \rrbracket \eta\sigma_a & = & \mathit{lock}_{op}(\eta, \sigma_a) & \text{if } \sigma_{a,ssa} = \texttt{.shared} \\
\mathfrak{O}\llbracket \texttt{unlock} \rrbracket \eta\sigma_a & = & (\mathit{unlock}_{op}(\eta, \sigma_a), \sigma_a, \text{continue}) & \text{if } \sigma_{a,ssa} = \texttt{.shared} \\
\mathfrak{O}\llbracket \texttt{readL1} \rrbracket \eta\sigma_d & = & \mathit{L1}_{op}(\eta, \sigma_{d,pid}, \mathit{readCache}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{readL2} \rrbracket \eta\sigma_d & = & \mathit{L2}_{op}(\eta, \mathit{readCache}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{writeL1} \rrbracket \eta\sigma_d & = & \mathit{L1}_{op}(\eta, \sigma_{d,pid}, \mathit{writeCache}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{writeL2} \rrbracket \eta\sigma_d & = & \mathit{L2}_{op}(\eta, \mathit{writeCache}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{reserveL1} \rrbracket \eta\sigma_d & = & \mathit{L1}_{op}(\eta, \sigma_{d,pid}, \mathit{reserve}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{reserveL2} \rrbracket \eta\sigma_d & = & \mathit{L2}_{op}(\eta, \mathit{reserve}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{updateL1} \rrbracket \eta\sigma_d & = & \mathit{L1}_{op}(\eta, \sigma_{d,pid}, \mathit{update}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{updateL2} \rrbracket \eta\sigma_d & = & \mathit{L2}_{op}(\eta, \mathit{update}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{evictL1} \rrbracket \eta\sigma_d & = & \mathit{L1}_{op}(\eta, \sigma_{d,pid}, \mathit{evict}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{evictL2} \rrbracket \eta\sigma_d & = & \mathit{L2}_{op}(\eta, \mathit{evict}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{discardL1} \rrbracket \eta\sigma_d & = & \mathit{L1}_{op}(\eta, \sigma_{d,pid}, \mathit{discard}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{discardL2} \rrbracket \eta\sigma_d & = & \mathit{L2}_{op}(\eta, \mathit{discard}_{op}(\sigma_d)) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{classifyFirstL1} \rrbracket \eta\sigma_d & = & \mathit{L1}_{op}(\eta, \sigma_{d,pid}, \mathit{classify}_{op}(\sigma_d, \text{first})) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{classifyFirstL2} \rrbracket \eta\sigma_d & = & \mathit{L2}_{op}(\eta, \mathit{classify}_{op}(\sigma_d, \text{first})) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{classifyNormalL1} \rrbracket \eta\sigma_d & = & \mathit{L1}_{op}(\eta, \sigma_{d,pid}, \mathit{classify}_{op}(\sigma_d, \text{normal})) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \texttt{classifyNormalL2} \rrbracket \eta\sigma_d & = & \mathit{L2}_{op}(\eta, \mathit{classify}_{op}(\sigma_d, \text{normal})) & \text{if } \sigma_{d,cop_\varepsilon} \neq \varepsilon \\
\mathfrak{O}\llbracket \mathit{opExp} \rrbracket \eta\sigma & = & \perp & \text{otherwise}
\end{array}
$$

In the remainder of this section, we define the helper functions in the order they are used in the above definition of $\mathfrak{O}\llbracket - \rrbracket$. Most of these functions dissect the program's state and subsequently use the cache and memory operations presented in the previous sections to perform the actual operation.

The function $\mathit{readMem}_{op}$ invokes the read functions either for shared memory or DRAM depending on the program's state. The value read by $\mathit{readMem}_{op}$ is copied into the program state while the memory environment is left unchanged. For atomic memory program states, the value retrieved from memory is written to the Value_{out} domain.

$$
\begin{aligned}
&\mathit{readMem}_{op} : \mathit{MemEnv} \times \Sigma \to \Sigma_{\perp}\\
&\mathit{readMem}_{op}(\eta, \sigma_a \triangleright \mathit{pAddr}, \mathit{size}, \texttt{.global}) = \sigma_a \triangleleft \mathit{read}_{device}(\eta, \texttt{.global}, \mathit{pAddr}, \mathit{size})\\
&\mathit{readMem}_{op}(\eta, \sigma_a \triangleright \mathit{pAddr}, \mathit{size}, \mathit{pid}, \texttt{.shared}) = \sigma_a \triangleleft \mathit{read}_{shared}(\eta, \mathit{pid}, \mathit{pAddr}, \mathit{size})_0\\
&\mathit{readMem}_{op}(\eta, \sigma_s \triangleright \mathit{pid}, \overrightarrow{\mathit{pAddr}}, \overrightarrow{\mathit{size}}) = \sigma_s \triangleleft \mathit{read}_{shared}(\eta, \mathit{pid}, \overrightarrow{\mathit{pAddr}}, \overrightarrow{\mathit{size}})\\
&\mathit{readMem}_{op}(\eta, \sigma_d \triangleright \mathit{pAddr}, \mathit{size}, \mathit{ssd}) = \sigma_d \triangleleft \mathit{read}_{device}(\eta, \mathit{ssd}, \mathit{pAddr}, \mathit{size})
\end{aligned}
$$

The function $\mathit{writeMem}_{op}$ invokes the write functions either for shared memory or DRAM depending on the program's state. It does not alter the program state but instead changes the memory environment. For atomic memory program states, the written value is stored in the first Value_{in} domain.

$$
\begin{aligned}
&\mathit{writeMem}_{op} : \mathit{MemEnv} \times \Sigma \to \mathit{MemEnv}_{\perp}\\
&\mathit{writeMem}_{op}(\eta, \sigma_a \triangleright \mathit{pAddr}, \mathit{size}, \texttt{.global}, v_{in}) = \mathit{write}_{device}(\eta, \texttt{.global}, \mathit{pAddr}, \mathit{size}, v_{in})\\
&\mathit{writeMem}_{op}(\eta, \sigma_a \triangleright \mathit{pAddr}, \mathit{size}, \mathit{pid}, \texttt{.shared}, v_{in}) = \mathit{write}_{shared}(\eta, \mathit{pid}, \mathit{pAddr}, \mathit{size}, v_{in})\\
&\mathit{writeMem}_{op}(\eta, \sigma_s \triangleright \mathit{pid}, \overrightarrow{\mathit{pAddr}}, \overrightarrow{\mathit{size}}, \overrightarrow{v_{i/o}}) = \mathit{write}_{shared}(\eta, \mathit{pid}, \overrightarrow{\mathit{pAddr}}, \overrightarrow{\mathit{size}}, \overrightarrow{v_{i/o}})\\
&\mathit{writeMem}_{op}(\eta, \sigma_d \triangleright \mathit{pAddr}, \mathit{size}, \mathit{ssd}, v_{i/o}) = \mathit{write}_{device}(\eta, \mathit{ssd}, \mathit{pAddr}, \mathit{size}, v_{i/o})
\end{aligned}
$$

The function atom_{op} performs an atomic reduction on the data previously retrieved from the memory environment and on the value specified by the issuing thread. Before a memory program can use the `atomOp` instruction, it must load the value on which to perform the atomic operation into the program's state. After the call to `atomOp`, the program must write back the computed value. The program has to guarantee atomicity of these three operations, hence they must be executed in the same micro step or shared memory bank locking must be used. The memory programs for atomic operations on shared and global memory are presented in section 4.5. The semantics of the atomic reductions supported by CUDA are specified in [9, Table 105] and [3, B.11]; we introduce the abstract function *aop* to perform the atomic reductions in accordance with CUDA's atomic reduction semantics.

$$
\begin{aligned}
&\mathit{atom}_{op} : \Sigma_{atomic} \to \Sigma_{atomic}\\
&\mathit{atom}_{op}(\sigma_a \triangleright \mathit{aop}, v_{out}, v_{in,1}, v_{in,2}) = \sigma_a \triangleleft v_{out}, \mathit{aop}(v_{out}, v_{in,1}, v_{in,2}), v_{in,2}
\end{aligned}
$$

For reasons of clarity, we split the implementation of $\mathit{releaseRegs}_{op}$ into two functions. The first one unblocks all of the given registers in the given register file. The second one selects the register file of the streaming multiprocessor stored in the program's state. Then it uses the first function to unblock the registers affected by the program and writes the updated register file back into the memory environment.

$$
\begin{aligned}
&\mathit{releaseRegs}_{op} : \mathit{RegFile} \times \mathit{RegAddr}^{*} \to \mathit{RegFile}\\
&\mathit{releaseRegs}_{op}(\mathit{regFile}, \mathit{regAddr}_1 \ldots \mathit{regAddr}_n) =\\
&\quad \mathit{regFile}[\mathit{regAddr}_1 \ldots \mathit{regAddr}_n \mapsto \mathit{regFile}(\mathit{regAddr}_1) \triangleleft \mathrm{ready} \ldots \mathit{regFile}(\mathit{regAddr}_n) \triangleleft \mathrm{ready}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{releaseRegs}_{op} : \mathit{MemEnv} \times \Sigma \longrightarrow \mathit{MemEnv}\\
&\mathit{releaseRegs}_{op}(\eta \triangleright \overrightarrow{\mathit{procMem}}, \sigma \triangleright \mathit{pid}, \overrightarrow{\mathit{regAddr}}) =\\
&\qquad\qquad \eta \triangleleft \overrightarrow{\mathit{procMem}}[\mathit{pid} \mapsto \overrightarrow{\mathit{procMem}}_{\mathit{pid}} \triangleleft \mathit{releaseRegs}_{op}(\mathit{regFile}(\eta, \mathit{pid}), \overrightarrow{\mathit{regAddr}})]
\end{aligned}
$$

writeToRegs$_{op}$ copies the value retrieved from the memory environment to the destination registers of the memory program. To do this, the function invokes the memory program's decomposition function, if there is one. All *writeToRegs*$_{op}$ does itself is to select the register file corresponding to the memory program's streaming multiprocessor and to update it accordingly. The decomposition function does not unblock any registers.

$$
\begin{aligned}
&\mathit{writeToRegs}_{op} : \mathit{MemEnv} \times \Sigma \longrightarrow \mathit{MemEnv}_{\perp}\\
&\mathit{writeToRegs}_{op}(\eta \triangleright \overrightarrow{\mathit{procMem}}, \sigma \triangleright \mathit{pid}, \mathit{df}) = \eta \triangleleft \overrightarrow{\mathit{procMem}}[\mathit{pid} \mapsto \overrightarrow{\mathit{procMem}}_{\mathit{pid}} \triangleleft \mathit{regFile}]\\
&\qquad\qquad\qquad\qquad \text{where} \quad \mathit{regFile} = \mathit{df}(\mathit{regFile}(\eta, \mathit{pid}), \sigma)\\
&\mathit{writeToRegs}_{op}(\eta, \sigma) = \perp \quad \text{otherwise}
\end{aligned}
$$

The function *lock*$_{op}$ locks the banks the requested shared memory locations belong to. If one of the banks is already locked, the program yields execution and tries again during the next micro step.

$$
\begin{aligned}
&\mathit{lock}_{op} : \mathit{MemEnv} \times \Sigma_{\mathit{atomic}} \longrightarrow \mathit{MemEnv} \times \Sigma_{\mathit{atomic}} \times \mathit{YieldAction}\\
&\mathit{lock}_{op}(\eta \triangleright \overrightarrow{\mathit{procMem}}, \sigma_a \triangleright \mathit{pAddr}, \mathit{size}, \mathit{pid}) =\\
&\qquad\qquad (\eta \triangleleft \overrightarrow{\mathit{procMem}}[\mathit{pid} \mapsto \overrightarrow{\mathit{procMem}}_{\mathit{pid}} \triangleleft s], \sigma_a, \text{continue})\\
&\qquad \text{where} \quad s' = \mathit{sharedMem}(\eta, \mathit{pid}) \wedge s = s' \triangleleft \pi_2(s')[\mathit{banks}(\mathit{pAddr}, \mathit{size}) \mapsto \text{true}]\\
&\qquad\qquad \text{if} \quad \forall \mathit{bankIdx} \in \mathit{banks}(\mathit{pAddr}, \mathit{size}) \,.\, \pi_2(s')(\mathit{bankIdx}) = \text{false}\\
&\mathit{lock}_{op}(\eta, \sigma_a \triangleright \mathit{pAddr}, \mathit{size}, \mathit{pid}) = (\eta, \sigma_a, \text{yield})\\
&\qquad \text{if} \quad \exists \mathit{bankIdx} \in \mathit{banks}(\mathit{pAddr}, \mathit{size}) \,.\, \pi_2(\mathit{sharedMem}(\eta, \mathit{pid}))(\mathit{bankIdx}) = \text{true}
\end{aligned}
$$

unlock$_{op}$ unlocks the banks accessed by the shared memory request. It is always instantaneous because it simply unlocks all accessed shared memory banks. No deadlocks can occur due to shared memory bank locking as a request either locks or unlocks all accessed banks or none at all.

$$
\begin{aligned}
&\mathit{unlock}_{op} : \mathit{MemEnv} \times \Sigma_{\mathit{atomic}} \longrightarrow \mathit{MemEnv}\\
&\mathit{unlock}_{op}(\eta \triangleright \overrightarrow{\mathit{procMem}}, \sigma_a \triangleright \mathit{pAddr}, \mathit{size}, \mathit{pid}) = \eta \triangleleft \overrightarrow{\mathit{procMem}}[\mathit{pid} \mapsto \overrightarrow{\mathit{procMem}}_{\mathit{pid}} \triangleleft s]\\
&\qquad \text{where} \quad s' = \mathit{sharedMem}(\eta, \mathit{pid}) \wedge s = s' \triangleleft \pi_2(s')[\mathit{banks}(\mathit{pAddr}, \mathit{size}) \mapsto \text{false}]
\end{aligned}
$$

Section 4.2 defines all cache operations over sets of cache lines allowing us to use the same functions to operate on the L1 caches and the L2 cache. Ideally, we would also use one helper function per operational expression to wire up the semantics of the operational expression to the actual cache operation. However, with the L2 cache being embedded directly into the memory environment and the L1 caches being scattered throughout the processor memory environments, the process of selecting and updating the L1 and L2 caches differs significantly. But that is in fact the only difference; once a set of cache lines is selected, calling the actual cache operation and deciding whether execution is instantaneous or deferred works exactly the same regardless of the cache being modified. To avoid the redundancy, we define the hook-up functions only once for both types of caches and then pass these functions to the $L1_{op}$ and $L2_{op}$ functions that perform the actual cache selection and updates. For that matter, we define the domain of cache functions. Such a function manipulates cache lines as well as the state of the memory program and decides whether program execution should continue.

$$cf \in \mathit{CacheFunc} : \mathit{CacheLine}^{*} \rightarrow (\Sigma_{device} \times \mathit{CacheLine}^{*} \times \mathit{YieldAction})_{\perp}$$

Cache functions are passed to $L1_{op}$ and $L2_{op}$. There, the corresponding L1 cache or the L2 cache is selected and the cache lines are passed to the provided cache function. The values returned by the cache function are subsequently used to update the selected cache. The functions then return the updated memory environment and program state as well as the yield action generated by the cache function.

$$\begin{array}{l}
L1_{op} : \mathit{MemEnv} \times \mathit{ProcIdx} \times \mathit{CacheFunc} \rightarrow (\mathit{MemEnv} \times \Sigma \times \mathit{YieldAction})_{\perp} \\
L1_{op}(\eta \triangleright \overrightarrow{procMem}, pid, cf) = (\eta \triangleleft \overrightarrow{procMem}[pid \mapsto \overrightarrow{procMem}_{pid} \triangleleft \overrightarrow{cl}\,], \sigma_d, ya) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where} \quad cf(l1Cache(\eta, pid)) = (\sigma_d, \overrightarrow{cl}, ya) \\
L1_{op}(\eta, pid, cf) = \perp \quad \text{otherwise}
\end{array}$$

$$\begin{array}{lll}
L2_{op} : \mathit{MemEnv} \times \mathit{CacheFunc} \rightarrow (\mathit{MemEnv} \times \Sigma \times \mathit{YieldAction})_{\perp} & & \\
L2_{op}(\eta \triangleright l2Cache, cf) = (\eta \triangleleft \overrightarrow{cl}, \sigma_d, ya) & \text{where} & cf(l2Cache) = (\sigma_d, \overrightarrow{cl}, ya) \\
\qquad L2_{op}(\eta, pid, cf) = \perp & \text{otherwise} &
\end{array}$$

The following helper functions are passed to $L1_{op}$ and $L2_{op}$ using function currying. In general, they call the cache operations defined in section 4.2 and decide whether the memory program is allowed to continue execution. Otherwise, the program yields execution and retries at a later point in time. Some cache operations, like eviction reclassification, are always instantaneous however. The conversion of an element of the domain $Value_{in/out}$ into an element of the domain *CacheWord* always succeeds without a data loss because a cache line is big enough to store the largest possible memory request.

$$\begin{array}{lll}
readCache_{op} : \Sigma_{device} \rightarrow \mathit{CacheFunc} & & \\
readCache_{op}(\sigma_d \triangleright pAddr, mid)(\overrightarrow{cl}) = (\sigma_d \triangleleft cw, \overrightarrow{cl'}, \text{continue}) & & \\
 & \text{if} & (\overrightarrow{cl'}, cw) = read(\overrightarrow{cl}, pAddr, mid) \\
readCache_{op}(\sigma_d \triangleright pAddr, mid)(\overrightarrow{cl}) = (\sigma_d, \overrightarrow{cl'}, \text{yield}) & \text{if} & (\overrightarrow{cl'}, \varepsilon) = read(\overrightarrow{cl}, pAddr, mid) \\
\qquad\qquad readCache_{op}(\sigma_d)(\overrightarrow{cl}) = \perp & \text{otherwise} &
\end{array}$$

$$\begin{array}{lll}
writeCache_{op} : \Sigma_{device} \rightarrow \mathit{CacheFunc} & & \\
writeCache_{op}(\sigma_d \triangleright pAddr, v_{i/o})(\overrightarrow{cl}) = (\sigma_d, \overrightarrow{cl'}, \text{continue}) & & \\
 & \text{if} & \overrightarrow{cl'} = write(\overrightarrow{cl}, pAddr, v_{i/o}) \wedge \overrightarrow{cl'} \neq \varepsilon \\
writeCache_{op}(\sigma_d \triangleright pAddr, v_{i/o})(\overrightarrow{cl}) = (\sigma_d, \overrightarrow{cl}, \text{yield}) & \text{if} & \varepsilon = write(\overrightarrow{cl}, pAddr, v_{i/o}) \\
\qquad\qquad writeCache_{op}(\sigma_d)(\overrightarrow{cl}) = \perp & \text{otherwise} &
\end{array}$$

$$\begin{array}{ll}
\mathit{reserve}_{op} : \Sigma_{device} \rightarrow \mathit{CacheFunc} & \\
\mathit{reserve}_{op}(\sigma_d \triangleright \mathit{pAddr}, \mathit{mid})(\overrightarrow{cl}) = (\sigma_d, cl \circ \overrightarrow{cl'}, \text{continue}) & \text{if } (cl, \overrightarrow{cl'}) = \mathit{reserve}(\overrightarrow{cl}, \mathit{pAddr}, \mathit{mid}) \\
\mathit{reserve}_{op}(\sigma_d \triangleright \mathit{pAddr}, \mathit{mid})(\overrightarrow{cl}) = (\sigma_d, \overrightarrow{cl}, \text{yield}) & \text{if } (\varepsilon, \varepsilon) = \mathit{reserve}(\overrightarrow{cl}, \mathit{pAddr}, \mathit{mid}) \\
\mathit{reserve}_{op}(\sigma_d)(\overrightarrow{cl}) = \bot & \text{otherwise}
\end{array}$$

$$\begin{array}{l}
\mathit{update}_{op} : \Sigma_{device} \rightarrow \mathit{CacheFunc} \\
\mathit{update}_{op}(\sigma_d \triangleright \mathit{pAddr}, v_{i/o})(\overrightarrow{cl}) = (\sigma_d, \mathit{update}(\overrightarrow{cl}, \mathit{pAddr}, v_{i/o}), \text{continue})
\end{array}$$

$$\begin{array}{ll}
\mathit{evict}_{op} : \Sigma_{device} \rightarrow \mathit{CacheFunc} & \\
\mathit{evict}_{op}(\sigma_d \triangleright \mathit{pAddr})(\overrightarrow{cl}) = (\sigma_d, \overrightarrow{cl'}, \text{continue}) & \text{if} \quad \overrightarrow{cl'} = \mathit{evict}(\overrightarrow{cl}, \mathit{pAddr}) \wedge \overrightarrow{cl'} \neq \varepsilon \\
\mathit{evict}_{op}(\sigma_d \triangleright \mathit{pAddr})(\overrightarrow{cl}) = (\sigma_d, \overrightarrow{cl}, \text{yield}) & \text{if} \quad \varepsilon = \mathit{evict}(\overrightarrow{cl}, \mathit{pAddr}) \\
\mathit{evict}_{op}(\sigma_d)(\overrightarrow{cl}) = \bot & \text{otherwise}
\end{array}$$

$$\begin{array}{ll}
\mathit{discard}_{op} : \Sigma_{device} \rightarrow \mathit{CacheFunc} & \\
\mathit{discard}_{op}(\sigma_d \triangleright \mathit{pAddr})(\overrightarrow{cl}) = (\sigma_d, \overrightarrow{cl'}, \text{continue}) & \text{if} \quad \overrightarrow{cl'} = \mathit{discard}(\overrightarrow{cl}, \mathit{pAddr}) \wedge \overrightarrow{cl'} \neq \varepsilon \\
\mathit{discard}_{op}(\sigma_d \triangleright \mathit{pAddr})(\overrightarrow{cl}) = (\sigma_d, \overrightarrow{cl}, \text{yield}) & \text{if} \quad \varepsilon = \mathit{discard}(\overrightarrow{cl}, \mathit{pAddr}) \\
\mathit{discard}_{op}(\sigma_d)(\overrightarrow{cl}) = \bot & \text{otherwise}
\end{array}$$

$$\begin{array}{l}
\mathit{classify}_{op} : \Sigma_{device} \times \mathit{EvClass} \rightarrow \mathit{CacheFunc} \\
\mathit{classify}_{op}(\sigma_d \triangleright \mathit{pAddr}, \mathit{ec})((cl \triangleright \mathit{pAddr}) \circ \overrightarrow{cl}) = (\sigma_d, (cl \triangleleft \mathit{ec}) \circ \overrightarrow{cl}, \text{continue})
\end{array}$$

4.4.4 Program Statements

To complete the formalization of memory programs, we have to define the grammar and semantics of program statements. As already mentioned above, we keep memory programs as simple as possible and hide most of the complexities behind the semantics of operations: For instance, we do not support variables and variable assignment, because the operations implicitly use the program's state to store input and result values. Programs also do not support while-loops, as they are subsumed by the combination of `yield` statements and deferred execution.

$$\begin{array}{rl}
\mathit{stm} \in \mathit{MemStm} ::= & \texttt{;} \mid \texttt{yield;} \mid \texttt{end;} \\
& \mid \mathit{MemStm}\ \mathit{MemStm} \\
& \mid \texttt{if (}\mathit{BExp}\texttt{) \{ }\mathit{MemStm}\texttt{ \} else \{ }\mathit{MemStm}\texttt{ \}} \\
& \mid \mathit{OpExp}\texttt{;}
\end{array}$$

For convenience, we establish the following convention when writing memory programs with case distinctions: If the else path of an if-else statement is empty, i.e. contains only `;`, we can omit the else. Thus, the two statements `if` (*bExp*) `{` *memStm* `} else { ; }` and `if` (*bExp*) `{` *memStm* `}` are equivalent.

Transition system $\rightarrow^{\mathrm{Op}}$ defines the semantics of memory program statements. Each rule outputs a yield action that indicates whether program execution should continue, the micro step is completed, or the program has terminated.

$$\begin{aligned} \mathit{Configurations} &: \langle \mathit{stm}, \eta, \sigma \rangle \in \mathit{MemStm} \times \mathit{MemEnv} \times \Sigma, \qquad \langle \eta, \sigma \rangle \in \mathit{MemEnv} \times \Sigma \\ \mathit{Transitions} &: \langle \mathit{stm}, \eta, \sigma \rangle \xrightarrow[ya]{}^{\mathrm{Op}} \langle \mathit{stm}', \eta', \sigma' \rangle, \qquad \langle \mathit{stm}, \eta, \sigma \rangle \xrightarrow[ya]{}^{\mathrm{Op}} \langle \eta', \sigma' \rangle \end{aligned}$$

Some transitions do not definitely result in either a configuration of type $\langle \mathit{stm}, \eta, \sigma \rangle$ or $\langle \eta, \sigma \rangle$, but rather both configurations are possible depending on the state of the executed program. Thus, a rule using such a transition would have to be split into two rules, one for each case. By convention, we only define one rule and use the notation $\langle [\mathit{stm},]\ \eta, \sigma \rangle$ to mean that either of the two configurations could be the result of the rule.

The remainder of this section presents the rules of transition system $\rightarrow^{\mathrm{Op}}$. The skip instruction `;` neither changes the state of the program nor the state of the memory environment. Furthermore, it does not end the micro step thus program execution continues.

(nop) $\langle \texttt{;}, \eta, \sigma \rangle \xrightarrow[\text{continue}]{}^{\mathrm{Op}} \langle \eta, \sigma \rangle$

Like skip, `yield` does not change the memory environment or program state. However, it ends the current micro step; the instruction following the yield will be executed during the next micro step of the program.

(yield) $\langle \texttt{yield;}, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}^{\mathrm{Op}} \langle \eta, \sigma \rangle$

Like skip and `yield`, `end` leaves all states unchanged and terminates the program, i.e. the program will not perform any further micro steps. `end` also ends the current micro step.

(end) $\langle \texttt{end;}, \eta, \sigma \rangle \xrightarrow[\text{end}]{}^{\mathrm{Op}} \langle \eta, \sigma \rangle$

There are three possible cases that need to be considered when defining the semantics of sequential composition of program statements. First of all, the first statement might yield execution. In that case, we have to execute the remainder of the first statement — if there is one — during the next micro step. We do not execute the second statement and make sure we execute it once the first statement is fully processed. Secondly, the first statement might execute an `end` instruction. In this case, the program terminates so there is nothing left to execute. Finally, the execution of the first statement might complete during the current micro step without yielding execution, thus we have to start executing the second one. It is possible that the second statement is also fully completed during the current micro step, otherwise its execution has to be continued during the program's next micro step.

$$(\mathrm{seq}_1)\ \frac{\langle \mathit{stm}_1, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}^{\mathrm{Op}} \langle [\mathit{stm}_1',]\ \eta', \sigma' \rangle}{\langle \mathit{stm}_1\ \mathit{stm}_2, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}^{\mathrm{Op}} \langle [\mathit{stm}_1']\ \mathit{stm}_2, \eta', \sigma' \rangle}$$

$$(\mathrm{seq}_2)\ \frac{\langle \mathit{stm}_1, \eta, \sigma \rangle \xrightarrow[\text{end}]{}^{\mathrm{Op}} \langle \eta', \sigma' \rangle}{\langle \mathit{stm}_1\ \mathit{stm}_2, \eta, \sigma \rangle \xrightarrow[\text{end}]{}^{\mathrm{Op}} \langle \eta', \sigma' \rangle}$$

$$(\text{seq}_3)\quad \frac{\langle \mathit{stm}_1, \eta, \sigma\rangle \xrightarrow[\text{continue}]{}{}^{\text{Op}} \langle \eta'', \sigma''\rangle \qquad \langle \mathit{stm}_2, \eta'', \sigma''\rangle \xrightarrow[\mathit{ya}]{}{}^{\text{Op}} \langle [\mathit{stm}_2',]\ \eta', \sigma'\rangle}{\langle \mathit{stm}_1\ \mathit{stm}_2, \eta, \sigma\rangle \xrightarrow[\mathit{ya}]{}{}^{\text{Op}} \langle [\mathit{stm}_2',]\ \eta', \sigma'\rangle}$$

Either the if or the else path of an if-else statement is executed depending on the value of the Boolean expression. As mentioned above, Boolean expressions are always instantaneous, hence the evaluation of the expression cannot possibly cause the program to yield execution. The statement of the path taken can either be fully executed during the current micro step or needs to continue execution during the next one. If the evaluation of the Boolean expression is unsuccessful, i.e. $\mathfrak{B}\llbracket - \rrbracket$ returns $\bot$, neither of the following two rules is applicable and the program is stuck.

$$(\text{if}_{\text{tt}})\quad \frac{\langle \mathit{stm}_1, \eta, \sigma\rangle \xrightarrow[\mathit{ya}]{}{}^{\text{Op}} \langle [\mathit{stm}_1',]\ \eta', \sigma'\rangle}{\langle \texttt{if (}\mathit{bExp}\texttt{) \{ }\mathit{stm}_1\texttt{ \} else \{ }\mathit{stm}_2\texttt{ \}}, \eta, \sigma\rangle \xrightarrow[\mathit{ya}]{}{}^{\text{Op}} \langle [\mathit{stm}_1',]\ \eta', \sigma'\rangle} \qquad \text{if} \quad \mathfrak{B}\llbracket \mathit{bExp} \rrbracket \eta\sigma$$

$$(\text{if}_{\text{ff}})\quad \frac{\langle \mathit{stm}_2, \eta, \sigma\rangle \xrightarrow[\mathit{ya}]{}{}^{\text{Op}} \langle [\mathit{stm}_2',]\ \eta', \sigma'\rangle}{\langle \texttt{if (}\mathit{bExp}\texttt{) \{ }\mathit{stm}_1\texttt{ \} else \{ }\mathit{stm}_2\texttt{ \}}, \eta, \sigma\rangle \xrightarrow[\mathit{ya}]{}{}^{\text{Op}} \langle [\mathit{stm}_2',]\ \eta', \sigma'\rangle} \qquad \text{if} \quad \neg\mathfrak{B}\llbracket \mathit{bExp} \rrbracket \eta\sigma$$

Executing an operational expression typically either changes the memory environment or the program state. For instantaneous operations, program execution continues with the next instruction. If an operation defers execution, the micro step ends and the operation is attempted again during the next macro step the program participates in. Executing an operation can never terminate a program in accordance with the definition of $\mathfrak{O}\llbracket - \rrbracket$. If the operation cannot be performed, i.e. $\mathfrak{O}\llbracket - \rrbracket$ returns $\bot$, the program is stuck because neither of the following two rules is applicable.

$$(\text{op}_1)\quad \langle \mathit{opExp}\texttt{;}, \eta, \sigma\rangle \xrightarrow[\text{continue}]{}{}^{\text{Op}} \langle \eta', \sigma'\rangle \qquad \text{if} \quad \mathfrak{O}\llbracket \mathit{opExp} \rrbracket \eta\sigma = (\eta', \sigma', \text{continue})$$

$$(\text{op}_2)\quad \langle \mathit{opExp}\texttt{;}, \eta, \sigma\rangle \xrightarrow[\text{yield}]{}{}^{\text{Op}} \langle \mathit{opExp}\texttt{;}, \eta', \sigma'\rangle \qquad \text{if} \quad \mathfrak{O}\llbracket \mathit{opExp} \rrbracket \eta\sigma = (\eta', \sigma', \text{yield})$$

4.5 Memory Operation Semantics

We have already defined the semantics of a store to global memory with the `.wb` cache operation in section 4.1. In this section, we present some additional memory programs to define the semantics of some other memory operations supported by PTX. However, we do not present programs for all supported operations, but only those that illustrate some points worth mentioning. As already stated above, the documentation does not explain how volatile memory operations affect matching cache lines in the caches, hence we leave volatile operations entirely unspecified.

Memory program 4.2 defines the semantics of a read of global or local memory with the `.ca` cache operation. First, we check whether the requested address is already stored in the L1 cache. If so, we read the data from L1 in line 22. We know the data is cached but we do not know whether the cache line is valid, so `readL1` might be instantaneous or it might defer execution. In any case, we yield execution after we have read the value to allow other programs to continue. If program 4.2 executes its next micro step, the results of the read are

```
1  if (!isCachedL1)
2  {
3    reserveL1;
4    yield;
5
6    if (!isCachedL2)
7    {
8      reserveL2;
9      yield;
10
11     readMem;
12     yield;
13
14     updateL2;
15   }
16
17   readL2;
18   yield;
19   updateL1;
20 }
21
22 readL1;
23 yield;
24
25 writeToRegs;
26 yield;
27
28 releaseRegs;
29 end;
```

Listing 4.2: Memory program for global and local reads using the `.ca` cache operation

written to the destination registers. At this point in time, the L1 cache might have already evicted the cache line; this is not a problem, however, as the data read from the cache is stored in the program's implicit state. After yielding, we release the destination registers in line 28 and end the program.

On the other hand, if the L1 cache does not contain a matching cache line, we have to retrieve the value from L2 and cache the data in L1. Thus, we reserve a cache line in line 3. `reserveL1` is either instantaneous or defers execution, depending on whether there is an eligible cache line that is either unused or can be evicted. Reserving also ensures that the cache line cannot be evicted until we read the value in line 22. Once `reserveL1` succeeds, we yield execution. In the next micro step, we check whether the L2 cache contains a matching cache line. If so, we retrieve the value from L2 in line 17 and yield again once the read succeeds. Next we update the reserved L1 cache line. `updateL1` is definitively instantaneous, as the matching cache line is pinned by the program. However, `updateL1` might not write the value retrieved from L2 into L1, because there might have been another memory program that wrote a newer value to L1 while the value was being retrieved from L2. That cannot happen for global data — as global data never writes to L1 —, but it might potentially happen if a thread reads local data and writes to the same location without waiting for the previous read to complete. However, if this situation can indeed arise is unknown. In any case, `readL1` instantaneously reads the value and unpins the cache line and the program continues to execute as outline above.

If the data is not cached in L2 either, the data has to be retrieved from global or local memory and must be cached in L2 first. This works in an analogous manner to updating L1, with the exception that `readMem` is used to retrieve the value from memory.

```
writeL1;
classifyFirstL1;
yield;

releaseRegs;
end;
```

Listing 4.3: Memory program for local writes using the `.cg` cache operation

The semantics of local writes with cache operation `.cg` are defined by program 4.3. The data is written to the L1 cache, which either happens instantaneously or is deferred. Afterwards, the cache line's eviction class is set to first as specified by [9, Table 81]. The program ends after releasing the source registers.

```
evictL1;
yield;

if (!isCachedL2)
{
  reserveL2;
  classifyNormalL2;
  yield;

  readMem;
  yield;

  updateL2;
  yield;
}

readL2;
atomOp;
writeL2;
yield;
writeToRegs;
yield;
releaseRegs;
end;
```

Listing 4.4: Memory program for atomic operations on global memory

Memory program 4.4 defines the semantics of an atomic memory operation on global memory. First, the matching cache line in the L1 cache is evicted — this seems to be a reasonable thing to do, although the PTX specification does not explicitly mention this step. [7, 12] gives a hint that atomic operations are performed by the raster operation units on data cached in L2. Consequently, we check if the requested address is cached in L2 and load it from global memory if no matching cache line is found. This works in a similar manner to program 4.2. However, as we are dealing with atomic operations, micro steps play an important role here. Loading the data into L2 is by no means atomic. After the data is written to L2, the program yields execution and other operations, including other atomic operations, may access the cache line. In any case, the cache line cannot be evicted, because `updateL2` in line 13 does not undo the pinning of the cache line caused by `reserveL2` in line 6. In line 17, program 4.4 executes the core of the atomic operation: The data is read from L2 either instantaneously or at a later point in time if `isCachedL2` was true but the cache line was only reserved. Once the read succeeds, the atomic reduction is performed and the computed value is written back to L2 during the same micro step, i.e. atomically. After the successful write, which is guaranteed to be instantaneous because the address is cached, the micro step ends. During the subsequent micro steps, the destination registers are unblocked and updated with the result values.

```
lock;
readMem;
yield;

atomOp;
yield;

writeMem;
yield;

unlock;
yield;

writeToRegs;
yield;
releaseRegs;
end;
```

Listing 4.5: Memory program for atomic operations on shared memory

Although not officially documented, atomic operations on shared memory are executed differently by the hardware than atomic operations on global memory. As explained in reference to program 4.4, the raster operation units appear to contain dedicated hardware units to process atomic operations. In contrast, atomic operations on shared memory are emulated by the compiler as can be seen by using Nvidia's cuobjdump disassembler to inspect a compiled program containing atomic shared memory instructions. The value at the requested address is first loaded into a register and the computation of the atomic reduction is performed using the regular instruction set of the CUDA cores. Afterwards, the computed value is written to shared memory. The compiler uses a special flag of the load and store instructions to signal to the streaming multiprocessor that all accessed shared memory banks should be locked or unlocked. We assume that loads and stores of shared memory that do not have the lock flag set do not respect locked banks and access them anyway. This would explain why atomic operations on shared memory are not atomic with respect to non-atomic operations on the same address [9, Table 105]. Our formalization treats atomic operations on both global and shared memory as parts of the memory model for reasons of brevity. We are confident that the semantics of the PTX compiler's emulation is equivalent to our formalization.

Program 4.5 defines the formal semantics of atomic operations on shared memory. In the first micro step, the program tries to acquire the lock for the accessed shared memory banks. If any of the banks are already locked, `lock` defers execution and therefore guarantees that only one single atomic operation operates concurrently on the same address. Additionally, no deadlocks can occur because either all or none of the accessed banks are locked. Once the program has acquired the lock, it reads the value, computes the reduction, and writes the result in three separate micro steps. Due to the locking, the program is atomic with respect to other atomic operations accessing the same shared memory locations, but a non-atomic store is indeed able to write a new value to the requested address at some point in between. Once the result is written to shared memory, the banks are unlocked and the program yields again. Releasing and writing to the destination registers is done in the usual way.

```
writeToRegs;
yield;
releaseRegs;
end;
```

Listing 4.6: Memory program for writes to registers

An instance of program 4.6 is launched whenever threads update the value of a register, i.e. for all arithmetic, logic, and shift instructions supported by PTX. As the writes are performed by the decomposition function, there is not much work left for the program. It merely invokes the decomposition function and releases the blocked registers during a subsequent macro step.

4.6 Formal Semantics of the Memory Environment

Memory programs allow us to define the semantics of CUDA memory operations in isolation. They specify which operations to perform on caches and memory and in which order to perform them. Memory program statements are grouped into micro steps; in between two micro steps of some program, any number of other programs may execute any number of micro steps. In this section, we formalize the semantics of the memory environment as a whole. The memory environment either processes a micro step of some program, flushes one of the L1 caches, or evicts or writes-back a cache line of one of the caches. A macro step comprises several of those actions.

The CUDA specification does not state in which order memory operations are processed. For instance, if a thread writes to some location l in global memory and reads location l at the next instruction, it is unclear whether the thread reads the original value or the value it has written one instruction ago. Although tests indicate that the newly written value would be fetched, we do not want to rely on this assumption and rather allow the memory environment to process all in-flight memory operations in any order. The only exception to this are volatile reads and writes. The PTX documentation clearly states that the `.volatile` specifier enforces sequential consistency [9, Table 84], i.e. all volatile operations of the same thread are processed in the order they are issued. To support this scenario, we introduce an abstract priority mechanism for memory operations. The operations with the highest priorities are always processed first, thus the order of volatile operations can be guaranteed. Once more information about CUDA's memory model is available, this mechanism can also be used to give some ordering guarantees of non-volatile operations. Priorities are compared using the abstract quasi-order $\leq$.

$$\pi \in \mathit{Priority}$$

A memory operation therefore consists of a memory program that defines its semantics, a program state the memory program operates on, and a priority that determines when the operation must be processed.

$$op \in \mathit{MemOp} = \mathit{MemStm} \times \Sigma \times \mathit{Priority}$$

We define the function *highestPrio* to check whether some priority exceeds or at least matches all of the priorities in a set of memory operations. Typically, there may be many different priorities for which this function returns true, because most likely non-volatile memory operations that were issued independently by distinct threads are equally prioritized. Again, we do not know the order and thus the prioritization of memory operations in the general case. Referring back to the example above, we do not know whether CUDA's memory model prioritizes a thread's write over the subsequent read of the same address to prevent a read-after-write hazard.

$$\begin{aligned}
&\mathit{highestPrio} : \mathit{Priority} \times \mathit{MemOp}^{*} \to \mathbb{B} \\
&\mathit{highestPrio}(\pi, \overrightarrow{op}) \Leftrightarrow \forall (op \blacktriangleright \pi') \in \overrightarrow{op}\ .\ \pi' \leq \pi
\end{aligned}$$

Transition system $\rightarrow^{\mathrm{M}}$ defines the semantics of the memory environment. It consists of two types of rules: The first type modifies the state of the memory environment according to the semantics of a memory program. The other type are silent rules, i.e. rules that may fire at any time without being explicitly invoked or requested. Examples for the latter type are evictions of L1 or L2 cache lines. The CUDA specification gives no hint as to when and how a cache decides to evict a cache line. Therefore, we have to assume that it can happen at any time. This is the reason why cache lines must be pinned when memory programs request a cache line whose data is being retrieved from a higher level of the memory hierarchy: Without the pinning, the cache line might be evicted before the requesting operation gets a chance to read it. In that case, the program is stuck unless some other operation requests the same cache line again and the stuck program gets a cache to retrieve the value before it is evicted again. The pinning mechanism avoids this issue.

$$\begin{aligned}
\mathit{Configurations} &: \langle \eta \mid \overrightarrow{op} \rangle \in \mathit{MemEnv} \times \mathit{MemOp}^{*} \\
\mathit{Transitions} &: \langle \eta \mid \overrightarrow{op} \rangle \rightarrow^{\mathrm{M}} \langle \eta' \mid \overrightarrow{op'} \rangle
\end{aligned}$$

The (step) and (end) rules select some memory operation which has one of the highest priorities and execute the next micro step of the operation's program. If the program ends, the memory operation is completed and is therefore removed from the list of in-flight operations.

$$\text{(step)}\quad \frac{\langle stm, \eta, \sigma \rangle \xrightarrow[\text{yield}]{}{}^{\mathrm{Op}} \langle stm', \eta', \sigma' \rangle}{\langle \eta \mid (op \blacktriangleright stm, \sigma, \pi) \circ \overrightarrow{op} \rangle \rightarrow^{\mathrm{M}} \langle \eta' \mid (op \blacktriangleleft stm', \sigma') \circ \overrightarrow{op} \rangle} \qquad \text{if} \quad \mathit{highestPrio}(\pi, \overrightarrow{op})$$

$$\text{(end)}\quad \frac{\langle stm, \eta, \sigma \rangle \xrightarrow[\text{end}]{}{}^{\mathrm{Op}} \langle \eta', \sigma' \rangle}{\langle \eta \mid (op \blacktriangleright stm, \sigma, \pi) \circ \overrightarrow{op} \rangle \rightarrow^{\mathrm{M}} \langle \eta' \mid \overrightarrow{op} \rangle} \qquad \text{if} \quad \mathit{highestPrio}(\pi, \overrightarrow{op})$$

The remaining rules are all silent rules which can be chosen at any time, provided their side conditions hold. All of the streaming multiprocessors' L1 caches as well as the L2 cache are allowed to evict some non-dirty cache line at any time in accordance with the eviction policy.

$$(\text{evict}_{\text{L1}})\quad \langle \eta \triangleright \overrightarrow{\mathit{procMem}} \mid \overrightarrow{\mathit{op}} \rangle \rightarrow^{\text{M}} \langle \eta \triangleleft \overrightarrow{\mathit{procMem}}[\mathit{pid} \mapsto \overrightarrow{\mathit{procMem}}_{\mathit{pid}} \triangleleft \overrightarrow{\mathit{cl}}] \mid \overrightarrow{\mathit{op}} \rangle$$
$$\text{if}\quad \mathit{canEvict}(\mathit{l1Cache}(\eta, \mathit{pid}), \mathit{pAddr}) \land \overrightarrow{\mathit{cl}} = \mathit{evict}(\mathit{l1Cache}(\eta, \mathit{pid}), \mathit{pAddr}) \land \overrightarrow{\mathit{cl}} \neq \varepsilon$$

$$(\text{evict}_{\text{L2}})\quad \langle \eta \triangleright \mathit{l2Cache} \mid \overrightarrow{\mathit{op}} \rangle \rightarrow^{\text{M}} \langle \eta \triangleleft \overrightarrow{\mathit{cl}} \mid \overrightarrow{\mathit{op}} \rangle$$
$$\text{if}\quad \mathit{canEvict}(\mathit{l2Cache}, \mathit{pAddr}) \land \overrightarrow{\mathit{cl}} = \mathit{evict}(\mathit{l2Cache}, \mathit{pAddr}) \land \overrightarrow{\mathit{cl}} \neq \varepsilon$$

Similarly, one of the L1 caches can write back a value to the L2 cache at any time. However, a write-back is only possible if the write to the L2 cache succeeds instantaneously. Otherwise, a write-back can only be performed after evicting an eligible line from L2 first. A write-back of some L2 cache line is always possible as writes to global or local memory are guaranteed to be instantaneous.

$$(\text{wb}_{\text{L1}})\quad \langle \eta \triangleright \mathit{l2Cache}, \overrightarrow{\mathit{procMem}} \mid \overrightarrow{\mathit{op}} \rangle \rightarrow^{\text{M}}$$
$$\langle \eta \triangleleft \overrightarrow{\mathit{cl}_2}, \overrightarrow{\mathit{procMem}}[\mathit{pid} \mapsto \overrightarrow{\mathit{procMem}}_{\mathit{pid}} \triangleleft (\mathit{pAddr}, \mathit{cw}, \text{false}, \text{true}, \overrightarrow{\mathit{mid}}, \mathit{ec}) \circ \overrightarrow{\mathit{cl}_1}] \mid \overrightarrow{\mathit{op}} \rangle$$
$$\text{if}\quad (\mathit{pAddr}, \mathit{cw}, \text{true}, \text{true}, \overrightarrow{\mathit{mid}}, \mathit{ec}) \circ \overrightarrow{\mathit{cl}_1} = \mathit{l1Cache}(\eta, \mathit{pid})$$
$$\land\ \overrightarrow{\mathit{cl}_2} = \mathit{write}(\mathit{l2Cache}, \mathit{pAddr}, \mathit{cw}) \land \overrightarrow{\mathit{cl}_2} \neq \varepsilon$$

$$(\text{wb}_{\text{L2}})\quad \langle \eta \triangleright (\mathit{pAddr}, \mathit{cw}, \text{true}, \text{true}, \overrightarrow{\mathit{mid}}, \mathit{ec}) \circ \overrightarrow{\mathit{cl}} \mid \overrightarrow{\mathit{op}} \rangle \rightarrow^{\text{M}}$$
$$\langle \mathit{write}_{\mathit{device}}(\eta, \mathit{ssd}, \mathit{pAddr}, \mathit{MaxMemOpSize}, \mathit{cw}) \triangleleft (\mathit{pAddr}, \mathit{cw}, \text{false}, \text{true}, \overrightarrow{\mathit{mid}}, \mathit{ec}) \circ \overrightarrow{\mathit{cl}} \mid \overrightarrow{\mathit{op}} \rangle$$
$$\text{where}\quad \mathit{ssd} = \begin{cases} \texttt{.global} & \text{if}\quad \mathit{pAddr} \in \mathit{GlobalAddr} \\ \texttt{.local} & \text{if}\quad \mathit{pAddr} \in \mathit{LocalAddr} \end{cases}$$

Since the L1 caches are not kept coherent, reading a global memory address might return stale data from the cache as explained in section 2.3.2. However, CUDA does guarantee that global L1 cache lines are invalidated between "dependent grids" [9, Table 81], hence the second grid indeed fetches the correct value from either L2 or DRAM. Still, it is underspecified when and how global L1 cache lines are evicted, thus we use a silent rule to support this scenario. The (flush) rule selects one of the L1 caches and sets all of its global lines to the default state if none of the global cache lines are dirty. Local cache lines remain unchanged.

$$(\text{flush})\quad \langle \eta \triangleright \overrightarrow{\mathit{procMem}} \mid \overrightarrow{\mathit{op}} \rangle \rightarrow^{\text{M}} \langle \eta \triangleleft \overrightarrow{\mathit{procMem}}[\mathit{pid} \mapsto \overrightarrow{\mathit{procMem}}_{\mathit{pid}} \triangleleft \overrightarrow{\mathit{cl}_l} \circ \textstyle\bigcup_{0..|\overrightarrow{\mathit{cl}_g}|} \mathit{init}] \mid \overrightarrow{\mathit{op}} \rangle$$
$$\text{where}\quad \overrightarrow{\mathit{cl}} = \mathit{l1Cache}(\eta, \mathit{pid}) \land \overrightarrow{\mathit{cl}_g} = \{\mathit{cl} \in \overrightarrow{\mathit{cl}} \mid \mathit{cl}_{\mathit{pAddr}} \in \mathit{GlobalAddr}\} \land$$
$$\overrightarrow{\mathit{cl}_l} = \{\mathit{cl} \in \overrightarrow{\mathit{cl}} \mid \mathit{cl}_{\mathit{pAddr}} \in \mathit{LocalAddr}\} \qquad \text{if}\quad \forall \mathit{cl} \in \overrightarrow{\mathit{cl}_g}\ .\ \neg \mathit{cl}_{\mathit{dirty}}$$

A macro step is a sequence of rule applications of transition system $\rightarrow^{\text{M}}$. For some memory environment η and some in-flight memory operations $\overrightarrow{\mathit{op}}$, a macro step is the sequence of transitions $\langle \eta, \overrightarrow{\mathit{op}} \rangle \rightarrow^{\text{M},*} \langle \eta', \overrightarrow{\mathit{op}'} \rangle$.

4.7 Summary

We formalize the semantics of a declarative programming language that we in turn base our formal definition of memory operation semantics on to cope with the complexities of CUDA's memory model. The memory model is currently underspecified in most areas. For instance, in almost all cases the value returned to a thread that reads a location l is unknown if some other thread or even the same thread writes to l at some point in the program. Other examples of undefined behavior are the coalescing of memory operations, the interplay of caches and volatile operations, and how atomic operations on global memory affect the L1 caches. Furthermore, the memory programs defined in this chapter might yield execution too often and therefore allow memory programs to be deferred longer than possible on the hardware. Because of this, memory operations that are completely independent on real hardware might have negative effects on each another in our formalization of the memory model. For example, the way that memory program 4.1 defines the semantics of global writes with the `.wb` cache operation makes it possible that a read of global address a loads the value into the L1 cache after a store to address a has executed its first micro step. In that case, the old value is stored in the L1 cache and the new value might never be returned to any threads of the same streaming multiprocessor that request the value at address a. We do not know whether this issue might occur on real hardware. Even worse, our formalization does most likely not guarantee liveness, i.e. a program might never be able to execute its remaining micro steps. Future work would probably benefit the most from trying to fill in the missing gaps and giving stronger, more reliable guarantees for some of the most significant problem cases listed throughout this chapter.

With the release of CUDA 3.2 RC, Nvidia provided another detail about memory transaction sizes in [10, G.4.2]. In contrast to what we assumed before the release of version 3.2 RC, only memory accesses that are cached in the L1 cache are serviced with 128 byte transactions, whereas memory accesses that are only cached in L2 are serviced with 32 byte transactions to reduce over-fetch. While our formalization circumvents this issue by having each memory operation specify both its base address and the access size, there might be a problem if a 128 byte L1 access is split into four or less 32 byte L2 accesses. However, aside from possible performance gains, the semantical behavior should be the same whether there is one 128 byte access or up to four 32 byte accesses of the L2 cache.

Currently, the formalization of the memory environment does not support the reconfigurable L1 cache and shared memory sizes offered by Fermi-based GPUs. Since the same on-chip memory is used to implement both the L1 cache and shared memory, a streaming multiprocessor can be configured to use 16 KByte of shared memory and 48 KByte of L1 cache or vice versa. The host program decides which configuration to use. However, the configuration selected by the host program is only a suggestion to the CUDA runtime according to the reference manual [25, 4.7.2.3]. Consequently, the runtime is free to choose any of the two configurations even though the host program explicitly specified one of them. We assume this is at least in part due to the fact that the host program can choose different configurations for each kernel. If a streaming multiprocessor executes thread blocks of different kernels with different memory configurations, it has to choose one of them. We currently avoid this issue altogether by assuming fixed L1 cache and shared memory sizes.

5 Formal Semantics of CUDA's Model of Computation

The formal definition of CUDA's program semantics reflects CUDA's thread hierarchy introduced in section 2.3.1. The semantics of each layer in the hierarchy is built on top of the semantics of the lower layers. Thus, we first define the semantics of a single thread in section 5.3 and formalize the semantics of a single warp on top of that in chapter 5.4. The semantics of thread blocks presented in section 5.5 rests upon the warp semantics. Similarly, grids build on thread blocks, contexts on grids, and the semantics of the GPU as a whole depends on contexts as presented in sections 5.6, 5.7, and 5.8, respectively. The connection between the formalization of the program semantics and the memory environment is established at the device level. A function is handed down to the threads that allow them to instantaneously retrieve a value from a register. To issue memory requests, a thread that executes a memory instruction creates a memory action message with all required information that travels up the hierarchy. At the device level, the action is given to the memory environment that instantiates a corresponding memory program to service the request. At some levels of the hierarchy, actions are merged or amended with further data that is required to process the request but is unknown at the thread level; for instance, the issuing thread's processor index is amended at the thread block level. Actions are also used to handle warp level branching and thread block synchronization.

In the next section, we define an extended subset of PTX on which we base the formalization of the semantics. The subsequent sections each define the semantics of one level of the thread hierarchy. We also present a formalization of the warp branching algorithm and the mechanism used to handle thread block synchronization in this chapter.

5.1 Formalization of PTX

We do not define a formal semantics for full PTX as specified in [9]. Rather, we restrict the discussion to an extended subset of the PTX programming language on which we base the formalization of the semantics of CUDA's model of computation. Basically, we formalize the domain of program environments *ProgEnv* and informally describe how a real PTX program can be converted into its mathematical representation in the form of an element of *ProgEnv*. For the purpose of defining a formal semantics, this approach has several advantages over basing the semantics on PTX directly as the following sections explain.

We have to define a couple of basic domains that we need in the subsequent sections to formalize PTX. First, we introduce the following abstract domains to denote constants, variables, registers, labels, and function names in PTX programs.

$$c \in \mathit{Const} \qquad v \in \mathit{Var} \qquad r \in \mathit{Reg} \subseteq \mathit{Var}$$
$$f \in \mathit{FuncName} \qquad l \in \mathit{Label}$$

We support only a few of the special registers listed in [9, 9], namely the thread index `%tid`, the thread block size `%ntid`, the thread block index `%ctaid`, and the grid size `%nctaid`. The rest of the special registers exposed by the hardware are mostly used for debugging or diagnostic purposes and are thus less relevant to our formalization.

$$\begin{aligned} \mathit{Dim} &::= \texttt{x} \mid \texttt{y} \mid \texttt{z} \\ \mathit{SReg} &::= \texttt{\%tid.}\mathit{Dim} \mid \texttt{\%ntid.}\mathit{Dim} \mid \texttt{\%ctaid.}\mathit{Dim} \mid \texttt{\%nctaid.}\mathit{Dim} \end{aligned}$$

Program addresses are used by branch instructions to jump to another location in the program and can be stored in registers and other types of memory. Consequently, program addresses are just a special type of data and we define the domain of program addresses as a subset of data words.

$$pc \in \mathit{ProgAddr} \subseteq \mathit{DataWord}$$

When a thread accesses memory, it does so using virtual memory addresses of its context's virtual address space. Like program addresses, virtual memory addresses are just a special type of data that we define as follows.

$$\mathit{VirtMemAddr} \subseteq \mathit{DataWord}$$

The value stored by a variable is either a register address if the variable represents a register or a virtual memory address if the variable points to either shared, global, local, or constant memory.

$$\mathit{VarValue} = \mathit{RegAddr} \cup \mathit{VirtMemAddr}$$

In the next section, we take a look at the PTX instructions and features supported by the program environments and explain why we add or omit certain instructions or features. Subsequently, we present the formal definition of the domain of program environments in section 5.1.2. We informally explain how a PTX program is converted into a program environment in section 5.1.3.

5.1.1 PTX Instructions and Features Included in the Formalization

The formal semantics presented in this chapter only supports a subset of the full instruction set of PTX as specified in [9]. Since the memory environment does not support texture memory, there is obviously no point in supporting instructions that deal with textures or surfaces. Also, we leave out all instructions that are only useful in debugging or diagnostic scenarios.

We abstract from data types which are explicitly specified by most instructions of PTX. For example, we do not differentiate between arithmetic instructions operating on integer or floating point data. Only instructions that deal with memory access need to know the size of the accessed memory; nevertheless, memory operations do not care about the actual type of the data either.

We introduce the meta-operation `dop` that represents all of PTX' operations that take some number of operands, perform some operation, and return the resulting data word. We do not support all or at least some of the arithmetic, logic and shift instructions individually, because the basic structure of the rules dealing with those instructions is always the same. Thus, we restrict the discussion to the abstract data operation `dop` for reasons of brevity.

We do not support any of PTX' syntactic sugar, including vector variables, arrays, or the special variable declaration syntax `r<n>` which automatically introduces the variables `r1` ... `rn`. Furthermore, we replace commas between parameters of instructions with spaces to better cope with optional parameters.

In accordance with patent [26], we add the control flow instructions `brk`, `preBrk`, `preRet`, `ssy`, and `sync` to the instruction set. The reasons why we need those instructions explicitly in the program are discussed in section 5.4.1; usually the PTX compiler injects those instructions automatically into the program without the programmer's knowledge. The rest of the branching instructions — `ret`, `exit`, `bra`, and `call` — are regular PTX instructions. However, in our formalization the `call` instruction does not support any function call parameters or return values. We deal with those in a preprocessing step when loading a program onto the GPU. Further details about this topic can be found in section 5.1.3. Moreover, we do not allow the use of the `.uni` specifier that may optionally be used in the case that a branch instruction is guaranteed to be non-divergent for all threads of a warp. However, `.uni` is solely relevant for performance optimization. Since non-divergent control flow instructions without the `.uni` specifier are properly handled by our formalization of the warp level branching algorithm, we do not include this specifier for reasons of brevity.

The Fermi architecture introduced generic addressing which makes it possible to omit the state space when reading or writing global, local, or shared memory. If generic addressing is used, the GPU figures out the type of the memory referenced by the memory operation on its own. While this is an important feature to fully support pointers in CUDA-C, from the point of view of our semantics it only means that some value is subtracted from the specified memory address. Thus, we do not offer generic addressing and require that the state space always be specified on all load and store instructions.

In addition to the aforementioned instructions, we support `setp` to set the value of a predicate register based on the output of some comparison applied to some operands. Available comparison operators are:

$$comp \in Comparison ::= \texttt{.eq} \mid \texttt{.ne} \mid \texttt{.lt} \mid \texttt{.le} \mid \texttt{.gt} \mid \texttt{.ge}$$

Memory barriers force a thread to idle until all outstanding memory operations can be seen by all threads on the thread block, global, or system level.

$$mbl \in MemBarLevel ::= \texttt{.cta} \mid \texttt{.gl} \mid \texttt{.sys}$$

Warp level votes decide whether the predicates of all or any of the threads participating in the vote are true or whether all threads voted uniformly. Furthermore, the ballot mode constructs a data word where the bit corresponding to a thread's position in the warp is set to the value of the thread's predicate. A thread can use logic and shift instructions to use the result of a ballot in a meaningful way.

$$vm \in VoteMode ::= \texttt{.all} \mid \texttt{.any} \mid \texttt{.uni} \mid \texttt{.ballot}$$

When using the `bar` instruction, the barrier type specifies whether it is a thread block synchronization point with or without a reduction operation and whether the threads executing the instruction wait for the barrier to complete or may continue execution immediately, only marking their arrival at the barrier. Supported barrier reduction operations are conjunction, disjunction, and population count, i.e. the number of threads whose predicate is true.

$$
\begin{aligned}
\mathit{bt} \in \mathit{BarrierType} &::= \texttt{.sync} \mid \texttt{.arrive} \mid \texttt{.red}\\
\mathit{br} \in \mathit{BarrierRed} &::= \texttt{.and} \mid \texttt{.or} \mid \texttt{.popc}
\end{aligned}
$$

Cached memory operations specify the cache operation that should be used to service the request. Furthermore, the `.volatile` specifier causes a volatile read or write; we support volatile memory operations even though the semantics of volatile memory requests remain undefined for the reasons mentioned in chapter 4.

$$
\begin{aligned}
\mathit{cop} \in \mathit{CacheOp} &::= \texttt{.ca} \mid \texttt{.cg} \mid \texttt{.cs} \mid \texttt{.lu} \mid \texttt{.cv} \mid \texttt{.wb} \mid \texttt{.wt}\\
\mathit{volatile} \in \mathit{Volatility} &::= \texttt{.volatile}
\end{aligned}
$$

Atomic memory operations read the value at the specified address, perform some reduction operation and write the computed value back to the same location. Supported reduction operations include the following:

$$
\mathit{aop} \in \mathit{AtomicOp} ::= \texttt{.and} \mid \texttt{.or} \mid \texttt{.cas} \mid \texttt{.exch} \mid \texttt{.add} \mid \texttt{.min} \mid \texttt{.max}
$$

The grammar of the supported instruction set is listed below. Variable declarations, function declarations, and function pointer declarations are not part of the instruction set, as those are dealt with in a preprocessing step outlined in section 5.1.3. A guard can be defined for each instruction; a thread only executes an instruction if the guard evaluates to true. In our formalization, guards are handled separately.

$$
\begin{aligned}
\mathit{Instr} ::=\ & \texttt{dop}\ \mathit{Reg}\ \mathit{Expr}^{*}\\
\mid\ & \texttt{setp}\ \mathit{Comparison}\ \mathit{Reg}\ \mathit{Expr}\ \mathit{Expr}\\
\mid\ & \texttt{mov}\ \mathit{Reg}\ \mathit{Expr}\\
\mid\ & \texttt{ld}\ \mathit{Volatility}_{\varepsilon}\ \mathit{StateSpace}\ \mathit{CacheOp}_{\varepsilon}\ \mathit{MemOpSize}\ \mathit{Reg}\ \mathit{Expr}\\
\mid\ & \texttt{st}\ \mathit{Volatility}_{\varepsilon}\ \mathit{StateSpace}\ \mathit{CacheOp}_{\varepsilon}\ \mathit{MemOpSize}\ \mathit{Expr}\ \mathit{Reg}\\
\mid\ & \texttt{sync} \mid \texttt{brk} \mid \texttt{ret} \mid \texttt{exit}\\
\mid\ & \texttt{bra}\ \mathit{Expr} \mid \texttt{call}\ \mathit{Expr}\\
\mid\ & \texttt{preBrk}\ \mathit{Label} \mid \texttt{preRet}\ \mathit{Label} \mid \texttt{ssy}\ \mathit{Label}\\
\mid\ & \texttt{membar}\ \mathit{MemBarLevel}\\
\mid\ & \texttt{atom}\ \mathit{StateSpace}\ \mathit{AtomicOp}\ \mathit{MemOpSize}\ \mathit{Reg}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr}_{\varepsilon}\\
\mid\ & \texttt{vote}\ \mathit{VoteMode}\ \mathit{Reg}\ \mathit{Expr}\\
\mid\ & \texttt{bar.sync}\ \mathit{Expr}\ \mathit{Expr}_{\varepsilon}\\
\mid\ & \texttt{bar.arrive}\ \mathit{Expr}\ \mathit{Expr}\\
\mid\ & \texttt{bar.red}\ \mathit{BarrierRed}_{\varepsilon}\ \mathit{Reg}\ \mathit{Expr}\ \mathit{Expr}_{\varepsilon}\ \mathit{Expr}
\end{aligned}
$$

5.1.2 Definition of Program Environments

Program environments comprise several other environments which we define first. These environments are functions that return $\perp$ if they are passed undefined or unused input. $\perp$

serves as a sanity check because it causes the semantics to get stuck whenever it tries to do something undefined.

The instruction environment maps a program address to its corresponding instruction. If a program ever branches to an undefined program address, the program's behavior is generally unpredictable: it might crash instantly, it might execute instructions of some other program, or it might try to execute data. However, the way we formalize PTX guarantees that labels always point to valid program addresses. Consequently, only indirect branches can lead to undefined behavior, but that can also happen to real-world PTX programs.

$$\mathit{InstrEnv} = \mathit{ProgAddr} \rightarrow \mathit{Instr}_{\perp}$$

The guard environment returns an expression that represents the guard of the instruction at the specified program address. Expressions are defined in section 5.2; guards that evaluate to false cause a thread to skip the execution of the instruction.

$$\mathit{GuardEnv} = \mathit{ProgAddr} \rightarrow \mathit{Expr}_{\perp}$$

The label environment returns the program address the given label is an alias for. As mentioned above, the program address of a label is always well-defined, i.e. a valid program address.

$$\mathit{LabelEnv} = \mathit{Label} \rightarrow \mathit{ProgAddr}_{\perp}$$

The function environment returns the address of the first instruction of a function denoted by a certain function name.

$$\mathit{FuncEnv} = \mathit{FuncName} \rightarrow \mathit{ProgAddr}_{\perp}$$

Furthermore, we define the variable environment that maps a variable name to its value regardless of the current call stack location. Just like programs running on x86 CPUs, PTX programs have a call stack that identifies the local variables that should be accessed when a function reads or writes non-global data. For that, we define the domain of call stacks that keeps a stack of variable environments and only looks at the topmost of those when a function accesses locally defined variables. However, the call stack does not keep track of return addresses. Once a function returns, it is the responsibility of the warp level branching algorithm to return to the original function call site. This is in stark contrast to the way many x86 programming languages handle their call stack.

$$\begin{aligned} \nu &\in \mathit{VarEnv} = \mathit{Var} \rightarrow \mathit{VarValue}_{\perp} \\ \kappa &\in \mathit{CallStack} = \mathit{VarEnv}^{*} \end{aligned}$$

The local environment returns the variable environment of a function denoted by the program address of its first instruction. According to [9, 5.1.5], only variables pointing to local memory defined locally in a function are allocated on the stack. Hence, the variable environment returned for a specific function only returns valid data for locally declared local memory variables, but not for locally defined registers or variables of other state spaces. When a thread calls a function, the semantics uses the local environment to retrieve the variable environment that contains the function's local variables. This environment is pushed onto the call stack and popped off the stack once the function has completed execution.

$$\mathit{LocalEnv} = \mathit{ProgAddr} \rightarrow \mathit{VarEnv}_{\perp}$$

We introduce the abstract domain of program indices to uniquely identify a program environment.

$$progid \in ProgIdx$$

A program environment incorporates all of the aforementioned environments. It is the mathematical representation of the extended subset of the PTX programming language that we define the formal semantics for. The program environment's variable environment handles all globally defined variables and all variables that are defined locally in a function but do not point to a local memory location. As mentioned above, these variables are not stored on the stack. It is not possible to store the call stack in the program environment, because its state might differ for each thread during the execution of the program. Hence, the threads manage their call stacks independently.

$$\phi \in ProgEnv = InstrEnv \times GuardEnv \times LabelEnv \times FuncEnv \times LocalEnv \times VarEnv \times ProgIdx$$

There can be several entry points defined for a single PTX program. After the host has loaded a program onto the GPU, it can launch any number of kernels using any of the entry points defined in the program with any grid and thread block dimensions. The grid and block dimensions are limited by the capabilities of the underlying GPU. However, at least one thread must be launched to execute a program.

$$\begin{aligned} GridSize &= \textstyle\prod_{dim \in Dim} \{1, \dots, MaxGridSize_{dim}\} \\ BlockSize &= \textstyle\prod_{dim \in Dim} \{1, \dots, MaxBlockSize_{dim}\} \end{aligned}$$

The amount of registers used per thread and the amount of shared memory used per thread block affect how many warps and thread blocks can be allocated concurrently on a single streaming multiprocessor. The lower the amount of allocated warps, the more likely it is that instruction latencies cannot be completely hidden by executing other non-blocked warps.

$$RegsPerThread \subseteq \mathbb{N} \qquad SharedMemPerBlock \subseteq \mathbb{N}$$

We introduce the domain of program configurations that store which entry point to call, the number of thread blocks to launch, the number of threads and the amount of shared memory per thread block as well as the amount of registers required by each thread. All threads start executing the program at the first instruction of the kernel function specified by their program configuration. The Giga Thread scheduler uses the program configuration to issue thread blocks to streaming multiprocessors with available execution capacity.

$$\xi \in ProgConf = FuncName \times GridSize \times BlockSize \times RegsPerThread \times SharedMemPerBlock$$

The following two axioms hold for all program configurations $\xi \triangleright gridSize, blockSize$; the grid and the block dimensions of a kernel must not exceed the maximum allowed values:

$$\begin{aligned} gridSize_x \cdot gridSize_y \cdot gridSize_z &\leq MaxBlocksPerGrid \\ blockSize_x \cdot blockSize_y \cdot blockSize_z &\leq MaxThreadsPerBlock \end{aligned}$$

5.1.3 Representation of PTX Programs as Program Environments

Obviously, it is only possible to construct a program environment for PTX programs that solely use the instructions found in our formalized instruction set. In this section, we informally outline how the process of transforming a PTX program into a program environment works. At the same time, we discuss some PTX complexities that we get rid of during the process.

The process is split into two major steps. The first one applies some code transformations to the PTX program in order to facilitate the second step, where the different environments that form up the program environment are created. Each of the two step consists of several sub steps.

Code Transformations

The purpose of the code transformations is to normalize the structure of a PTX program. The construction of the various environments is facilitated because the normalization eliminates special cases or avoids certain PTX complexities altogether. We illustrate the code transformations with the help of program 5.1 which is a valid program that the PTX compiler accepts even though it does nothing meaningful. Applying the code transformations to program 5.1 results in the transformed version of the program shown in listing 5.2; the transformed program is no longer a valid PTX program. We use the domain *Prog* to designate transformed PTX programs.

```
.global .u32 var = 10 / 2 + 3;

.func function(.param .f64 p)
{
  .reg .f64 a;
  ld .param .f64 a, [p];
}

.entry transformationKernel()
{
  .reg .f64 var;
  .reg .pred p;

  @p call function, (var);
}
```

Listing 5.1: The PTX program used to illustrate the effects of the code transformations

Step 1. The first step is to check whether the given PTX program is actually a valid program, i.e. whether it is syntactically correct and whether all referenced variables, labels, and functions are defined. Furthermore, type checks are performed and instruction parameters are validated as far as possible. This can be done by passing the given PTX program to Nvidia's PTX compiler. If the compiler does not produce any errors, the program is valid and we can continue to step 2.

Step 2. PTX programs support complex arithmetic operations on constant values. This step replaces values that can be evaluated at compile-time by their computed result. The

```
.global var0 = 8;
.reg %stackBase;

.func function(.local p0)  // p0 has offset 0
{
  .reg a, addr;
  add addr, %stackBase, [p0];
  ld .local 64 a, [addr];
  ret;
}

.entry transformationKernel()
{
  .reg addr, var1, p1;
  .local l_var1; // offset 0

  @p1 add addr, %stackBase, [l_var1];
  @p1 st .local 64 [addr], var1;
  @p1 add %stackBase, %stackBase, 0; // The kernel does not use any local variables, thus the stack base address does not change
  @p1 preRet;
  @p1 call function;
  @p1 sub %stackBase, %stackBase, 0;
  exit;
}
```

Listing 5.2: Program 5.1 after the application of the code transformations

result is an element of the domain *Const*.

Step 3. According to the PTX specification, local memory variables declared within a function body are allocated on the stack [9, 5.1.5]. Semantically, we represent the stack with the *CallStack* domain defined in section 5.1.2 which manages a stack of local variable environments. To keep things simple, the semantics does not attempt any kind of automatic register preservation when a `call` or `ret` instruction is encountered. Instead, we rely on a code transformation to take care of register preservation as well as function parameters and return values. After this step, implicit register preservation is made explicit by a series of regular load/store instructions before and after a function call. Function parameters and return values are handled by the local variable sharing mechanism outlined below. Consequently, the rules of the formal semantics do not have to deal with those two issues; the only thing done by the rules is to push and pop variable environments onto and off the threads' call stacks.

When a function calls another function, some registers might have to be saved. The called function might reuse registers of the caller function, thus the original state of the registers might have to be restored if the caller needs the original value again. In this step, all registers whose state must be preserved are spilled to local memory before the function call and the values in local memory are copied back to the registers once the function call completes.

Local memory variables defined in the function's body are allocated on the stack. When we construct the variable environments in later steps, we assign fixed addresses to all variables found in the program, including stack-allocated variables to avoid special casing those. To ensure stack-semantics for stack-allocated variables, we introduce a new register in the

```
// Called function; offset of r is 0, offset of p is 8 byte
.func (.param .u64 r) function(.param .u32 p) { ... }

// Before the code transformation
.entry kernelEntryPoint()
{
  .reg .pred p;
  .reg .u64 rr;
  .local .u128 x; // Unused, introduced for illustration purposes
  .param .u64 r;
  .param .u32 a;

  st .param .u32 [a], 9;
  @p call (r), function, (a);
  ld .param .u64 rr, [r];
}

// After the code transformation
.reg .u64 %stackBase; // Added by code transformation
.entry kernelEntryPoint()
{
  .reg .pred p;
  .reg .u64 rr;
  .local .u128 x; // Offset 0 byte
  .local .u64 r; // Offset 16 byte
  .local .u32 a; // Offset 24 byte

  add .u64 rr, %stackBase, [a]; // Computes absolute address of a
  st .local .u32 [rr], 9;
  @p add .u64 %stackBase, %stackBase, 16; // Now r lies at offset 0
  @p call function; // Parameters are removed
  @p sub .u64 %stackbase, %stackBase, 16; // Now x lies at offset 0 again
  add .u64 rr, %stackBase, [r]; // Computes absolute address of r
  ld .local .u64 rr, [rr];
}
```

Listing 5.3: Illustration of the stack construction and deconstruction code transformation

program named `%stackBase`; if there is already another register with the same name, the other register is renamed. `%stackBase` stores the address of the current stack frame. The address is increased before a `call` instruction and reset to the original value after a `call` instruction. Program 5.3 shows the effects of this step for a simple function call. Figure 5.1 shows the stack layout before and after the function call of program 5.3. The initial value of `%stackBase` depends on the amount of globally declared local memory variables in the program and their sizes.

We assume that we later assign relative local memory addresses in the following way to locally-scoped local memory variables: The address of the function's return parameter, if there is one, has offset 0. The offset is relative to the current value of `%stackBase`. The function's parameters are located at successive locations, taking into account the size of the parameters. So if a function returns a 64 bit value, the offset of the first parameter is 8 byte. When a function calls another function, it can either use registers to pass input parameters and return values, or the parameter state space. In the latter case, this step ensures that

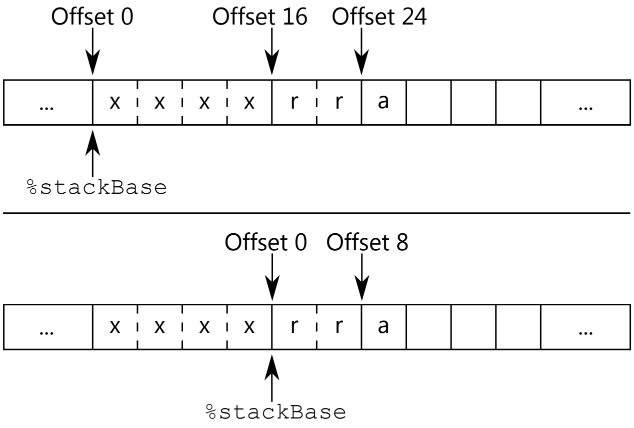


Figure 5.1: Stack layout before (top) and after (bottom) the function call of program 5.3

locally-scoped local memory variables are created for the return and input values such that the layout of those variables matches the rules defined above. By incrementing `%stackBase` to the address of the return parameter of the called function, the called function can again access its return parameter with offset 0 and all of its parameters with their relative offsets. Of course, the variables used to store the return and input values of the called function must have higher offsets than all other local variables of the caller function.

For guarded function calls, the same guard is used to enable or disable the execution of the stack management instructions added in this step. That way, if a thread does not participate in the function call, it does not unnecessarily perform all those memory operations. However, that is just an optimization.

Parameters of kernel functions, i.e. functions defined with the `.entry` specifier, are special in that the parameters lie in constant memory instead of local memory. Kernel functions cannot call themselves recursively and they also cannot be called by other PTX functions. The CUDA runtime ensures that the kernel parameters point to the appropriate addresses in constant memory where the input values specified by the host program are stored.

This step also replaces the `.param` state space of all parameters with either `.local`, `.const`, or `.reg` depending on the actual location of the parameter. Load and store instructions operating on parameters are consequently replaced with regular register accesses or with accesses of local or constant memory. Except for kernel parameters that are guaranteed to lie in constant memory, the PTX compiler tries to use the most optimal state space for function parameters. Typically, this means passing parameters in registers whenever possible. We assume the compiler uses the correct state spaces — i.e. it does not pass parameters via registers if the function is called recursively — but are otherwise not interested at all in the optimizations performed by the compiler.

Step 4. Data type specifiers on memory operation instructions are replaced with the corresponding memory operation size. All data type specifiers are removed from all other instructions. As mentioned in chapter 4, we assume that all registers are sufficiently large to hold all values that are passed to and returned from any of the instructions. Memory operations, on the other hand, need to know the size of the accessed memory.

Step 5. In this step, it is ensured that `ret` is the last instruction executed on any path in any function. The return statement is implicit in PTX if not present, so enforcing its presence does not affect the semantics of the program but allows us to eliminate a special case. Afterwards, it is ensured that the last instruction executed by any thread is `exit`. Basically this means that all `ret` statements are replaced with `exit` statements in functions marked with the `.entry` specifier.

Step 6. Similar to what the Nvidia's PTX compiler does, we inject the additional control flow instructions into the program that are required to handle divergent control flow. We do not know what algorithm the compiler uses to insert the instructions, but we have to assume they are inserted correctly in accordance with the branching algorithm specified in [26]. In contrast to what the compiler does, we do not use sync flags on instructions but rather a concrete `sync` instruction for reasons outlined in section 5.4.1. Consequently, when the compiler adds a sync flag to an instruction, a `sync` instruction should be added directly before the flagged instruction and the flag should be removed.

Step 7. Like C, PTX uses curly braces to create scopes of declaration [9, Table 97]. However, the precise semantics of the scoping rules are undocumented. Program 5.4 demonstrates some of the rules and their oddities. The register declared in line 3 is still visible in line 9. In line 12, a new register with name `r` is declared that hides the outer declaration. Consequently, the `add` instruction in line 15 operates on the newly declared `r`, curiously even though the thread definitely skipped line 12 because of the unconditional jump to the label `SKIP`. Furthermore, the `bra` statement in line 16 causes the control flow to jump to line 20 because inner label declarations hide outer labels of the same name. The branch instruction in line 17 shows that it is possible to jump out of a block, however, it is not possible to jump into a block; in particular, it is impossible to jump into a function body. We avoid the complexities

```
 1 .entry kernelEntryPoint()
 2 {
 3   .reg .s32 r;
 4   .reg .pred p;
 5
 6   BLOCK:
 7   {
 8     {
 9       add .s32 r, r, r;
10
11       bra .uni SKIP;
12       .reg .u64 r;
13
14       SKIP:
15         add .u64 r, r, r;
16         @!p bra BLOCK;
17         @p bra CONTINUE;
18     }
19
20     BLOCK:
21   }
22
23   CONTINUE:
24 }
```

Listing 5.4: An incomplete PTX program that illustrates the scoping rules of PTX

of the undocumented scoping rules by renaming hidden variables and labels. In this step, it is ensured that variable and label names are unique, otherwise they are renamed in accordance with the scoping rules. Hence, this step does not change the behavior of the program, but the transformed program no longer has variables or labels that hide an outer definition. This also applies to function pointer hiding. Interestingly, function pointer definitions behave like variable names even though a label is used to name them.

Program Environment Generation

In this step, we define the function *load* that constructs a program environment ϕ for a PTX program generated by the above code transformations. The function is passed the memory environment and a transformed PTX program as well as the index used to identify the generated program environment. It returns an updated memory environment and the constructed program environment, unless there are any errors.

$$load : MemEnv \times Prog \times ProgIdx \rightarrow (MemEnv \times ProgEnv)_{\perp}$$

Step 1. The instruction environment is constructed by assigning a program address to each instruction. The semicolon at the end of each instruction as well as the the commas between instruction parameters are removed or replaced by spaces, respectively, as those are not part of the grammar of instructions. Additionally, the address operator `[]` is removed from all expressions, i.e. an expression `[e]` is replaced with `e`.

Step 2. The optional guard of each instruction is added to the guard environment using its instruction's program address. If an instruction has no guard defined, true is added to the guard environment.

Step 3. A program address is assigned to each label in order to construct the label environment. The label's address is the address of its subsequent instruction. The code transformations ensure that there is at least one instruction following each label. If a label at the end of a function body did not have any instructions following it in the original PTX program, the code transformations ensure that the label is now followed by either a `ret` or an `exit`. Additionally, the function environment is created by assigning the program address of the first instruction in a function's body to the function's name.

Step 4. The global variable environment and the local environment are constructed. For the former, a virtual memory address or register address is assigned to each variable or register declared by the program. This does not include locally-scoped local memory variables in function bodies as those are used to build up the local environment. We do not assign absolute virtual memory addresses to stack-allocated variables, but offsets relative to the current stack frame address as explained above. This step must guarantee the correct parameter layout: The return parameter lies at offset 0, the input parameters at successive offsets. The local variables used for function calls must have higher offsets than all other local variables declared in the function that are used only internally by the function.

Step 5. Variables living in global or constant memory can be initialized at program load-time. We do this by directly modifying the memory environment, because memory initialization has to happen synchronously. Once *load* returns, threads can immediately start to execute the program and the updated memory environment guarantees that they read the initial values of load-time initialized variables in any case.

5.2 Expression Semantics

Some PTX instructions support parameters whose values are determined by evaluating an expression. Not all expressions are valid in all cases according to the specification of PTX [9], so we have to rely on the preprocessing mechanism outlined in section 5.1.3 to only generate valid program environments. Expressions support referencing a variable, a constant, a special register, a label, or a function as well as negation of a variable's value and the addition of a constant to a variable. We leave out the address operator `[]` because in our formalization, variables generally refer to an address of their storage location.

$$e \in \mathit{Expr} ::= \ \mathit{Var} \ | \ !\,\mathit{Var} \ | \ \mathit{Const} \ | \ \mathit{Var} + \mathit{Const} \ | \ \mathit{Label} \ | \ \mathit{SReg} \ | \ \mathit{FuncName}$$

The function $\mathcal{E}[\![-]\!]$ defines the semantics of expressions. It depends on several other functions that we specify before we give the formal definition of $\mathcal{E}[\![-]\!]$ itself. First of all, the abstract function $\mathcal{C}[\![-]\!]$ evaluates a constant expression and returns the corresponding data word.

$$\mathcal{C}[\![-]\!] : \mathit{Const} \rightarrow \mathit{DataWord}$$

Secondly, function $\mathcal{V}[\![-]\!]$ retrieves the value of a variable. If the variable is a register, $\mathcal{V}[\![-]\!]$ returns the address of the register, i.e. the location of the register in the register file of the calling thread's streaming multiprocessor. All other variables represent a virtual memory address of some type of CUDA memory. Furthermore, $\mathcal{V}[\![-]\!]$ takes the current call stack into account. If a variable is defined locally in the current function and it points to an address in local memory, the address stored in the call stack's variable environment is returned. Otherwise, the value stored in the program's global variable environment is retrieved. Since we assume variable names to be unique within a program, the case distinction used in the definition of $\mathcal{V}[\![-]\!]$ is well-defined.

$$\mathcal{V}[\![-]\!] : \mathit{Var} \rightarrow (\mathit{ProgEnv} \times \mathit{CallStack}) \rightarrow \mathit{VarValue}_\bot$$

$$\mathcal{V}[\![v]\!]\phi(\kappa \triangleright \nu :: \vec{\nu}) = \begin{cases} \nu(v) & \text{if} \quad \phi_\nu(v) = \bot \\ \phi_\nu(v) & \text{otherwise} \end{cases}$$

As seen in chapter 4, memory operations are serviced asynchronously. However, most instructions encountered by the threads depend on registers as either input operands or output destinations. To avoid further complicating the semantics, we do not issue asynchronous requests to the memory environment in order to read the value of a register. Instead, we define a function that allows a thread to synchronously look up the values of its registers — provided the registers in question are not currently blocked because of a pending memory operation accessing them. The register look up function is specified at the device level of the semantics and is passed all the way down to the semantic rules of threads and the evaluation function of expressions. Whereas memory actions are amended with certain information not known at the thread level as they travel up the hierarchy, the register look up function is also given more detailed information on its way down the hierarchy using function currying. We use the meta-variable ϱ to denote the curried register look up function at each level of the hierarchy and give a concrete implementation at the device level in section 5.8. For expression evaluation, we need a register look up function that returns the value of special registers or a regular registers accessed by an expression.

$$\varrho \in \mathit{ExprRegLookup} = \mathit{RegAddr} \cup \mathit{SReg} \rightarrow \mathit{DataWord}_\bot$$

The expression evaluation function $\mathcal{E}[\![-]\!]$ uses the program environment, an expression register look up function, and a call stack to compute the value of an expression. It returns an expression value which is obviously equal to the domain of data words; we define the domain of expression values anyway to highlight the types of values that an expression might evaluate to.

$$\mathit{ExprValue} = \mathit{VirtMemAddr} \cup \mathit{DataWord} \cup \mathit{ProgAddr} = \mathit{DataWord}$$

If a variable v is evaluated by $\mathcal{E}[\![-]\!]$, v is either a register or another type of variable. If it is a register, that is $v \in \mathit{Reg} \subseteq \mathit{Var}$, the evaluation of v with $\mathcal{V}[\![-]\!]$ returns a register address as explained above. This register address is passed to the expression register look up function which returns the data word stored by the register in the case that the register is not blocked. If it is blocked, ϱ returns $\perp$, thus $\mathcal{E}[\![-]\!]$ returns $\perp$ and the program is stuck. Therefore, the formalization of the warp scheduler must guarantee that a warp is not scheduled if one of its threads would access a blocked register. If v is not a register, the virtual memory address that v represents is returned and since v is a meta-variable for the domain of variables, v cannot possibly refer to a special register. Hence, $\mathcal{E}[\![-]\!]$ is well-defined.

$$\mathcal{E}[\![-]\!] : \mathit{Expr} \to (\mathit{ProgEnv} \times \mathit{ExprRegLookup} \times \mathit{CallStack}) \to \mathit{ExprValue}_\perp$$

$$\begin{aligned}
\mathcal{E}[\![c]\!]\phi\varrho\kappa &= \mathcal{C}[\![c]\!] \\
\mathcal{E}[\![v]\!]\phi\varrho\kappa &= \begin{cases} \varrho(\mathcal{V}[\![v]\!]\phi\kappa) & \text{if} \quad v \in \mathit{Reg} \\ \mathcal{V}[\![v]\!]\phi\kappa & \text{otherwise} \end{cases} \\
\mathcal{E}[\![!v]\!]\phi\varrho\kappa &= \begin{cases} 0 & \text{if} \quad \mathcal{E}[\![v]\!]\phi\varrho\kappa \neq 0 \\ 1 & \text{otherwise} \end{cases} \\
\mathcal{E}[\![v + c]\!]\phi\varrho\kappa &= \mathcal{E}[\![v]\!]\phi\varrho\kappa + \mathcal{E}[\![c]\!]\phi\varrho\kappa \\
\mathcal{E}[\![l]\!]\phi\varrho\kappa &= \phi_{\mathit{labelEnv}}(l) \\
\mathcal{E}[\![f]\!]\phi\varrho\kappa &= \phi_{\mathit{funcEnv}}(f) \\
\mathcal{E}[\![\mathit{sreg}]\!]\phi\varrho\kappa &= \varrho(\mathit{sreg}) \\
\mathcal{E}[\![e]\!]\phi\varrho\kappa &= \perp \qquad \text{otherwise}
\end{aligned}$$

For reasons of convenience, we define a lifted version of the expression evaluation function that returns ε if it is given an empty expression.

$$\mathcal{E}_\varepsilon[\![-]\!] : \mathit{Expr}_\varepsilon \to (\mathit{ProgEnv} \times \mathit{ExprRegLookup} \times \mathit{CallStack}) \to \mathit{ExprValue}_{\varepsilon,\perp}$$

$$\mathcal{E}_\varepsilon[\![e_\varepsilon]\!]\phi\varrho\kappa = \begin{cases} \varepsilon & \text{if} \quad e_\varepsilon = \varepsilon \\ \mathcal{E}[\![e_\varepsilon]\!]\phi\varrho\kappa & \text{otherwise} \end{cases}$$

Similarly, we define a convenience function that converts the evaluated value of an expression into a Boolean value. $\mathcal{B}[\![-]\!]$ uses the C semantics of Boolean expressions to decide whether true or false should be returned: If the expression evaluates to something other than zero, true is returned. Otherwise, false is returned.

$$\mathcal{B}[\![-]\!] : \mathit{Expr} \to (\mathit{ProgEnv} \times \mathit{ExprRegLookup} \times \mathit{CallStack}) \to \mathbb{B}_\perp$$

$$\mathcal{B}[\![e]\!]\phi\varrho\kappa = \begin{cases} \text{true} & \text{if} \quad \mathcal{E}[\![e]\!]\phi\varrho\kappa \neq 0 \\ \text{false} & \text{otherwise} \end{cases}$$

5.3 Thread Semantics

In this section, we define the semantics of a single thread using two transition systems. The first one, presented in section 5.3.2, defines the semantics of instructions as they are executed by a thread. The second one, found in section 5.3.3, is based on the first one and checks whether the thread is actually allowed to execute the instruction; the thread could be disabled by the warp or the instruction's guard could evaluate to false. A thread executes the instruction pointed to by its warp's program counter. Additionally, the warp might decide that the thread is not allowed to execute the current instruction in the case that the warp is executing a path the thread has not taken. Other than that, the warp has no influence on the instruction semantics at the thread level.

CUDA assigns a thread index to each thread that is unique within the thread's thread block. We use three-dimensional thread indices; [3, 2.2] shows how a three-dimensional index can be converted into a one-dimensional one.

$$tid \in ThreadIdx = \prod_{dim \in Dim} \{0, \ldots, MaxBlockSize_{dim} - 1\}$$

At the warp level, threads are often identified by their warp lane. The warp lane specifies the position of a thread within a warp. For instance, the ballot mode of the `vote` instruction stores the Boolean value of a thread's predicate inside a data word, where the position of the bit inside the data word is determined by the thread's warp lane.

$$wl \in WarpLane = \{0, \ldots, WarpSize - 1\}$$

When a thread issues a memory request to the memory environment, the requested address is obtained by evaluating an expression possibly containing variables. However, addresses are assigned globally to the variables of a program, meaning that a variable referred to in an instruction actually references the same memory location for all threads executing the program. To solve this issue, we define distinct memory offsets for each thread. For registers, a unique register offset is assigned to each thread running on the same streaming multiprocessor. The address assigned to a register at program load-time is then added to each thread's individual offset to compute a unique register address. Similarly, we assign a local memory offset to each thread. In contrast to the register offset, the local memory offset must be unique for all threads running on the GPU. This is because all threads access the same local memory in DRAM, whereas there is a unique register file for each streaming multiprocessor. We assume the offsets are given in the physical memory space, however, it could just as well be the virtual memory space. Obviously, threads also need an offset for shared memory accesses. However, shared memory information is stored at the thread block level only, so the thread has no knowledge of its thread block's shared memory offset. Therefore, memory requests are augmented with the thread block's shared memory offset at the thread block level. Global and constant memory accesses, on the other hand, do not require the use of offsets because all addresses in those types of memory are accessible by all threads belonging to the same context. Different contexts use distinct virtual memory spaces to avoid sharing global and constant physical memory addresses.

$$\begin{aligned} ro &\in RegOffset \subseteq PhysMemAddr \\ lo &\in LocalMemOffset \subseteq PhysMemAddr \end{aligned}$$

The domain of threads consists of the thread's index and warp lane as well as the offsets for register and local memory accesses. Additionally, a thread keeps track of its function call stack and the level of the memory barrier its waits for. A thread is not allowed to execute any further instruction if it executed a memory barrier that is not yet completed. The stored memory barrier value is reset to ε as soon as all of the thread's pending memory operations surpass the specified barrier level as explained in section 5.8.1. Only if the memory barrier equals ε can the thread and consequently its warp be scheduled again.

$$\theta \in \mathit{Thread} = \mathit{CallStack} \times \mathit{ThreadIdx} \times \mathit{RegOffset} \times \mathit{LocalMemOffset} \times \mathit{MemBarLevel}_\varepsilon \times \mathit{WarpLane}$$

In each dimension, the thread index ranges between 0 and the block size for all threads $\theta \triangleright \mathit{tid}$ and all program configurations $\xi \triangleright \mathit{blockSize}$.

$$0 \leq \mathit{tid}_x < \mathit{blockSize}_x \land 0 \leq \mathit{tid}_y < \mathit{blockSize}_y \land 0 \leq \mathit{tid}_z < \mathit{blockSize}_z$$

5.3.1 Thread Actions

Threads emit thread actions that travel up the hierarchy to allow control flow instructions to be handled at the warp level, barrier instructions to be handled at the thread block level, and memory operations to be handled by the memory environment at the device level. Consequently, actions travel up to the point where they are handled and disappear afterwards; for instance, control flow actions are no longer relevant at the thread block or even grid level, thus the warp semantics do not pass them up the hierarchy after dealing with them. In some cases, a higher level might amend certain information to an action. For example, the shared memory offset is added to memory operation actions at the thread block level as mentioned above.

We introduce the domain of thread actions that comprises control flow actions, memory actions, and communication actions that a thread may send up the hierarchy.

$$\mathit{tact} \in \mathit{Act}_\Theta ::= \mathit{FlowAct} \;\mid\; \mathit{MemAct}_\Theta \;\mid\; \mathit{CommAct}_\Theta$$

Control flow actions inform the warp about the branching instruction that was executed and might also specify some additional information such as the target address of a jump or function call as well as the warp lane of the thread that generated the action.

$$\begin{aligned}\mathit{tact}_{\mathit{flow}} \in \mathit{FlowAct} ::= {} & \texttt{sync} \\ & \mid \texttt{break}\ \mathit{WarpLane} \mid \texttt{return}\ \mathit{WarpLane} \mid \texttt{exit}\ \mathit{WarpLane} \\ & \mid \texttt{ssy}\ \mathit{ProgAddr} \mid \texttt{preRet}\ \mathit{ProgAddr} \mid \texttt{preBrk}\ \mathit{ProgAddr} \\ & \mid \texttt{bra}\ \mathit{WarpLane}\ \mathit{ProgAddr} \\ & \mid \texttt{call}\ \mathit{WarpLane}\ \mathit{ProgAddr}\end{aligned}$$

Load, store, and atomic instructions generate memory actions that contain a copy of the information specified by the instruction. Using this information, the memory environment services the requests at the device level. Additionally, an instruction might not access memory but only write a new value to a register. We define a grammar for memory requests that we use to construct memory actions.

$$\mathit{LdReq} ::= \mathit{RegAddr} \leftarrow \mathit{VirtMemAddr}\ \mathit{StateSpace}\ \mathit{MemOpSize}\ \mathit{Volatility}_\varepsilon\ \mathit{CacheOp}_\varepsilon$$

$$StReq ::= VirtMemAddr\ StateSpace\ MemOpSize\ ExprValue\ Volatility_\varepsilon\ CacheOp_\varepsilon$$

$$AtomicReq ::= RegAddr \leftarrow VirtMemAddr\ StateSpace\ AtomicOp\ MemOpSize\ ExprValue\ ExprValue_\varepsilon$$

$$RegReq ::= RegAddr \leftarrow ExprValue$$

$$Req ::= LdReq \mid StReq \mid AtomicReq \mid RegReq$$

Based on the data provided by a memory request, a memory action adds the thread's index as well as the thread's register and local memory offsets to the action. When the memory environment processes the action, it needs this data to calculate physical addresses and to coalesce requests of threads of the same warp into fewer memory transaction for performance reasons. Additionally, all registers that are somehow affected by the operation, i.e. registers that are source or destination operands of the instruction, are stored in the action. The memory environment blocks these registers for as long as the memory operation is processed. A thread cannot be scheduled if its next instruction requires access to a currently blocked register. We group all necessary information for memory actions in the domain *MemInf* that we can use for vote and barrier actions too. Even though votes and barriers do not generate memory operations at the thread level, some of them generate memory operations for the thread higher up the hierarchy at the warp or thread block level. There, all necessary memory operation data must be present, hence it must be stored in the original vote and barrier actions as shown later on.

$$MemInf ::= ThreadIdx\ RegOffset\ LocalMemOffset\ RegAddr^*$$

$$tact_{mem} \in MemAct_\Theta ::= Req\ MemInf$$

The abstract function *regs* retrieves all register addresses that are accessed by an instruction. Since register addresses are independent of the current call stack, the function can use the program environment's global variable environment to retrieve the addresses of all of the registers referenced by a specific instruction.

$$regs : Instr \times ProgEnv \to RegAddr^*$$

When a thread encounters a barrier instruction, it generates an action to inform the thread block that it has arrived at the barrier. Each barrier is uniquely identified by its barrier index; the number of available barrier indices depends on the underlying hardware. Additionally, the program might optionally specify a thread count, i.e. the number of threads that has to arrive at the barrier before the barrier completes. The thread count must be a multiple of the warp size, because barrier arrival is handled at the warp level. If only one thread of a warp encounters a `bar` instruction, it is as if all threads of the warp encountered it.

$$barid \in BarrierIdx = \{0, \dots, NumBarrierIndices - 1\}$$

$$tc \in ThreadCnt = \{n \in \mathbb{N} \mid n \bmod WarpSize = 0\}$$

Communication actions either inform the thread's warp about the values specified by a `vote` instruction or they are used by the thread's thread block to update the state of a synchronization barrier. Both vote and barrier actions include all the necessary information to issue a memory operation for the thread at the warp or thread block level.

$$p \in Predicate = \mathbb{B}$$

$$\begin{aligned} tact_{vote} &\in VoteAct ::= \texttt{vote}\ RegAddr\ Predicate\ WarpLane\ MemInf \\ tact_{bar} &\in BarrierAct_{\Theta} ::= \texttt{bar}\ BarrierIdx\ ThreadCnt_{\varepsilon}\ RegAddr_{\varepsilon}\ Predicate_{\varepsilon}\ MemInf_{\varepsilon} \\ tact_{comm} &\in CommAct_{\Theta} ::= VoteAct \mid BarrierAct_{\Theta} \end{aligned}$$

5.3.2 Thread Rules

Transition system $\rightarrow^{\Theta}$ defines the semantics of instructions at the thread level. In order to be able to evaluate expressions, the program environment and an expression register look up function are given to the rules of $\rightarrow^{\Theta}$. Typically, a thread either generates a control flow, memory, or communication action when it executes an instruction. There is only one case in which no action is generated.

$$\begin{aligned} Configurations &: \langle \theta \mid instr, \phi, \varrho \rangle \in Thread \times Instr \times ProgEnv \times ExprRegLookup, \\ &\quad\ \theta \in Thread \\ Transitions &: \langle \theta \mid instr, \phi, \varrho \rangle \xrightarrow[tact_{\varepsilon}]{}{}^{\Theta}\ \theta' \end{aligned}$$

The (dop) rule handles all operations that operate on data words. This includes addition, multiplication, bit shifting, bitwise and, calculation of the reciprocal square root, and many more. The rule is based on the abstract function *dop* that computes the resulting data word in accordance with the semantics of the corresponding operation specified in [9, 8.7.1, 8.7.2, and 8.7.4]. The expression value returned by the *dop* function is stored in a memory action that is passed up the hierarchy together with other necessary information to process the action at the device level. The expressions $e_1 \dots e_n$ are evaluated using the expression evaluation function $\mathcal{E}[\![-]\!]$. The address of the target register is retrieved by invoking the function $\mathcal{V}[\![-]\!]$.

$$\text{(dop)}\quad \langle \theta \triangleright \kappa, tid, ro, lo \mid instr \triangleright \texttt{dop}\ r\ e_1 \dots e_n, \phi, \varrho \rangle$$
$$\xrightarrow[\mathcal{V}[\![r]\!]\phi\kappa \leftarrow dop(\mathcal{E}[\![e_1]\!]\phi\varrho\kappa, \dots, \mathcal{E}[\![e_n]\!]\phi\varrho\kappa)\ tid\ ro\ lo\ regs(instr, \phi)]{}{}^{\Theta}\ \theta$$

A `bra` instruction has no meaning at the thread level; all branching related decisions are made exclusively at the warp level. However, the warp's branching algorithm needs to know the target address of the branch that might differ for each thread taking the branch. Together with the warp lane, the target program address is stored in a control flow action that is subsequently processed by the warp. The expression e is either a label, in which case $\mathcal{E}[\![-]\!]$ returns a valid program address because we only consider valid PTX programs, or a register. If the value stored in the register is not a valid program address, program behavior is undefined.

$$\text{(bra)}\quad \langle \theta \triangleright \kappa, wl \mid \texttt{bra}\ e, \phi, \varrho \rangle \xrightarrow[\texttt{bra}\ wl\ \mathcal{E}[\![e]\!]\phi\varrho\kappa]{}{}^{\Theta}\ \theta$$

A function call is similar to a branch instruction in that the actual jump to the first instruction of the function is performed at the warp level. The expression e is either a valid function name, in which case $\mathcal{E}[\![-]\!]$ returns a valid program address, or a register storing the first address of a function with a matching signature. Otherwise, program behavior is unpredictable. Additionally, the local variable environment of the called function is pushed onto the thread's call stack. Return addresses are also kept track of at the warp level instead of individually for each thread.

$$\text{(call)}\ \langle \theta \triangleright (\kappa \triangleright \vec{v}), wl \mid \texttt{call}\ e, \phi, \varrho \rangle \xrightarrow[\texttt{call}\ wl\ \mathcal{E}[\![e]\!]\phi\varrho\kappa]{}^{\Theta} \langle \theta \triangleleft \phi_{localEnv}(\mathcal{E}[\![e]\!]\phi\varrho\kappa) :: \vec{v} \rangle$$

The following control flow operations also have no meaning at the thread level and are only used to inform the warp which threads actually executed the instructions. Only labels are valid for `ssy`, `preBrk`, and `preRet` instructions because all threads of the same warp must agree on the address specified by these instructions. The semantics of the control flow instructions are explained in greater detail in section 5.4.1.

$$\text{(ssy)}\ \langle \theta \triangleright \kappa \mid \texttt{ssy}\ l, \phi, \varrho \rangle \xrightarrow[\texttt{ssy}\ \mathcal{E}[\![l]\!]\phi\varrho\kappa]{}^{\Theta} \theta$$

$$\text{(preBrk)}\ \langle \theta \triangleright \kappa \mid \texttt{preBrk}\ l, \phi, \varrho \rangle \xrightarrow[\texttt{preBrk}\ \mathcal{E}[\![l]\!]\phi\varrho\kappa]{}^{\Theta} \theta$$

$$\text{(preRet)}\ \langle \theta \triangleright \kappa \mid \texttt{preRet}\ l, \phi, \varrho \rangle \xrightarrow[\texttt{preRet}\ \mathcal{E}[\![l]\!]\phi\varrho\kappa]{}^{\Theta} \theta$$

$$\text{(sync)}\ \langle \theta \mid \texttt{sync}, \phi, \varrho \rangle \xrightarrow[\texttt{sync}]{}^{\Theta} \theta$$

$$\text{(break)}\ \langle \theta \triangleright wl \mid \texttt{break}, \phi, \varrho \rangle \xrightarrow[\texttt{break}\ wl]{}^{\Theta} \theta$$

$$\text{(exit)}\ \langle \theta \triangleright wl \mid \texttt{exit}, \phi, \varrho \rangle \xrightarrow[\texttt{exit}\ wl]{}^{\Theta} \theta$$

Returning from a function and jumping back to the call site is handled at the warp level as mentioned above. However, at the thread level the topmost local variable environment must be popped off the call stack.

$$\text{(return)}\ \langle \theta \triangleright (\kappa \triangleright \nu :: \vec{v}), wl \mid \texttt{ret}, \phi, \varrho \rangle \xrightarrow[\texttt{return}\ wl]{}^{\Theta} \langle \theta \triangleleft \vec{v} \rangle$$

The `vote` instruction sends a vote action to the thread's warp. The result of the vote is computed by the warp and stored in destination register r. The rule evaluates the voting expression and stores it as a Boolean value inside the action. The memory actions to store the result of the vote in the destination registers specified by the threads are generated at the warp level, hence the warp needs to know the indices and register offsets of the threads participating in the vote.

$$\text{(vote)}\ \langle \theta \triangleright \kappa, tid, ro, lo, wl \mid instr \triangleright \texttt{vote}\ vm\ r\ e, \phi, \varrho \rangle \xrightarrow[\texttt{vote}\ \mathcal{V}[\![r]\!]\phi\kappa\ \mathcal{B}[\![e]\!]\phi\varrho\kappa\ wl\ tid\ ro\ lo\ regs(instr,\phi)]{}^{\Theta} \theta$$

A memory barrier or memory fence causes the thread to wait for all prior memory operations to be performed at a certain level. The thread is allowed to resume execution once all prior writes are visible to all threads of the specified level and all prior reads can no longer be affected by writes of another thread at the specified level. For instance, when a thread continues execution after a memory fence of the thread block level, we know that all other threads of the same block will subsequently read the values previously written by the thread. Furthermore, all pending read operations by the thread can no longer be affected by a write of some other thread of the block. It is unspecified when exactly a memory operation is no longer affect by another one, hence memory barriers are not included in our formalization of the memory environment. To support memory barriers in the program semantics, we store the level of the memory barrier a thread waits for. As long as the stored value is not ε, the thread cannot be scheduled because a prior memory fence has not yet completed. At the device level, we define an abstract function that checks whether a thread's memory fence is completed. If so, the function resets the thread's memory barrier level to ε which allows the warp scheduler to schedule the thread's warp again. The (membar) rule is the only rule that does not generate a thread action.

$$\text{(membar)}\ \langle \theta \mid \mathtt{membar}\ \mathit{mbl}, \phi, \varrho \rangle \rightarrow^{\Theta} \langle \theta \triangleleft \mathit{mbl} \rangle$$

A thread block barrier is a synchronization point for all threads of the same thread block. Once a thread reaches a barrier, it is not allowed to continue program execution until a specific amount of threads of the same block has reached the barrier instruction as well. Barriers are identified by an index and the amount of threads participating in the barrier can optionally be specified. A thread that processes a `bar.sync` or `bar.red` instruction automatically establishes a memory barrier of the `.cta` level. The thread is not allowed to resume execution as long as either of the two barriers is not yet completed. Both the memory barrier and the synchronization barrier are dealt with at higher levels of the hierarchy and have no meaning at the thread level. If the evaluation of expression e does not yield a valid barrier index, the program is stuck.

$$(\mathrm{bar}_{\mathrm{sync}})\ \langle \theta \triangleright \kappa \mid \mathtt{bar.sync}\ e\ e'_{\varepsilon}, \phi, \varrho \rangle \xrightarrow[\mathtt{bar}\ \mathcal{E}\llbracket e \rrbracket \phi\varrho\kappa\ \mathcal{E}_{\varepsilon}\llbracket e'_{\varepsilon} \rrbracket \phi\varrho\kappa]{}{}^{\Theta} \langle \theta \triangleleft \mathtt{.cta} \rangle$$

The `bar.arrive` instruction only marks the arrival of the thread at the barrier but does not cause any waiting due to thread block synchronization or an implicit memory barrier. At the thread level, the only difference to the `bar.sync` instruction is that the number of threads participating in the barrier is not optional and therefore must be specified.

$$(\mathrm{bar}_{\mathrm{arrive}})\ \langle \theta \triangleright \kappa \mid \mathtt{bar.arrive}\ e\ e', \phi, \varrho \rangle \xrightarrow[\mathtt{bar}\ \mathcal{E}\llbracket e \rrbracket \phi\varrho\kappa\ \mathcal{E}\llbracket e' \rrbracket \phi\varrho\kappa]{}{}^{\Theta} \theta$$

`bar.red` is similar to `bar.sync` in that the thread has to wait for the barrier to complete before it is allowed to execute further instructions. An implicit thread block level memory fence is initiated as well. Additionally, the thread block performs a reduction operation once the barrier completes. The result of the reduction is stored in register r; the thread block generates corresponding memory actions once it has computed the result of the reduction. The expression e_2 acts as the input of the reduction operation. The rule evaluates e_2 and stores it as a Boolean value inside the communication action.

$$(\mathrm{bar}_{\mathrm{red}})\quad \langle \theta \triangleright \kappa, \mathit{tid}, \mathit{ro}, \mathit{lo} \mid \mathit{instr} \triangleright \mathtt{bar.red}\ \mathit{br}\ r\ e\ e_{1,\varepsilon}\ e_2, \phi, \varrho \rangle$$

$$\xrightarrow[\mathtt{bar}\ \mathcal{E}\llbracket e \rrbracket \phi\varrho\kappa\ \mathcal{E}_{\varepsilon}\llbracket e_{1,\varepsilon} \rrbracket \phi\varrho\kappa\ \mathcal{V}\llbracket r \rrbracket \phi\kappa\ \mathcal{B}\llbracket e_2 \rrbracket \phi\varrho\kappa\ \mathit{tid}\ \mathit{ro}\ \mathit{lo}\ \mathit{regs}(\mathit{instr},\phi)]{}{}^{\Theta} \langle \theta \triangleleft \mathtt{.cta} \rangle$$

The rules for atomic memory operations and non-atomic load and store operations generate a memory action containing the data specified by the instruction. At the device level, the action is added to the memory environment where a corresponding memory program is created to service the request.

$$\text{(atom)}\quad \langle \theta \triangleright \kappa, \mathit{tid}, \mathit{ro}, \mathit{lo} \mid \mathit{instr} \triangleright \mathtt{atom}\ \mathit{ss}\ \mathit{aop}\ \mathit{size}\ r\ e_1\ e_2\ e_{3,\varepsilon}, \phi, \varrho \rangle$$

$$\xrightarrow[\mathcal{V}\llbracket r \rrbracket \phi\varrho\kappa \leftarrow \mathcal{E}\llbracket e_1 \rrbracket \phi\varrho\kappa\ \mathit{ss}\ \mathit{aop}\ \mathit{size}\ \mathcal{E}\llbracket e_2 \rrbracket \phi\varrho\kappa\ \mathcal{E}_{\varepsilon}\llbracket e_{3,\varepsilon} \rrbracket \phi\varrho\kappa\ \mathit{tid}\ \mathit{ro}\ \mathit{lo}\ \mathit{regs}(\mathit{instr},\phi)]{}{}^{\Theta} \theta$$

$$\text{(st)}\quad \langle \theta \triangleright \kappa, tid, ro, lo \mid instr \triangleright \texttt{st}\ volatile_\varepsilon\ ss\ cop_\varepsilon\ size\ e\ r, \phi, \varrho\rangle \xrightarrow[\mathcal{E}\llbracket e\rrbracket\phi\varrho\kappa\ ss\ size\ \mathcal{E}\llbracket r\rrbracket\phi\varrho\kappa\ volatile_\varepsilon\ cop_\varepsilon\ tid\ ro\ lo\ regs(instr,\phi)]{}{}^{\Theta}\ \theta$$

$$\text{(ld)}\quad \langle \theta \triangleright \kappa, tid, ro, lo \mid instr \triangleright \texttt{ld}\ volatile_\varepsilon\ ss\ cop_\varepsilon\ size\ r\ e, \phi, \varrho\rangle \xrightarrow[\mathcal{V}\llbracket r\rrbracket\phi\kappa \leftarrow \mathcal{E}\llbracket e\rrbracket\phi\varrho\kappa\ ss\ size\ volatile_\varepsilon\ cop_\varepsilon\ tid\ ro\ lo\ regs(instr,\phi)]{}{}^{\Theta}\ \theta$$

The `mov` instruction evaluates an expression and stores the resulting value in the specified destination register. The instruction can be used to load constant values into a register, to copy the value of a register to another one, to load the program address denoted by a label or function into a register, or to load the address pointed to by a non-register variable into a register.

$$\text{(mov)}\quad \langle \theta \triangleright \kappa, tid, ro, lo \mid instr \triangleright \texttt{mov}\ r\ e, \phi, \varrho\rangle \xrightarrow[\mathcal{V}\llbracket r\rrbracket\phi\kappa \leftarrow \mathcal{E}\llbracket e\rrbracket\phi\varrho\kappa\ tid\ ro\ lo\ regs(instr,\phi)]{}{}^{\Theta}\ \theta$$

`setp` uses a comparison operator to compare two operands and stores the resulting value in the destination register. We use the abstract function *comp* to compute the resulting value in accordance with the default semantics of the comparison operators supported by PTX.

$$\text{(setp)}\quad \langle \theta \triangleright \kappa, tid, ro, lo \mid instr \triangleright \texttt{setp}\ comp\ r\ e_1\ e_2, \phi, \varrho\rangle \xrightarrow[\mathcal{V}\llbracket r\rrbracket\phi\kappa \leftarrow comp(\mathcal{E}\llbracket e_1\rrbracket\phi\varrho\kappa,\mathcal{E}\llbracket e_2\rrbracket\phi\varrho\kappa)\ tid\ ro\ lo\ regs(instr,\phi)]{}{}^{\Theta}\ \theta$$

5.3.3 Conditional Thread Execution

When a warp is scheduled for execution, all of its thread execute the instruction pointed to by the warp's program counter. However, a thread might not be allowed to execute an instruction because its guard evaluates to false. Additionally, the warp may decide to disable a thread because the active threads execute a branch that the disabled thread has not taken. To simplify the rules of the warp semantics presented in the next section, we always ask threads to execute the current instruction regardless of their guard or their activation state in the warp. It is the thread's own responsibility to check whether it should really execute the instruction or whether it should remain idle. To avoid cluttering up the rules of transition system $\rightarrow^{\Theta}$ with these checks, we define transition system $\rightarrow^{\Theta_e}$ that performs these checks once in a central place and uses the semantics of $\rightarrow^{\Theta}$ if the thread is allowed to execute the instruction.

The warp passes its active mask down to the threads. The active mask maps each warp lane to an active state. The thread checks the warp's active mask to determine whether its warp lane is active or inactive and thus whether or not it has to idle.

$$\begin{aligned} ActiveState &= \{\ \text{active},\ \text{inactive}\ \} \\ \alpha \in ActiveMask &= WarpLane \rightarrow ActiveState \end{aligned}$$

Additionally, the warp hands a thread register look up function down to the threads. To evaluate expressions, however, an expression register look up function is required. The latter can be constructed from the former by invoking the thread register look up function with the thread's index and register offset. For reasons of brevity, we do not create the expression register look up function in each rule of transition system $\rightarrow^{\Theta}$ but rather in the rules of transition system $\rightarrow^{\Theta_e}$ only.

$$\varrho \in \mathit{ThreadRegLookup} = \mathit{ThreadIdx} \times \mathit{RegOffset} \rightarrow \mathit{ExprRegLookup}$$

Transition system $\rightarrow^{\Theta_e}$ does not change the thread actions generated by $\rightarrow^{\Theta}$. Thread actions are sent unchanged to the warp transition system.

$$\begin{aligned} \mathit{Configurations} : {} & \langle \theta \mid \phi, pc, \alpha, \varrho \rangle \in \mathit{Thread} \times \mathit{ProgEnv} \times \mathit{ProgAddr} \times \mathit{ActiveMask} \\ & \qquad \times \mathit{ThreadRegLookup}, \\ & \theta \in \mathit{Thread} \\ \mathit{Transitions} : {} & \langle \theta \mid \phi, pc, \alpha, \varrho \rangle \xrightarrow[\mathit{tact}_\varepsilon]{}^{\Theta_e} \theta' \end{aligned}$$

If the thread is active according to its warp's active mask and the guard evaluates to true, the thread executes the current instruction as defined by the rules of $\rightarrow^{\Theta}$. Otherwise, the thread does not do anything at all and remains idle.

$$(\mathrm{Exec_{tt}}) \quad \frac{\langle \theta \mid \phi_{\mathit{instrEnv}}(pc), \phi, \varrho(\mathit{tid}, \mathit{ro}) \rangle \xrightarrow[\mathit{tact}_\varepsilon]{}^{\Theta} \theta'}{\langle \theta \triangleright \kappa, \mathit{ro}, \mathit{tid}, \mathit{wl} \mid \phi, pc, \alpha, \varrho \rangle \xrightarrow[\mathit{tact}_\varepsilon]{}^{\Theta_e} \theta'}$$

$$\text{if} \quad \alpha(\mathit{wl}) = \text{active} \wedge \mathcal{B}[\![\phi_{\mathit{guardEnv}}(pc)]\!]\phi\varrho(\mathit{tid}, \mathit{ro})\kappa$$

$$(\mathrm{Exec_{ff}}) \quad \langle \theta \triangleright \kappa, \mathit{ro}, \mathit{tid}, \mathit{wl} \mid \phi, pc, \alpha, \varrho \rangle \rightarrow^{\Theta_e} \theta$$

$$\text{if} \quad \alpha(\mathit{wl}) \neq \text{active} \vee \neg\mathcal{B}[\![\phi_{\mathit{guardEnv}}(pc)]\!]\phi\varrho(\mathit{tid}, \mathit{ro})\kappa$$

5.4 Warp Semantics

Warps execute several threads in lockstep. The formalization of the warp semantics assumes that all threads of the same warp are executed concurrently. In reality, however, this is not the case. On GPUs based on the Fermi architecture, each warp can only use 16 of the 32 CUDA cores to execute a floating point instruction, thus threads are split into two groups of 16 threads each and process the instruction over the course of two clock cycles. For instructions that use the streaming multiprocessor's special function units, the warp is split into eight groups of four threads each, so eight clock cycles are needed to process the instruction. However, we do not represent this behavior in our semantics. While it is possible that an instruction scheduled at a later point in time finishes before an instruction scheduled beforehand, the later instruction cannot affect the outcome of the earlier one. For that to happen, the later instruction would have to write to a register that the earlier instruction reads; but the earlier instruction blocks its source registers, hence no other instruction could possibly write to those registers. It is also not possible that a later instruction writes to a memory location that an earlier instruction is currently issuing read requests for, because there is only one set of load-store units on the streaming multiprocessor, hence another load-store instruction cannot be scheduled before the earlier one finishes. Still, the memory environment might choose to reorder incoming memory requests, but this is an unrelated problem.

A warp can either be in active or completed state. When a thread block is issued to a streaming multiprocessor, the SM creates all warps with active state. The warp schedulers subsequently schedule all warps until all of a warp's threads processed an `exit` instruction. When this happens, the warp changes to the completed state and its resources are eventually freed, allowing further thread blocks to be issued to the SM as soon as there are enough free resources.

$$state \in WarpState = \{\ \text{active},\ \text{completed}\ \}$$

Aside from executing several threads concurrently, the warps are also responsible for merging the memory actions generated by the threads into one warp memory action. The memory subsystem uses this information to coalesce memory requests into fewer memory transactions on a per-warp basis. Additionally, the warp handles `vote` instructions and adds some further information to barrier actions that are later processed by the semantics of thread blocks. The warp handles all control flow and vote actions generated by the threads, so these are not passed further up the hierarchy.

$$\begin{aligned} wact_{mem} \in MemAct_{\Omega} &::= MemAct^{*}_{\Theta} \\ wact_{bar} \in BarrierAct_{\Omega} &::= \texttt{bar}\ BarrierType\ BarrierRed_{\varepsilon}\ BarrierAct^{*}_{\Theta} \\ wact \in Act_{\Omega} &::= MemAct_{\Omega}\ \mid\ BarrierAct_{\Omega} \end{aligned}$$

We introduce the domain of warps which stores the warp's state, the threads it manages, and the indices of the barriers it waits for. It is possible that — though undocumented, so this is just an assumption as explained in section 5.5.1 — some threads that execute a barrier instruction specify a different barrier index. If so, we do not allow the warp scheduler to schedule the warp again until all barriers have completed, i.e. the list of barrier indices stored in the warp is empty. Furthermore, a warp manages its execution state. We define the domain of execution states in the next section.

$$\omega \in Warp = WarpState \times ExecState \times BarrierIdx^{*} \times Thread^{WarpSize}$$

In section 5.4.1, we formalize the branching algorithm used by a warp to handle divergent control flow of its threads. We formalize the handling of communication actions at the warp level in section 5.4.2 and define the rules making up the warp semantics in section 5.4.3.

5.4.1 Formalization of the Branching Algorithm

GPUs employ warps instead of individual thread execution for reasons of performance: Instruction fetching and decoding costs are amortized over 32 threads, memory requests of 32 threads might be coalesced into fewer ones for improved memory access efficiency, and the warp schedulers only have to consider warps instead of threads of which there are fewer by a factor of $\frac{1}{32}$. However, it does get more complicated once the control flow of individual threads of the same warp diverges, i.e. when a warp's threads execute data-dependant branching instructions, function calls, breaks, or returns. In fact, the branching algorithm is not equivalent to individual thread branching as found on x86 CPUs as we show in chapter 6. While we can prove the equivalence for terminating programs, the branching algorithm can negatively affect program behavior in the general case. In this section, however, we only focus on how the algorithm works. The CUDA specification does not contain many details about the algorithm, so we base this discussion on a patent filed by Nvidia [26]. As already said about the patents used to formalize the memory environment, we do not know whether the hardware actually works the way the patent depicts. However, our observations indicate that the patent indeed describes the hardware implementation: The algorithm described in the patent relies on instructions that are not available in PTX but are indeed injected into the compiled program by the PTX compiler. Furthermore, the depicted algorithm matches the behavior that can be observed in several test cases. We therefore assume that the algorithm outlined informally in patent [26] and formalized in this section accurately reflects the hardware implementation of Fermi-based GPUs.

The branching algorithm relies on the following domains and functions. We give a formal definition and a brief overview of the motivation behind them. We then informally illustrate how the branching algorithm works and subsequently present a formalization.

We repeat the definition of the active mask already shown in section 5.3.3 for reasons of completeness. The active mask stores the activation state of each thread. A thread whose warp lane is active according to the active mask participates in an execution step, whereas inactive threads do not. α^{inact} denotes the active masks where all threads are inactive.

$$\begin{aligned} \mathit{ActiveState} &= \{\, \text{active},\ \text{inactive} \,\} \\ \alpha \in \mathit{ActiveMask} &= \mathit{WarpLane} \rightarrow \mathit{ActiveState} \end{aligned}$$

Additionally, the warp manages a disable mask that keeps track of if and why a thread is disabled. The disable mask ensures that threads that executed a return or break statement are not reactivated before all threads of the warp complete the execution of the function call or the loop. Furthermore, it makes sure that threads that completed the execution of the program are not reactivated again. Without a disable mask, PTX programs could not use nested control flow instructions like branches inside a path of another branch or inside a function body.

$$\begin{aligned} \mathit{DisableState} &= \{\, \text{enabled},\ \text{break},\ \text{return},\ \text{exit} \,\} \\ \delta \in \mathit{DisableMask} &= \mathit{WarpLane} \rightarrow \mathit{DisableState} \end{aligned}$$

The warp uses a stack to keep track of the different code paths its threads need to execute. There are four different types of tokens that can be pushed onto the stack. Besides its type, a token also stores an active mask and a program counter that identify the threads that should be activated and the address of the instruction that should be executed once the token is popped off the stack. The token's type determines how the disable mask affects thread reactivation.

$$\begin{aligned} \mathit{TokenType} &= \{\, \text{sync},\ \text{diverge},\ \text{call},\ \text{break} \,\} \\ \tau \in \mathit{ExecToken} &= \mathit{TokenType} \times \mathit{ActiveMask} \times \mathit{ProgAddr} \end{aligned}$$

We already used the domain of execution states to define the domain of warps. A warp's execution state consists of the warp's current program address, the warp's current active and disable masks as well as the token stack. It seems that the hardware has special memory dedicated to storing a token stack for each warp. In the theoretical worst case, however, the size of the token stack is unlimited. The hardware is capable of spilling the oldest entries to global memory if the stack becomes too large, but eventually, it will run out of memory. Spilling parts of the stack to memory and loading it back in when needed happens behind the scenes and is completely transparent to CUDA programs. To avoid further complicating the formalization, we assume that the streaming multiprocessors can store token stacks of unlimited size. The stack allows a CUDA program to arbitrarily branch, limited only by the size of the available memory.

$$\varsigma \in \mathit{ExecState} = \mathit{ProgAddr} \times \mathit{ActiveMask} \times \mathit{DisableMask} \times \mathit{ExecToken}^{*}$$

If the number of threads belonging to a thread block is not a multiple of the warp size, the last warp is underpopulated. What the hardware does in that case is unspecified, so we assume that dummy threads are allocated whose active mask is set to inactive and whose disable mask is set to exit. Consequently, these dummy threads will never execute a single instruction.

Informal Overview

We informally illustrate how the branching algorithm works based on the following programs and figures. The programs presented in this section are not valid PTX programs: Register declarations and data types are omitted for reasons of brevity. Moreover, the control flow instructions normally injected by the PTX compiler are shown in the programs as these are the main focus of attention.

```
1    setp .eq p, a, b;
2    ssy 10;
3    @p bra DIVERGE;
4    sub c, a, b;
5    bra SYNC;
6  DIVERGE:
7    mul c, a, b;
8  SYNC:
9    sync;
10   div r, c, d;
```

Listing 5.5: A PTX program with a conditional branch statement

Program 5.5 shows a PTX program with a conditional branch statement in line 3. For the sake of this discussion, we assume the program is executed by two threads; the first one takes the branch and the second one does not. We explain how program 5.5 is executed step by step. Figure 5.2 shows the warp's execution state for each step. The program counter is the address of the instruction that is executed during a step; the address of an instruction is equal to the line number shown in listing 5.5. The active mask shows which threads are active after the completion of a step. Because of space constraints, we use a bit notation for the active mask. For instance, an active mask value of 01 means that thread 1 is inactive whereas thread 2 is active. The stack column lists the tokens on the stack after the execution of a step. Initially, the program counter is 1, both threads are active according to the active mask and enabled according to the disable mask, and the stack is empty. The disable mask is not shown because both threads remain enabled during the execution of the program.

pc	α	$\vec{\tau}$
1	11	ε
2	11	(sync, 11, 10)
3	10	(diverge, 01, 4) :: (sync, 11, 10)
7	10	(diverge, 01, 4) :: (sync, 11, 10)
9	01	(sync, 11, 10)
4	01	(sync, 11, 10)
5	01	(sync, 11, 10)
9	11	ε
10	11	ε

Figure 5.2: Execution state $\varsigma \triangleright pc, \alpha, \vec{\tau}$ during the execution of program 5.5

In line 1, the threads execute the `setp` instruction. As it is not a control flow instruction, neither the active mask nor the execution stack changes. We assume that predicate register `p` is true for thread 1 and false for thread 2. The warp increments the program counter to 2 and the threads execute the `ssy` instruction. The `ssy` instruction causes the warp to push a sync token onto the stack. Subsequently, the control flow will diverge and both paths are executed with only one of the threads being active. However, the division at line 10 could be executed by both threads in lockstep again. Of course, this is not necessary for correctness, but highly desirable for reasons of performance. The `sync` instruction in line 9 will resynchronize both threads such that the division is processed by both threads again. The resynchronization works by popping the sync token off the stack that the `ssy` instruction pushed onto the stack. The token is of type sync, indicating that a previous level of thread synchronization should be reestablished without making any changes to the disable mask. The token contains the current active mask. Once the token is popped off the stack, the warp's active mask — which at that point only has one of the threads active — is replaced by the token's one, thus both threads are active again. Additionally, the warp's program counter is set to the program counter stored in the token, which is the address of the division instruction in line 10. Hence, after diverging on two different paths that are serially executed, both threads execute all instructions in lockstep again beginning with line 10.

Program execution then continues at line 3. Since the predicate is true only for thread 1, thread 1 executes the branch instruction and jumps to line 7, whereas thread 2 does not take the branch and executes line 4 next. We call the path executed by thread 1 the taken path and the path executed by thread 2 the not-taken path. Now that there are two different paths to execute, the warp has to serialize execution. By convention, the GPU always executes the taken path first. Consequently, thread 2 is set to inactive because it should not execute the multiplication in line 7. Additionally, the warp pushes a diverge token onto the stack. The purpose of a diverge token is to keep track of other paths that need to be executed later on. In this case, thread 2 must process the subtraction in line 4. Consequently, the token's program counter is set to 4 and the active mask stored in the token activates thread 2 and deactivates thread 1. So when the diverge token is popped off the stack, thread 2 will execute the not-taken path.

The warp now executes the taken path. Thus, thread 1 performs the multiplication in line 7. As the multiplication is not a control flow instruction, both the active mask and the stack remain unchanged. The warp increments the program counter to the next address and thread

1 encounters the `sync` instruction. `sync` causes the warp to pop a token off the stack. The diverge token sets the warp's program counter to address 4 and the active mask to the one stored in the token. Here this results in thread 1 being deactivated and thread 2 executing the not-taken path of the conditional branch in line 3. Thread 2 begins the execution of the not-taken path by processing the subtraction in line 4. Again, this is not a control flow instruction so the active mask and the stack are not changed. The warp increments the program counter and thread 2 encounters the branch instruction in line 5. The branch instruction causes thread 2 to skip the multiplication in line 7, because the multiplication may only be performed by threads taking the taken path. The branch instruction causes no further divergence in this case because all active threads executing the current path, here only thread 2, take the branch. Uniform branches do not modify the active mask or the stack. Thus, program execution continues at line 9, where the `sync` instruction is encountered again. As before, the `sync` instruction forces the warp to pop a token off the stack. Now the sync token lies on top of the stack, thus when it is popped it causes both threads to be active again and sets the program counter to line 10. Next, both threads execute the division instruction at line 10 concurrently.

```
1    ssy 13;
2    bra r;
3  PATH-1:
4    mul c, a, b;
5    bra SYNC;
6  PATH-2:
7    add c, a, b;
8    bra SYNC;
9  PATH-3:
10   sub c, a, b;
11 SYNC:
12   sync;
13   div r, c, d;
```

Listing 5.6: A PTX program with an indirect branch statement

Program 5.6 and figure 5.3 demonstrate how the branching algorithm handles indirect branches. We assume that the warp size is four and that one thread executes the path denoted by the label PATH-1, two threads execute PATH-2, and the remaining thread executes PATH-3. In figure 5.3, we again list the current program counter as well as the active mask and the stack after the execution of the instruction pointed to by the program counter. As the disable mask does not change in this example, figure 5.3 omits it. Again, we use the bit notation for the active mask, so 0110 means that thread 1 and 4 are disabled, whereas threads 2 and 3 are active. Initially, all threads are active according to the active mask and enabled according to the disable mask, the program counter is 1, and the stack is empty.

At first, the `ssy` instruction is encountered at line 1. As before, it pushes a sync token onto the stack that is used to resynchronize all threads after all paths of the indirect branch instruction in line 2 are completed. The program counter is incremented to 2 and all four threads execute the `bra` instruction. In this case, the branch instruction has no guard, but it is indeed possible to use guarded indirect branches. There are three possible paths in this case. For indirect branches with more than one distinct target address, the patent does not say which address is executed first. Hence, we randomly choose PATH-2. Let's assume that

pc	α	$\vec{\tau}$
1	1111	(sync, 1111, 13)
2	0101	(diverge, 1010, 2) :: (sync, 1111, 13)
7	0101	(diverge, 1010, 2) :: (sync, 1111, 13)
8	0101	(diverge, 1010, 2) :: (sync, 1111, 13)
12	1010	(sync, 1111, 13)
2	1000	(diverge, 0010, 2) :: (sync, 1111, 13)
10	1000	(diverge, 0010, 2) :: (sync, 1111, 13)
12	0010	(sync, 1111, 13)
2	0010	(sync, 1111, 13)
4	0010	(sync, 1111, 13)
5	0010	(sync, 1111, 13)
12	1111	ε
13	1111	ε

Figure 5.3: Execution state $\varsigma \triangleright pc, \alpha, \vec{\tau}$ during the execution of program 5.6

threads 2 and 4 execute PATH-2, so threads 1 and 3 are disabled in the new active mask. If more than one thread branches to the same target address, all those threads execute their path concurrently. Additionally, the warp pushes a diverge token onto the stack. The token's active mask is the not-taken mask, i.e. all threads that later have to execute another path. The token's program address is the address of the branch instruction, meaning that the branch instruction will be executed again once the token is popped off the stack. But since threads 2 and 4 are not active in the token's active mask, it is guaranteed that the branch instruction will execute a different path next time.

Threads 2 and 4 perform the addition in line 7 and continue to execute the uniform branch in line 8. The threads skip the subtraction of PATH-3 and process the `sync` statement in line 12. As before, the `sync` instruction pops a token off the stack. As the diverge token is the topmost element of the stack, threads 1 and 3 are activated, threads 2 and 4 are deactivated and the warp's program counter is set to 2 again. This time, however, threads 2 and 4 are not active, hence the branch definitely takes another path. Again, it is undefined which one. We assume thread 1 is now allowed to execute PATH-3, so thread 1 remains active in the warp's active mask and thread 3 is the only thread to remain active in the token's active mask. The token's program counter is set to the indirect branch instruction's address again, as there is still a path left that needs to be executed.

Thread 1 executes the subtraction in line 10 and subsequently encounters the `sync` instruction in line 12. The diverge token is popped of the stack, consequently only thread 3 is active and executes the `bra` instruction in line 2 again. This time, thread 3 is allowed to execute PATH-1. As the branch is now no longer divergent, no diverge token is pushed onto the stack. Thread 3 performs the multiplication in line 4 and skips over the two other paths because of the uniform branch in line 5. For the last time, the `sync` instruction at line 12 is processed. The sync token is popped off the stack, causing all threads to process the division in line 13 concurrently.

Program 5.7 is intended to demonstrate the necessity of the disable mask. Except for lines 7 and 11, program 5.7 is identical to program 5.5. Here, one of the paths might cause a thread to return from the currently executed function. Consequently, threads that execute the taken

```
1   setp .eq p, a, b;
2   ssy 10;
3   @p bra DIVERGE;
4   sub c, a, b;
5   bra SYNC;
6 DIVERGE:
7   @p2 ret;
8 SYNC:
9   sync;
10  div r, c, d;
11  ret;
```

Listing 5.7: A PTX program with nested control flow instructions

path should only perform the division in line 10 if the predicate `p2` is false. Otherwise, they execute the return statement in line 7 and should no longer execute any instructions until all threads of the warp have completed execution of the function. Figure 5.4 illustrates the changes made to the warp's execution state as program 5.7 is processed. This time, we also show the disable mask in a shortened notation. A value of 0r0 means that thread 2 is disabled waiting for the other threads of the warp to return from a function, whereas threads 1 and 3 are enabled. Furthermore, we execute the program with 3 threads. Threads 2 and 3 take the taken path where only thread 2 executes the return, and thread 1 takes the not-taken path. Initially, the warp's program counter is set to 1, all threads are active and enabled and there is a call token already on the stack. The `preRet` instruction not shown in program 5.7 must be used to push a call token onto the stack before a function call is made. Once all threads return from the function, the call token is popped off the stack and execution continues at the instruction following the function call, denoted by x in this example.

pc	α	δ	$\vec{\tau}$
1	111	000	(call, 111, x)
2	111	000	(sync, 111, 10) :: (call, 111, x)
3	011	000	(diverge, 100, 4) :: (sync, 111, 10) :: (call, 111, x)
7	001	0r0	(diverge, 100, 4) :: (sync, 111, 10) :: (call, 111, x)
9	100	0r0	(sync, 111, 10) :: (call, 111, x)
4	100	0r0	(sync, 111, 10) :: (call, 111, x)
5	100	0r0	(sync, 111, 10) :: (call, 111, x)
9	101	0r0	(call, 111, x)
10	101	0r0	(call, 111, x)
11	111	000	ε
x	111	000	ε

Figure 5.4: Execution state $\varsigma \triangleright pc, \alpha, \delta, \vec{\tau}$ during the execution of program 5.7

Up to the point that thread 2 executes the `ret` statement in line 7, there are no important differences to point out compared to program 5.5. In line 7, threads 2 and 3 are active but only thread 2 returns from the function whereas thread 3 skips the return and its active and disable mask remain unaffected. Consequently, thread 2 becomes inactive according to the active mask and its disable mask is set to return. Yet, the active mask of the sync token still mentions thread 2 as being active even though thread 2 must not be reactivated once the sync token is popped off the stack. The disable mask for thread 2 guarantees that only a call token reactivates the thread again, regardless of the sync token normally doing so. An alternative to using a disable mask would be to walk up the stack and remove all disabled threads from the tokens' active masks up to some token that indeed reactivates the thread again. But as parts of the stack might be spilled to global memory, modifying the stack is potentially a slow operation. The disable mask avoids this performance issue.

Next, only thread 3 executes the `sync` statement. The divergence token is popped off the stack and thread 1 executes the not-taken path. When thread 1 encounters the `sync` statement in line 9 again, the sync token is popped off the stack. Without the disable mask, popping the sync token would reactivate thread 2 and it would execute the division in line 10 even though it should not. Thus, the warp removes all threads from the token's active mask that have been disabled in the meantime. That way it is guaranteed that only threads 1 and 3 execute the division together. Subsequently, they encounter the unguarded return statement in line 11. Since all active threads execute the return, the warp pops a token off the stack. In this example, the call token is popped. All threads are active in the token's active mask and because all threads returned from the function, thread 2 is reactivated and its disable mask is set to enabled again. All three threads continue to execute the instruction at program address x concurrently.

The token that is popped off the stack because of the return statement could also be a diverge, sync, or break token instead of a call token. This happens when all threads executing the same path of a branch or loop return from the function. In that case, the remaining paths are executed, if there are any. It is also possible that popping a token off the stack does not activate any threads. Suppose both paths of a branch have all threads execute a return instruction. When the threads of the not-taken path execute the return, there still is a sync token on the stack. However, the token does not activate any threads because all of them are disabled waiting for a function return. Hence, the warp immediately pops another token off the stack. If that token is a call token, it reactivates the threads and program execution continues. Otherwise, additional tokens are popped off the stack until the warp encounters a token that reactivates at least one thread. If all threads have executed an `exit` instruction, there are no threads that will be activated again and the warp has completed the execution of the program.

Formal Semantics of the Control Flow Instructions

We formalize the branching algorithm by defining a formal semantics for the control flow instructions. At the thread level, control flow instructions have no meaning — except for `call` and `ret` where the threads modify their call stack —, so we define their semantics at the warp level. We highlight any differences between our formalization and the informal specification in patent [26] and explain why it is necessary to deviate from the original semantics.

First, we define some helper functions that are used by the formalization of the branching algorithm. The *uni* function decides whether all active threads processed an instruction. When threads execute a control flow instruction, they often pass their warp lane inside the control flow action to the warp. The *uni* function checks whether all active threads passed their warp lane to the warp, i.e. whether there are any threads that did not execute the instruction because their guard evaluated to false. For instance, a direct branch is uniform if and only if all active threads execute the `bra` instruction, hence if the instruction has no guard or the guard evaluates to true for all active threads.

$$\begin{aligned}
&\mathit{uni} : \mathit{ActiveMask} \times \mathit{WarpLane}^* \to \mathbb{B}\\
&\mathit{uni}(\alpha, wl_1 \dots wl_n) \Leftrightarrow \forall wl \in \mathit{WarpLane} \,.\, \alpha(wl) = \text{active} \Rightarrow wl \in wl_1 \dots wl_n
\end{aligned}$$

Similarly, we define an overloaded version of the *uni* function that also takes the target addresses into account. Branches and function calls can be divergent because of guards and because of different target addresses. In general, a branch is uniform if and only if all active threads executed the branch instruction and all of them specified the same target address.

$$\begin{aligned}
&\mathit{uni} : \mathit{ActiveMask} \times \mathit{WarpLane}^* \times \mathit{ProgAddr}^* \to \mathbb{B}\\
&\mathit{uni}(\alpha, wl_1 \dots wl_n, pc_1 \dots pc_n) \Leftrightarrow \mathit{uni}(\alpha, wl_1 \dots wl_n) \wedge \forall i, j \in \{1 \dots n\} \,.\, pc_i = pc_j
\end{aligned}$$

The *taken* function computes the taken mask of a control flow instruction. All threads are inactive except for those that executed the instruction and intend to jump to the target address chosen by the warp. For instance, a thread is active in the taken mask of a direct branch if and only if it is active in the warp's current active mask and its guard evaluated to true. For indirect branches, a thread is active in the taken mask for the same reasons as for direct branches, plus if its target address matches the address chosen by the warp.

$$\begin{aligned}
&\mathit{taken} : (\mathit{WarpLane} \times \mathit{ProgAddr})^* \times \mathit{ProgAddr} \to \mathit{ActiveMask}\\
&\qquad \mathit{taken}(\varepsilon, pc) = \alpha^{\mathit{inact}}\\
&\mathit{taken}(wl\ pc \circ \overrightarrow{wp}, pc') = \begin{cases} \mathit{taken}(\overrightarrow{wp}, pc')[wl \mapsto \text{active}] & \text{if} \quad pc = pc' \\ \mathit{taken}(\overrightarrow{wp}, pc') & \text{otherwise} \end{cases}
\end{aligned}$$

On the other hand, the *notTaken* function computes the not-taken mask of a control flow instruction. Basically, the not-taken mask is the opposite of the taken mask except for threads that are already disabled. So we compute the not-taken mask by deactivating all threads in the current active mask that executed the control flow statement and that specified the target address chosen by the warp.

$$\begin{aligned}
&\mathit{notTaken} : \mathit{ActiveMask} \times (\mathit{WarpLane} \times \mathit{ProgAddr})^* \times \mathit{ProgAddr} \to \mathit{ActiveMask}\\
&\qquad \mathit{notTaken}(\alpha, \varepsilon, pc) = \alpha\\
&\mathit{notTaken}(\alpha, wl\ pc \circ \overrightarrow{wp}, pc') = \begin{cases} \mathit{notTaken}(\alpha, \overrightarrow{wp}, pc')[wl \mapsto \text{inactive}] & \text{if} \quad pc = pc' \\ \mathit{notTaken}(\alpha, \overrightarrow{wp}, pc') & \text{otherwise} \end{cases}
\end{aligned}$$

We define the operator $. \setminus .$ to deactivate all active threads in an active mask that are disabled according to the disable mask.

$$\begin{aligned}
&. \setminus . : \mathit{ActiveMask} \times \mathit{DisableMask} \to \mathit{ActiveMask}\\
&\alpha' = \alpha \setminus \delta \Leftrightarrow \forall wl \in \mathit{WarpLane} \,.\, \alpha'(wl) = \begin{cases} \text{inactive} & \text{if} \quad \delta(wl) \neq \text{enabled} \\ \alpha(wl) & \text{otherwise} \end{cases}
\end{aligned}$$

The *enable* function enables a thread in a disable mask if it is active in an active mask and the thread is disabled for the specified reason. It is imperative that a thread is not enabled as long as it is deactivated according to the active mask. Suppose a thread executed a return instruction, hence its disable mask is set to wait for all threads to return from the function. The other threads, however, call another function, hence an additional call token is pushed onto the stack. The thread is disabled because of the return, hence it is inactive in the call token's active mask. Once the call token is popped off the stack, the thread is not reactivated because it is inactive according to the token's active mask; it has to wait for the next call token to be popped off the stack before it is allowed to execute again. Without checking the token's active mask, the thread would have executed instructions of the same function it previously returned from.

$$
\begin{aligned}
&\mathit{enable} : \mathit{ActiveMask} \times \mathit{DisableMask} \times \mathit{DisableState} \to \mathit{DisableMask}\\
&\delta' = \mathit{enable}(\alpha, \delta, \mathit{disableState}) \Leftrightarrow\\
&\qquad \forall \mathit{wl} \in \mathit{WarpLane} \;.\; \delta'(\mathit{wl}) = \begin{cases} \text{enabled} & \text{if} \quad \alpha(\mathit{wl}) = \text{active} \wedge \delta(\mathit{wl}) = \mathit{disableState} \\ \delta(\mathit{wl}) & \text{otherwise} \end{cases}
\end{aligned}
$$

updExecState pops the topmost token off the stack. The warp's program counter is set to the address specified by the token. For call and break tokens, the disable mask is modified: All threads that waited for this token to be popped off the stack are reactivated. The warp's active mask is set to the token's active mask sans the threads that have been disabled in the meantime.

$$
\begin{aligned}
&\mathit{updExecState} : \mathit{ExecState} \to \mathit{ExecState}_\perp\\
&\qquad \mathit{updExecState}(\varsigma \triangleright \delta, (\text{sync}, \alpha, \mathit{pc}) :: \vec{\tau}) = \varsigma \triangleleft \mathit{pc}, \alpha \setminus \delta, \vec{\tau}\\
&\mathit{updExecState}(\varsigma \triangleright \delta, (\text{diverge}, \alpha, \mathit{pc}) :: \vec{\tau}) = \varsigma \triangleleft \mathit{pc}, \alpha \setminus \delta, \vec{\tau}\\
&\qquad\quad \mathit{updExecState}(\varsigma \triangleright \delta, (\text{call}, \alpha, \mathit{pc}) :: \vec{\tau}) = \varsigma \triangleleft \mathit{pc}, \alpha \setminus \delta', \delta', \vec{\tau}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if} \quad \delta' = \mathit{enable}(\alpha, \delta, \text{return})\\
&\quad \mathit{updExecState}(\varsigma \triangleright \delta, (\text{break}, \alpha, \mathit{pc}) :: \vec{\tau}) = \varsigma \triangleleft \mathit{pc}, \alpha \setminus \delta', \delta', \vec{\tau}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if} \quad \delta' = \mathit{enable}(\alpha, \delta, \text{break})\\
&\qquad\qquad\qquad\qquad \mathit{updExecState}(\varsigma) = \perp \qquad\qquad\qquad \text{otherwise}
\end{aligned}
$$

The function *unwind* unwinds the token stack until either at least one thread is active or the stack is empty. Unwinding guarantees that the warp always makes progress when it is scheduled. We also check for inconsistent token stack states: If the stack is empty but there is at least one thread waiting for a function return or the end of a loop, something has obviously gone wrong. The patent only lists threads waiting for a break as an error case. But as we assume all threads execute an `exit` as their last instruction, threads waiting for a return are also errors in our formalization. These error cases show that the branching algorithm only works correctly if the compiler injects the right control flow instructions at the right locations. The dependency on the behavior of the compiler also plays an important role in chapter 6 where we prove the correctness of the branching algorithm for terminating programs.

$$
\begin{aligned}
&\mathit{unwind} : \mathit{ExecState} \to (\mathit{WarpState} \times \mathit{ExecState})_\perp\\
&\mathit{unwind}(\varsigma \triangleright \mathit{pc}, \alpha, \delta, \varepsilon) = \begin{cases} (\text{completed}, \varsigma) & \text{if} \quad \forall \mathit{wl} \in \mathit{WarpLane} \;.\; \delta(\mathit{wl}) = \text{exit} \\ \perp & \text{otherwise} \end{cases}\\
&\quad \mathit{unwind}(\varsigma \triangleright \tau :: \vec{\tau}) = \begin{cases} (\text{active}, \varsigma') & \text{if} \quad \varsigma' \triangleright \alpha = \mathit{updExecState}(\varsigma) \wedge \\ & \qquad \exists \mathit{wl} \in \mathit{WarpLane} \;.\; \alpha(\mathit{wl}) = \text{active} \\ \mathit{unwind}(\varsigma \triangleleft \vec{\tau}) & \text{otherwise} \end{cases}
\end{aligned}
$$

We use transition system $\rightarrow^{\text{CRS}}$ to define the semantics of control flow instructions — the term CRS is established by the patent and stands for call, return, synchronization [26, 0059]. Actually, the transition system is just a function that maps an execution state and a set of control flow actions to a new execution state and a warp state. Nevertheless, we use a transition system instead of a function because it makes the case distinctions more obvious and it increases readability that would otherwise suffer because of space constraints.

$$\mathit{Configurations} : \langle \mathit{state}, \varsigma \rangle \in \mathit{WarpState} \times \mathit{ExecState}, \qquad \varsigma \in \mathit{ExecState}$$

$$\mathit{Transitions} : \varsigma \xrightarrow{\overrightarrow{\mathit{tact}_{\mathit{flow}}}}{}^{\text{CRS}} \langle \mathit{state}, \varsigma \rangle$$

A `ssy` instruction does not affect the warp's current active and disable masks but it pushes a sync token onto the stack. Execution continues with the instruction following the `ssy`. The token's active mask is the warp's current one and its program counter is the one specified by the instruction. All threads must specify the same program counter; different program counters make no sense and result in the warp semantics getting stuck. The program counter of a `ssy` instruction must be the address of the instruction following the corresponding `sync` instruction. This is different from the patent, where there are no `sync` instructions but rather sync flags that can be set on all instructions. In the patent, the program address of a `ssy` instruction is the address of the instruction with the corresponding sync flag set. There are two reasons why introducing an additional instruction for synchronization is superior to a flag: First, it allows $\rightarrow^{\text{CRS}}$ to handle all control flow actions, including `sync`, uniformly. Otherwise, we would have to check the sync flag separately in each rule of the warp semantics defined in section 5.4.3. More importantly though, there is a problem with sync flags the patent provides no workaround for. Consider the case where all threads executing the not-taken path of a conditional branch uniformly execute a return. Since no threads are active, the warp pops a token off the stack. Being inside the not-taken path, the topmost token that is popped is a sync token. According to the patent, the sync token contains the address of the instruction that should actually have been used to pop the token off the stack. Thus, the threads execute the instruction with the sync flag — provided not all threads of the taken path are disabled. The sync flag causes another token to be popped off the stack even though it should not; if the token is a call token, execution of the function is aborted before all threads executed all instructions they should execute. Interestingly, this is not a problem on the actual hardware. Test cases indicate that this situation is handled correctly. There are two possible explanations: Either, the patent is too inaccurate or the hardware somehow knows that the sync token was already popped off the stack and ignores the subsequent sync flag. The patent gives a hint that the GPU might indeed ignore sync flags after a sync token was popped, but the statement found in the patent is open for interpretation [26, 0108]. We avoid the issue altogether by introducing a dedicated `sync` instruction.

$$(\text{ssy}) \quad \langle \varsigma \triangleright pc, \alpha, \vec{\tau} \rangle \xrightarrow{\texttt{ssy}\ pc_1 \ldots \texttt{ssy}\ pc_n}{}^{\text{CRS}} \langle \text{active}, \varsigma \triangleleft pc + 1, (\text{sync}, \alpha, pc_1) :: \vec{\tau} \rangle$$
$$\text{if} \quad \forall i, j \in \{1 \ldots n\} \,.\, pc_i = pc_j$$

The `preBrk` and `preRet` instructions push a break or call token onto the stack, respectively. The token's active mask is set to the warp's current one and the program counter is set to the one provided by the instruction. As for `ssy`, all threads must agree on the program address. The active and disable masks remain unchanged.

$$\text{(preBrk)}\quad \langle \varsigma \blacktriangleright pc, \alpha, \vec{\tau} \rangle \xrightarrow{\texttt{preBrk}\ pc_1 \ldots \texttt{preBrk}\ pc_n}{}^{\text{CRS}} \langle \text{active}, \varsigma \blacktriangleleft pc + 1, (\text{break}, \alpha, pc_1) :: \vec{\tau} \rangle$$

$$\text{if} \quad \forall i, j \in \{1 \ldots n\} \, . \, pc_i = pc_j$$

$$\text{(preRet)}\quad \langle \varsigma \blacktriangleright pc, \alpha, \vec{\tau} \rangle \xrightarrow{\texttt{preRet}\ pc_1 \ldots \texttt{preRet}\ pc_n}{}^{\text{CRS}} \langle \text{active}, \varsigma \blacktriangleleft pc + 1, (\text{call}, \alpha, pc_1) :: \vec{\tau} \rangle$$

$$\text{if} \quad \forall i, j \in \{1 \ldots n\} \, . \, pc_i = pc_j$$

The sync instruction unwinds the token stack. If the compiler injected the control flow instructions correctly into the program, this either pops a divergence or a synchronization token from the stack. The patent uses a sync flag instead of a dedicated instruction. For the reasons outlined above, we prefer a dedicated instruction.

$$\text{(sync)}\quad \varsigma \xrightarrow{\texttt{sync} \ldots \texttt{sync}}{}^{\text{CRS}} \mathit{unwind}(\varsigma)$$

Breaks, returns, or exits are either uniform or divergent. In the divergent case, the threads that executed the instruction are disabled and their disable mask is set accordingly to indicate that the threads terminated or are waiting for a function call or a loop to complete. The remaining threads execute the instruction following the break, return, or exit.

$$\text{(brk}_{\text{div}}\text{)}\quad \langle \varsigma \blacktriangleright pc, \alpha, \delta \rangle \xrightarrow{\texttt{break}\ wl_1 \ldots \texttt{break}\ wl_n}{}^{\text{CRS}} \langle \text{active}, \varsigma \blacktriangleleft pc + 1, \alpha[wl_1 \ldots wl_n \mapsto \text{inactive}],$$

$$\delta[wl_1 \ldots wl_n \mapsto \text{break}] \rangle \quad \text{if} \quad \neg \mathit{uni}(\alpha, wl_1 \ldots wl_n)$$

$$\text{(ret}_{\text{div}}\text{)}\quad \langle \varsigma \blacktriangleright pc, \alpha, \delta \rangle \xrightarrow{\texttt{return}\ wl_1 \ldots \texttt{return}\ wl_n}{}^{\text{CRS}} \langle \text{active}, \varsigma \blacktriangleleft pc + 1, \alpha[wl_1 \ldots wl_n \mapsto \text{inactive}],$$

$$\delta[wl_1 \ldots wl_n \mapsto \text{return}] \rangle \quad \text{if} \quad \neg \mathit{uni}(\alpha, wl_1 \ldots wl_n)$$

$$\text{(exit}_{\text{div}}\text{)}\quad \langle \varsigma \blacktriangleright pc, \alpha, \delta \rangle \xrightarrow{\texttt{exit}\ wl_1 \ldots \texttt{exit}\ wl_n}{}^{\text{CRS}} \langle \text{active}, \varsigma \blacktriangleleft pc + 1, \alpha[wl_1 \ldots wl_n \mapsto \text{inactive}],$$

$$\delta[wl_1 \ldots wl_n \mapsto \text{exit}] \rangle \quad \text{if} \quad \neg \mathit{uni}(\alpha, wl_1 \ldots wl_n)$$

In the uniform case, there are no active threads left, hence the warp unwinds the stack. In contrast to the patent, however, we update the active mask and disable mask for all deactivated threads. While setting the active mask is just convenient for the proof later on — the active mask is overwritten anyway during the stack unwinding process —, not updating the disable mask is a mistake. For instance, consider the case that all threads executing the taken path of a conditional branch instruction uniformly execute a return. Because no active threads are left to execute, the diverge token is popped off the stack and some other threads execute the not-taken path. Once they are done, the sync token is popped off the stack. However, as we did not update the disable mask of the threads that executed the return, they are reactivated by the sync token and execute instructions they should not execute. This clearly is a mistake in the patent, however, test programs indicate that the hardware implementation is correct.

$$\text{(brk}_{\text{uni}}\text{)}\quad \langle \varsigma \blacktriangleright \alpha, \delta \rangle \xrightarrow{\texttt{break}\ wl_1 \ldots \texttt{break}\ wl_n}{}^{\text{CRS}} \mathit{unwind}(\varsigma \blacktriangleleft \alpha[wl_1 \ldots wl_n \mapsto \text{inactive}],$$

$$\delta[wl_1 \ldots wl_n \mapsto \text{break}]) \qquad \text{if} \quad \mathit{uni}(\alpha, wl_1 \ldots wl_n)$$

$$(\mathrm{ret}_{\mathrm{uni}}) \quad \langle \varsigma \triangleright \alpha, \delta \rangle \xrightarrow{\texttt{return}\ wl_1 \ldots \texttt{return}\ wl_n}{}^{\mathrm{CRS}} \mathit{unwind}(\varsigma \triangleleft \alpha[wl_1 \ldots wl_n \mapsto \text{inactive}], \delta[wl_1 \ldots wl_n \mapsto \text{return}]) \quad \text{if} \quad \mathit{uni}(\alpha, wl_1 \ldots wl_n)$$

$$(\mathrm{exit}_{\mathrm{uni}}) \quad \langle \varsigma \triangleright \alpha, \delta \rangle \xrightarrow{\texttt{exit}\ wl_1 \ldots \texttt{exit}\ wl_n}{}^{\mathrm{CRS}} \mathit{unwind}(\varsigma \triangleleft \alpha[wl_1 \ldots wl_n \mapsto \text{inactive}], \delta[wl_1 \ldots wl_n \mapsto \text{exit}]) \quad \text{if} \quad \mathit{uni}(\alpha, wl_1 \ldots wl_n)$$

Uniform branches or function calls do not change the active or disable masks and also do not push any tokens onto the stack. Only the warp's program counter is set to the target address of the branch or function call.

$$(\mathrm{bra}_{\mathrm{uni}}) \quad \langle \varsigma \triangleright pc, \alpha \rangle \xrightarrow{\texttt{bra}\ wl_1\ pc_1 \ldots \texttt{bra}\ wl_n\ pc_n}{}^{\mathrm{CRS}} \langle \text{active}, \varsigma \triangleleft pc_1 \rangle \quad \text{if} \quad \mathit{uni}(\alpha, wl_1 \ldots wl_n, pc_1 \ldots pc_n)$$

$$(\mathrm{call}_{\mathrm{uni}}) \quad \langle \varsigma \triangleright pc, \alpha \rangle \xrightarrow{\texttt{call}\ wl_1\ pc_1 \ldots \texttt{call}\ wl_n\ pc_n}{}^{\mathrm{CRS}} \langle \text{active}, \varsigma \triangleleft pc_1 \rangle \quad \text{if} \quad \mathit{uni}(\alpha, wl_1 \ldots wl_n, pc_1 \ldots pc_n)$$

The $(\mathrm{bra}_{\mathrm{guard}})$ rule is used for branch instructions that are divergent because of some guards evaluating to false but all specified target addresses are the same. The warp computes the taken mask and the not-taken mask and pushes a divergence token onto the stack. The token consists of the incremented program counter, i.e. the first address of the not-taken path, and the not-taken mask. The warp continues execution at the target address of the branch, activating only the threads that are active according to the taken mask.

$$(\mathrm{bra}_{\mathrm{guard}}) \quad \langle \varsigma \triangleright pc, \alpha, \vec{\tau} \rangle \xrightarrow{\texttt{bra}\ wl_1\ pc_1 \ldots \texttt{bra}\ wl_n\ pc_n}{}^{\mathrm{CRS}} \langle \text{active}, \varsigma \triangleleft pc_1, \mathit{taken}(wl_1\ pc_1 \ldots wl_n\ pc_n, pc_1), (\text{diverge}, \mathit{notTaken}(\alpha, wl_1\ pc_1 \ldots wl_n\ pc_n, pc_1), pc + 1) :: \vec{\tau} \rangle \quad \text{if} \quad \neg \mathit{uni}(\alpha, wl_1 \ldots wl_n) \wedge \forall i, j \in \{1 \ldots n\} \,.\, pc_i = pc_j$$

Similarly, the $(\mathrm{call}_{\mathrm{guard}})$ rule is used for function calls that are divergent because of some guards evaluating to false but all specified target addresses are the same. The warp computes the taken mask and lets the threads that are active according to the taken mask execute the function's instructions. Conditional function calls do not have a not-taken path as threads not participating in the call simply do not execute the function's instructions. Hence, no divergence token is pushed onto the stack in the case of a conditional call. In contrast to the patent, we require a call token be pushed onto the stack prior to all divergent and uniform function calls using the `preRet` instruction; the patent has the call instruction push a call token onto the stack itself. The call token ensures that execution resumes at the instruction following the call once all threads return from the function. We have to deviate from the patent because the patent does not allow conditional function calls, but our formalization does. If we did not deviate from the patent, the stack would become corrupted when a program uses indirect function calls, as it would be possible that there are two call tokens on the stack for the same function call. We can avoid this issue by either not having the call instruction push a call token onto the stack, or by not requiring that there is a `preRet` instruction before any indirect function call. We choose the former approach because the latter would make the `preRet` instruction redundant which would detach this formalization from the patent even further. In any case, it is imperative that the `preRet` instruction before a function call has the same guard as the function call itself; otherwise, a call token might be pushed onto a stack but the function it was intended for is never called, resulting in a corrupted stack.

$$
\begin{aligned}
&(\mathrm{call}_{\mathrm{guard}})\quad \langle \varsigma \triangleright \mathit{pc}, \alpha\rangle \xrightarrow{\texttt{call}\ \mathit{wl}_1\ \mathit{pc}_1 \ldots \texttt{call}\ \mathit{wl}_n\ \mathit{pc}_n}{}^{\mathrm{CRS}} \langle \text{active}, \varsigma \triangleleft \mathit{pc}_1, \mathit{taken}(\mathit{wl}_1\ \mathit{pc}_1 \ldots \mathit{wl}_n\ \mathit{pc}_n, \mathit{pc}_1)\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if}\quad \neg \mathit{uni}(\alpha, \mathit{wl}_1 \ldots \mathit{wl}_n) \wedge \forall i, j \in \{1 \ldots n\}\ .\ \mathit{pc}_i = \mathit{pc}_j
\end{aligned}
$$

Indirect branches work in a similar way to conditional branches. We randomly choose one of the target addresses — which one the hardware chooses is undefined — and use this address as the beginning of the taken path. All threads wishing to execute the taken path remain active, whereas the rest is deactivated. The diverge token pushed onto the stack comprises the not-taken mask and the address of the indirect branch instruction. Once the token is popped off the stack, the other threads get a chance to execute their designated path. Since the not-taken mask stored in the token and the taken mask used by the warp do not have any active threads in common, it is guaranteed that some other threads are allowed to continue executing the next time the indirect branch instruction is encountered. Indirect branches can also use guards to prevent some threads from jumping to another address; this is not handled by the ($\mathrm{bra}_{\mathrm{target}}$) rule. Eventually, the warp will return to the indirect branch instruction and only one target address will be left to execute. In that case, the ($\mathrm{bra}_{\mathrm{guard}}$) rule applies if there are any guards that evaluate to false. Otherwise, the branch is uniform and rule ($\mathrm{bra}_{\mathrm{uni}}$) is used to handle that case. Indirect function calls are handled in an analogous manner.

$$
\begin{aligned}
&(\mathrm{bra}_{\mathrm{target}})\quad \langle \varsigma \triangleright \mathit{pc}, \alpha, \vec{\tau}\rangle \xrightarrow{\texttt{bra}\ \mathit{wl}_1\ \mathit{pc}_1 \ldots \texttt{bra}\ \mathit{wl}_n\ \mathit{pc}_n}{}^{\mathrm{CRS}} \langle \text{active}, \varsigma \triangleleft \mathit{pc}_k, \mathit{taken}(\mathit{wl}_1\ \mathit{pc}_1 \ldots \mathit{wl}_n\ \mathit{pc}_n, \mathit{pc}_k), \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\text{diverge}, \mathit{notTaken}(\alpha, \mathit{wl}_1\ \mathit{pc}_1 \ldots \mathit{wl}_n\ \mathit{pc}_n, \mathit{pc}_k), \mathit{pc}) :: \vec{\tau}\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if}\quad \exists i, j \in \{1 \ldots n\}\ .\ \mathit{pc}_i \neq \mathit{pc}_j \wedge k \in \{1 \ldots n\}
\end{aligned}
$$

$$
\begin{aligned}
&(\mathrm{call}_{\mathrm{target}})\quad \langle \varsigma \triangleright \mathit{pc}, \alpha, \vec{\tau}\rangle \xrightarrow{\texttt{call}\ \mathit{wl}_1\ \mathit{pc}_1 \ldots \texttt{call}\ \mathit{wl}_n\ \mathit{pc}_n}{}^{\mathrm{CRS}} \langle \text{active}, \varsigma \triangleleft \mathit{pc}_k, \mathit{taken}(\mathit{wl}_1\ \mathit{pc}_1 \ldots \mathit{wl}_n\ \mathit{pc}_n, \mathit{pc}_k), \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\text{diverge}, \mathit{notTaken}(\alpha, \mathit{wl}_1\ \mathit{pc}_1 \ldots \mathit{wl}_n\ \mathit{pc}_n, \mathit{pc}_k), \mathit{pc}) :: \vec{\tau}\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if}\quad \exists i, j \in \{1 \ldots n\}\ .\ \mathit{pc}_i \neq \mathit{pc}_j \wedge k \in \{1 \ldots n\}
\end{aligned}
$$

If the threads generate no control flow actions, either the guards of all active threads evaluated to false or all threads executed an instruction that does not generate any control flow action. In that case, the program counter is incremented while the rest of the execution state remains untouched.

$$
(\mathrm{next})\quad \langle \varsigma \triangleright \mathit{pc}\rangle \xrightarrow{\varepsilon}{}^{\mathrm{CRS}} \langle \text{active}, \varsigma \triangleleft \mathit{pc} + 1\rangle
$$

5.4.2 Formalization of Votes and Barriers

Threads of the same warp can choose between four different vote modes to perform a reduction operation on the specified source predicates. The warp makes the result of the vote accessible to all participating threads. Only active threads participate in a vote; this is not a problem for the `.all`, `.any`, and `.uni` vote modes, but the value written to a thread's bit in the resulting data word is undefined for the `.ballot` mode. We assume that inactive threads are treated as if they voted false in `.ballot` mode.

The *vote* function computes the result of the vote based on the vote actions generated by the threads. The operation $d[n \mapsto_{bit} 1]$ changes the value of the n-th bit of a data word d to 1. The `.all` and `.any` vote modes are simply conjunctions or disjunctions over all votes cast, respectively. The `.uni` vote mode returns true if and only if the predicates of all threads participating in the vote have the same value. The `.ballot` vote mode returns a data word which contains a copy of all source predicates. The n-th bit corresponds to the predicate value of the thread at warp lane n.

$$
\begin{aligned}
&\mathit{vote} : \mathit{VoteMode} \times \mathit{VoteAct}^{*} \rightarrow \mathit{DataWord}\\
&\mathit{vote}(\texttt{.all}, \overrightarrow{\mathit{tact}_{vote}}) = \textstyle\bigwedge_{t \in \overrightarrow{\mathit{tact}_{vote}}} t_p\\
&\mathit{vote}(\texttt{.any}, \overrightarrow{\mathit{tact}_{vote}}) = \textstyle\bigvee_{t \in \overrightarrow{\mathit{tact}_{vote}}} t_p\\
&\mathit{vote}(\texttt{.uni}, \overrightarrow{\mathit{tact}_{vote}}) = \forall t_1, t_2 \in \overrightarrow{\mathit{tact}_{vote}} \,.\, t_{1,p} = t_{2,p}\\
&\mathit{vote}(\texttt{.ballot}, \varepsilon) = 0\\
&\mathit{vote}(\texttt{.ballot}, \mathit{tact}_{vote} \circ \overrightarrow{\mathit{tact}_{vote}}) = \begin{cases} \mathit{vote}(\texttt{.ballot}, \overrightarrow{\mathit{tact}_{vote}})[\mathit{tact}_{vote,wl} \mapsto_{bit} 1] & \text{if} \quad \mathit{tact}_{vote,p} \\ \mathit{vote}(\texttt{.ballot}, \overrightarrow{\mathit{tact}_{vote}}) & \text{otherwise} \end{cases}
\end{aligned}
$$

The *comm* function handles votes and barriers at the warp level. While votes are entirely processed at the warp level, thread barrier actions are coalesced and passed on to the warp's thread block. The function extracts the barrier indices the warp has to wait for, if any, and generates a warp action. For barriers, the warp's thread block updates its barrier states based on the information stored in the warp action. Votes, on the other hand, generate a warp memory action that contains a memory action for each participating thread. The memory action writes the result of the vote into the destination registers of the threads. We use _ to omit irrelevant parts of an instruction.

$$
\begin{aligned}
&\mathit{comm} : \mathit{Instr} \times \mathit{CommAct}_{\Theta}^{*} \rightarrow (\mathit{BarrierIdx}^{*} \times \mathit{Act}_{\Omega,\varepsilon})_{\perp}\\
&\mathit{comm}(\texttt{vote}\ \mathit{vm}\ \_, \overrightarrow{\mathit{tact}_{vote}}) = (\varepsilon, \textstyle\bigcup_{t \in \overrightarrow{\mathit{tact}_{vote}}} t_{regAddr} \leftarrow \mathit{vote}(\mathit{vm}, \overrightarrow{\mathit{tact}_{vote}})\ t_{memInf})\\
&\mathit{comm}(\texttt{bar.sync}\ \_, \overrightarrow{\mathit{tact}_{bar}}) = (\textstyle\bigcup_{t \in \overrightarrow{\mathit{tact}_{bar}}} t_{barid}, \texttt{bar.sync}\ \overrightarrow{\mathit{tact}_{bar}})\\
&\mathit{comm}(\texttt{bar.red}\ \mathit{br}\ \_, \overrightarrow{\mathit{tact}_{bar}}) = (\textstyle\bigcup_{t \in \overrightarrow{\mathit{tact}_{bar}}} t_{barid}, \texttt{bar.red}\ \mathit{br}\ \overrightarrow{\mathit{tact}_{bar}})\\
&\mathit{comm}(\texttt{bar.arrive}\ \_, \overrightarrow{\mathit{tact}_{bar}}) = (\varepsilon, \texttt{bar.arrive}\ \overrightarrow{\mathit{tact}_{bar}})\\
&\mathit{comm}(\mathit{instr}, \overrightarrow{\mathit{tact}_{comm}}) = \perp \qquad \text{otherwise}
\end{aligned}
$$

5.4.3 Warp Rules

Transition system $\rightarrow_{\Omega}$ defines the rules of the warp semantics. It optionally passes a warp action on to the warp's thread block.

$$
\begin{aligned}
&\mathit{Configurations} : \langle \omega \mid \phi, \varrho \rangle \in \mathit{Warp} \times \mathit{ProgEnv} \times \mathit{ThreadRegLookup}, \qquad \omega \in \mathit{Warp}\\
&\mathit{Transitions} : \langle \omega \mid \phi, \varrho \rangle \xrightarrow[\mathit{wact}_{\varepsilon}]{}^{\Omega} \omega'
\end{aligned}
$$

Each rule invokes the $\rightarrow^{\Theta_e}$ transition system for each thread. Whether a thread is allowed to execute the warp's current instruction, i.e. whether the thread is inactive according to the warp's active mask or whether the thread's guard evaluates to false, is checked at the thread level and is of no interest to the warp rules.

Threads generate three different types of thread actions: control flow actions, communication actions, or memory actions. As threads of the same warp execute a program in lockstep, we know that all threads generate the same type of thread actions during each step. This enables us to define a distinct rule in transition system $\rightarrow_\Omega$ for each of the three thread action types. Rule (bra) handles control flow actions. As those can be handled at the warp level, no warp action travels up the hierarchy. The warp uses transition system $\rightarrow^{\mathrm{CRS}}$ to update its execution state. The rules of $\rightarrow^{\mathrm{CRS}}$ might set the warp's state to completed provided that all threads have completed execution. Rule (bra) is also used when the threads do not generate any thread actions; in this case, rule (next) of transition system $\rightarrow^{\mathrm{CRS}}$ increments the program counter, so the threads execute the next instruction when the warp is scheduled again.

$$\text{(bra)}\quad \frac{\langle \theta_1 \mid \phi, pc, \alpha, \varrho\rangle \xrightarrow[\mathit{tact}_{\mathit{flow},1_\varepsilon}]{}{}^{\Theta_e} \theta_1' \quad \ldots \quad \langle \theta_n \mid \phi, pc, \alpha, \varrho\rangle \xrightarrow[\mathit{tact}_{\mathit{flow},n_\varepsilon}]{}{}^{\Theta_e} \theta_n'}{\langle \omega \triangleright (\varsigma \triangleright pc, \alpha), \theta_1 \ldots \theta_n \mid \phi, \varrho\rangle \rightarrow^{\Omega} \langle \omega \blacktriangleleft \mathit{state}', \varsigma', \theta_1' \ldots \theta_n'\rangle}$$

$$\text{if}\quad \varsigma \xrightarrow{\bigcup_{i\in\{1\ldots n\}} \mathit{tact}_{\mathit{flow},i_\varepsilon}}{}^{\mathrm{CRS}} \langle \mathit{state}', \varsigma'\rangle$$

The (comm) rule uses the *comm* function to process vote and barrier instructions. The warp action generated by the *comm* function is passed on to the warp's thread block. The rule also increments the warp's program counter; no special control flow handling is required.

$$\text{(comm)}\quad \frac{\langle \theta_1 \mid \phi, pc, \alpha, \varrho\rangle \xrightarrow[\mathit{tact}_{\mathit{comm},1_\varepsilon}]{}{}^{\Theta_e} \theta_1' \quad \ldots \quad \langle \theta_n \mid \phi, pc, \alpha, \varrho\rangle \xrightarrow[\mathit{tact}_{\mathit{comm},n_\varepsilon}]{}{}^{\Theta_e} \theta_n'}{\langle \omega \triangleright (\varsigma \triangleright pc, \alpha), \theta_1 \ldots \theta_n \mid \phi, \varrho\rangle \xrightarrow[\mathit{wact}_\varepsilon]{}{}^{\Omega} \langle \omega \blacktriangleleft (\varsigma \blacktriangleleft pc + 1), \overrightarrow{\mathit{barid}}, \theta_1' \ldots \theta_n'\rangle}$$

$$\begin{aligned} \text{if}\quad (\overrightarrow{\mathit{barid}}, \mathit{wact}_\varepsilon) &= \mathit{comm}(\phi_{\mathit{instrEnv}}(pc), \textstyle\bigcup_{i\in\{1\ldots n\}} \mathit{tact}_{\mathit{comm},i_\varepsilon}) \\ &\wedge \exists i \in \{1 \ldots n\} \;.\; \mathit{tact}_{\mathit{comm},i_\varepsilon} \neq \varepsilon \end{aligned}$$

When the threads execute a memory operation, the warp coalesces all of the threads' memory actions into a single warp memory action. The memory environment can use this grouping information to optimize memory accesses. Again, the warp's program counter is incremented as memory operations do not affect the control flow of the threads.

$$\text{(mem)}\quad \frac{\langle \theta_1 \mid \phi, pc, \alpha, \varrho\rangle \xrightarrow[\mathit{tact}_{\mathit{mem},1_\varepsilon}]{}{}^{\Theta_e} \theta_1' \quad \ldots \quad \langle \theta_n \mid \phi, pc, \alpha, \varrho\rangle \xrightarrow[\mathit{tact}_{\mathit{mem},n_\varepsilon}]{}{}^{\Theta_e} \theta_n'}{\langle \omega \triangleright (\varsigma \triangleright pc, \alpha), \theta_1 \ldots \theta_n \mid \phi, \varrho\rangle \xrightarrow[\bigcup_{i\in\{1\ldots n\}} \mathit{tact}_{\mathit{mem},i_\varepsilon}]{}{}^{\Omega} \langle \omega \blacktriangleleft (\varsigma \blacktriangleleft pc + 1), \theta_1' \ldots \theta_n'\rangle}$$

$$\text{if}\quad \exists i \in \{1 \ldots n\} \;.\; \mathit{tact}_{\mathit{mem},i_\varepsilon} \neq \varepsilon$$

5.5 Thread Block Semantics

When a kernel is launched on the GPU, the thread blocks that execute the kernel are not immediately created in the general case. A grid may consist of more threads than can concurrently be handled by the GPU's streaming multiprocessors. The Giga Thread scheduler checks whether any SMs have enough free resources to execute an additional thread block. If so, one of the available SMs is selected. The SM allocates the warps and threads, reserves the registers for each thread as well as the shared memory for the thread block. After a thread block has been assigned to a SM, it cannot be moved to another one. Furthermore, all threads and warps of a thread block are allocated immediately, hence the amount of threads and warps resident on the SM must not exceed the GPU's hardware limits. Additionally, there must be enough unassigned registers and a sufficient amount of shared memory to allocate the thread block. Once a warp is finished, its resources are freed. Another thread block may be assigned to an SM once a sufficient amount of resources has become available again, even when none of the SM's thread blocks are fully completed yet — provided that there is enough shared memory available. The lifetime of warps belonging to the same thread block can vary greatly; in an contrived program, one warp might execute forever, whereas all other warps terminate immediately.

A thread block can be in one of three states: It is either actively executed, completed, or it has not yet been scheduled, i.e. it has not yet been assigned to a streaming multiprocessor.

$$\mathit{BlockState} = \{\ \text{active},\ \text{unscheduled},\ \text{completed}\ \}$$

Like threads, thread blocks are uniquely identified by a three-dimensional index. Each dimension of the index is limited by the maximum grid size supported by the hardware.

$$\mathit{bid} \in \mathit{BlockIdx} = \prod_{\mathit{dim} \in \{x,y,z\}} \{0, \ldots, \mathit{MaxGridSize}_{\mathit{dim}} - 1\}$$

Within a thread block with warps $\vec{\omega}$, all threads have unique thread indices. Different thread blocks of the same grid, however, share all of their thread indices. Therefore, if a global thread index is required, the thread's thread block index must be taken into account.

$$\forall \theta_1, \theta_2 \in \{\theta \mid \omega \in \vec{\omega} \wedge \theta \in \omega_{\vec{\theta}}\}\ .\ \theta_1 \neq \theta_2 \Rightarrow \theta_{1,\mathit{tid}} \neq \theta_{2,\mathit{tid}}$$

A streaming multiprocessor might execute several thread blocks concurrently. Each of these thread blocks has access to a region of shared memory. However, when threads execute a shared memory load or store instruction on a shared memory variable, they all request the same address because shared memory addresses are assigned globally to the variables. Similar to the register offset, we assign a shared memory offset to each thread block to allow the threads to access the correct region of shared memory. Therefore, the thread block rules amend memory requests with the thread block's offset into shared memory.

$$\mathit{so} \in \mathit{SharedMemOffset} \subseteq \mathit{PhysMemAddr}$$

The memory environment also needs to know the index of the streaming multiprocessor executing the threads that issued a memory request. Thus, memory request generated by the scheduled warps are coalesced into a thread block action which stores the thread block's processor index and shared memory offset. Since barrier actions are handled at the thread block level, only memory requests continue to travel up the hierarchy.

$$\mathit{bact} \in \mathit{Act}_{\mathrm{B}} ::= \mathit{ProcIdx}\ \mathit{SharedMemOffset}\ \mathit{MemAct}^{*}_{\Omega}$$

A thread block comprises its block and processor indices, its shared memory offset, the thread block's current state, and its warps. Additionally, a thread block keeps track of its barriers using the domain of barrier environments defined in the next section.

$$\beta \in \mathit{ThreadBlock} = \mathit{BlockIdx} \times \mathit{ProcIdx} \times \mathit{BlockState} \times \mathit{BarrierEnv} \times \mathit{SharedMemOffset} \times \mathit{Warp}^{*}$$

In each dimension, the thread block index ranges between 0 and the grid size for all thread blocks $\beta \triangleright bid$ and all program configurations $\xi \triangleright gridSize$.

$$0 \leq bid_x < gridSize_x \land 0 \leq bid_y < gridSize_y \land 0 \leq bid_z < gridSize_z$$

5.5.1 Formalization of the Thread Block Synchronization Mechanism

Threads belonging to the same thread block can synchronize execution using the `bar` instruction. Thread execution continues once the specified number of threads reached the barrier. Barriers can optionally be used to perform a reduction operation across threads. In this section, we formalize the domain of barriers and the operations used by the rules of the thread block semantics to initialize, update, and complete them. However, as the PTX specification only gives a rough outline, there are a lot of assumptions we have to make in this section.

For the sake of completeness, we repeat the definitions of the domains of barrier indices and thread counts below. Additionally, we introduce the domain of arrival counts. The thread count specifies how many threads have to reach a barrier before it completes, whereas the arrival count denotes the amount of threads that have already reached a barrier.

$$\begin{aligned} barid &\in BarrierIdx = \{0, \ldots, NumBarrierIndices - 1\} \\ tc &\in ThreadCnt = \{n \in \mathbb{N} \mid n \bmod WarpSize = 0\} \\ ac &\in ArrCnt \subseteq ThreadCnt \end{aligned}$$

The domain of barriers therefore stores both the thread and the arrival count. Optionally, a barrier reduction operation might have to be carried out. Hence the domain stores the reduction type and the thread barrier actions that define the destination registers as well as the input values for the reduction.

$$bar \in Barrier = ThreadCnt \times ArrCnt \times BarrierRed_\varepsilon \times BarrierAct^*_\Theta$$

The barrier environment maps a barrier index to a barrier. The value ε denotes a barrier index that is currently not in use, i.e. no barrier with the given index has been initiated or the barrier has already been completed. A program can reuse the same barrier index as often as necessary.

$$\chi \in BarrierEnv = BarrierIdx \to Barrier_\varepsilon$$

We define the function *updBar* that is responsible for initializing and updating barriers when the threads execute a barrier instruction. Before the function updates the barrier environment, however, it performs several consistency checks. First, warp barrier actions are converted into elements of the domain of barrier information. A barrier information element consists of the barrier index, the barrier's thread count as specified by the threads, the barrier type, and the reduction operation.

$$i \in BarInfo = BarrierIdx \times ThreadCnt_\varepsilon \times BarrierType \times BarrierRed_\varepsilon$$

During the conversion process, we perform several consistency checks. There might be thread barrier actions that contradict each another; for instance, one thread uses some barrier without a reduction operation, another thread uses the same barrier with a reduction operation. The PTX documentation clearly specifies that mixing `bar.red` with either `bar.arrive` or `bar.sync` results in unpredictable behavior. On the other hand, it does not state what happens if different thread counts are specified. Most likely, one of the specified thread counts is chosen randomly; but since randomly selecting some thread count also results in unpredictable behavior, we simply do not support inconsistent barrier information. The *barInfo* function converts warp barrier actions into barrier information elements. It goes recursively through all generated warp actions and their contained thread actions. If it encounters any contradictions, it returns ⊥. Barrier information is not duplicated, i.e. the list of barrier information elements returned by *barInfo* contains each barrier index once, at most.

$$
\begin{aligned}
&\mathit{barInfo} : \mathit{BarrierAct}^*_\Omega \to (\mathit{BarInfo}^*)_\bot \\
&\mathit{barInfo}(\varepsilon) = \varepsilon \\
&\mathit{barInfo}((\mathit{wact}_{bar} \triangleright \mathit{bt}, \mathit{br}_\varepsilon, \varepsilon) \circ \overrightarrow{\mathit{wact}_{bar}}) = \mathit{barInfo}(\overrightarrow{\mathit{wact}_{bar}}) \\
&\mathit{barInfo}((\mathit{wact}_{bar} \triangleright \mathit{bt}, \mathit{br}_\varepsilon, (\mathit{tact}_{bar} \triangleright \mathit{barid}, \mathit{tc}_\varepsilon) \circ \overrightarrow{\mathit{tact}_{bar}}) \circ \overrightarrow{\mathit{wact}_{bar}}) = \\
&\qquad\qquad\qquad\qquad \begin{cases} \vec{i} & \text{if} \quad \forall i' \in \vec{i}\,.\, i'_{barid} = \mathit{barid} \Rightarrow i = i' \\ i \circ \vec{i} & \text{if} \quad \forall i' \in \vec{i}\,.\, i'_{barid} \neq \mathit{barid} \\ \bot & \text{otherwise} \end{cases} \\
&\qquad\qquad \text{where} \quad i = (\mathit{barid}, \mathit{tc}_\varepsilon, \mathit{bt}, \mathit{br}_\varepsilon) \wedge \vec{i} = \mathit{barInfo}((\mathit{wact}_{bar} \triangleleft \overrightarrow{\mathit{tact}_{bar}}) \circ \overrightarrow{\mathit{wact}_{bar}})
\end{aligned}
$$

If the generated barrier actions are consistent, the function *prepBar* initializes uninitialized barriers referenced by the barrier actions and performs consistency checks on already initialized barriers. For uninitialized barriers, the thread count is either set to the value stored in the barrier information element or to the thread count passed to the function depending on whether the threads specify a thread count. The value passed to the function is the number of threads of the thread block. For already initialized barriers, another consistency check is performed because the barrier could have been initialized with data that contradicts the data stored in the barrier information elements.

$$
\begin{aligned}
&\mathit{prepBar} : \mathit{BarrierEnv} \times \mathit{ThreadCnt} \times \mathit{BarInfo}^* \to \mathit{BarrierEnv}_\bot \\
&\mathit{prepBar}(\chi, \mathit{tc}, \varepsilon) = \chi \\
&\mathit{prepBar}(\chi, \mathit{tc}, (i \triangleright \mathit{barid}, \mathit{tc}'_\varepsilon, \mathit{bt}, \mathit{br}_\varepsilon) \circ \vec{i}) = \\
&\quad \begin{cases} \mathit{prepBar}(\chi[\mathit{barid} \mapsto (\mathit{tc}'', 0, \mathit{br}_\varepsilon, \varepsilon)], \mathit{tc}, \vec{i}) & \text{if} \ \ \chi(\mathit{barid}) = \varepsilon \wedge \mathit{tc}'' = \begin{cases} \mathit{tc}'_\varepsilon & \text{if} \quad \mathit{tc}'_\varepsilon \neq \varepsilon \\ \mathit{tc} & \text{otherwise} \end{cases} \\ \mathit{prepBar}(\chi, \mathit{tc}, \vec{i}) & \text{if} \ \ (\mathit{bar} \triangleright \mathit{tc}', \mathit{br}_\varepsilon) = \chi(\mathit{barid}) \\ & \qquad \wedge (\mathit{tc}'_\varepsilon \neq \varepsilon \Rightarrow \mathit{tc}'_\varepsilon = \mathit{tc}') \\ & \qquad \wedge (\mathit{br}_\varepsilon \neq \varepsilon \Rightarrow \mathit{bt} = \texttt{red}) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Next, the newly generated thread actions are copied into the barrier environment. When a barrier completes, it uses the thread actions to generate the memory operations that store the result of the optional reduction operation. The specification does not state what happens if a barrier with a reduction operation completes but some threads did not execute a corresponding `bar.red` instruction because they were deactivated according to their warps' active masks. We assume that they are ignored during the reduction. The function *updAct* goes recursively through all generated warp actions and their contained thread actions and copies the thread actions into the barrier environment.

$$
\begin{aligned}
&\mathit{updAct} : \mathit{BarrierEnv} \times \mathit{BarrierAct}^{*}_{\Omega} \rightarrow \mathit{BarrierEnv}\\
&\mathit{updAct}(\chi, \varepsilon) = \chi\\
&\mathit{updAct}(\chi, (\mathit{wact}_{\mathit{bar}} \triangleright \mathit{bt}, \mathit{br}_{\varepsilon}, \varepsilon) \circ \overrightarrow{\mathit{wact}_{\mathit{bar}}}) = \mathit{updAct}(\chi, \overrightarrow{\mathit{wact}_{\mathit{bar}}})\\
&\mathit{updAct}(\chi, (\mathit{wact}_{\mathit{bar}} \triangleright \mathit{bt}, \mathit{br}_{\varepsilon}, (\mathit{tact}_{\mathit{bar}} \triangleright \mathit{barid}) \circ \overrightarrow{\mathit{tact}_{\mathit{bar}}}) \circ \overrightarrow{\mathit{wact}_{\mathit{bar}}}) =\\
&\qquad\qquad \mathit{updAct}(\chi[\mathit{barid} \mapsto \chi(\mathit{barid}) \triangleleft \mathit{tact}_{\mathit{bar}} \circ \chi(\mathit{barid})_{\overrightarrow{\mathit{tact}_{\mathit{bar}}}}], (\mathit{wact}_{\mathit{bar}} \triangleleft \overrightarrow{\mathit{tact}_{\mathit{bar}}}) \circ \overrightarrow{\mathit{wact}_{\mathit{bar}}})
\end{aligned}
$$

Last, the arrival counts of all barriers referenced by at least one thread action are incremented. Barriers are incremented with warp size granularity, i.e. if a thread executes a barrier instruction it is as if all threads of the same warp executed the barrier instruction, even those that are currently inactive because of the warp's active mask or because the guard evaluated to false. The function *updArrCnt* increments the arrival counts of all barriers referenced by at least one thread action; if n warps reference a barrier, its arrival count is incremented by n times the warp size.

$$
\begin{aligned}
&\mathit{updArrCnt} : \mathit{BarrierEnv} \times \mathit{BarrierAct}^{*}_{\Omega} \rightarrow \mathit{BarrierEnv}\\
&\mathit{updArrCnt}(\chi, \varepsilon) = \chi\\
&\mathit{updArrCnt}(\chi, \mathit{wact}_{\mathit{bar}} \circ \overrightarrow{\mathit{wact}_{\mathit{bar}}}) = \mathit{updArrCnt}(\chi[\mathit{barid}_1 \ldots \mathit{barid}_n \mapsto\\
&\qquad \chi(\mathit{barid}_1) \triangleleft \chi(\mathit{barid}_1)_{\mathit{ac}} + \mathit{WarpSize} \ldots \chi(\mathit{barid}_n) \triangleleft \chi(\mathit{barid}_n)_{\mathit{ac}} + \mathit{WarpSize}], \overrightarrow{\mathit{wact}_{\mathit{bar}}})\\
&\qquad\qquad \text{where} \quad \mathit{barid}_1 \ldots \mathit{barid}_n \in \{\mathit{tact}_{\mathit{bar},\mathit{barid}} \mid \mathit{tact}_{\mathit{bar}} \in \mathit{wact}_{\mathit{bar},\overrightarrow{\mathit{tact}_{\mathit{bar}}}}\}
\end{aligned}
$$

The *updBar* function assembles the four functions defined above. Both *barInfo* and *prepBar* might return $\perp$; however, $\perp$ can be assigned to neither $\vec{i}$ nor χ'. Hence, *updBar* returns $\perp$ if there are any inconsistencies found in the generated thread actions.

$$
\begin{aligned}
&\mathit{updBar} : \mathit{BarrierEnv} \times \mathit{ThreadCnt} \times \mathit{BarrierAct}^{*}_{\Omega} \rightarrow \mathit{BarrierEnv}_{\perp}\\
&\mathit{updBar}(\chi, \mathit{tc}, \overrightarrow{\mathit{wact}_{\mathit{bar}}}) = \begin{cases} \mathit{updArrCnt}(\mathit{updAct}(\chi', \overrightarrow{\mathit{wact}_{\mathit{bar}}}), \overrightarrow{\mathit{wact}_{\mathit{bar}}}) & \\ \qquad\qquad \text{if} \quad \vec{i} = \mathit{barInfo}(\overrightarrow{\mathit{wact}_{\mathit{bar}}}) \land \chi' = \mathit{prepBar}(\chi, \mathit{tc}, \vec{i}) & \\ \perp \qquad\qquad \text{otherwise} & \end{cases}
\end{aligned}
$$

The function *reduce* performs a barrier reduction operation. Supported reduction operations are `.and`, `.or`, and `.popc`. The former two operations perform a conjunction or disjunction over all source predicates, respectively. The latter operation returns the population count, i.e. the number of source predicates that are true. As mentioned above, threads that did not execute a corresponding `bar.red` instruction because they were inactive for some reason are ignored during the computation of the reduction. This is just an assumption, however, as the actual behavior is unspecified.

$$
\begin{aligned}
\mathit{reduce} &: \mathit{BarrierRed} \times \mathit{BarrierAct}^{*}_{\Theta} \rightarrow \mathit{DataWord}\\
\mathit{reduce}(\texttt{.and}, \overrightarrow{\mathit{tact}_{\mathit{bar}}}) &= \textstyle\bigwedge_{t \in \overrightarrow{\mathit{tact}_{\mathit{bar}}}} t_p\\
\mathit{reduce}(\texttt{.or}, \overrightarrow{\mathit{tact}_{\mathit{bar}}}) &= \textstyle\bigvee_{t \in \overrightarrow{\mathit{tact}_{\mathit{bar}}}} t_p\\
\mathit{reduce}(\texttt{.popc}, \overrightarrow{\mathit{tact}_{\mathit{bar}}}) &= \mid \{t \in \overrightarrow{\mathit{tact}_{\mathit{bar}}} \mid t_p\} \mid
\end{aligned}
$$

The *barAct* function generates a memory request to store the result of a reduction operation in the destination registers of all participating threads. Block actions consist of warp memory actions that in turn consist of thread memory actions. Therefore, the function has to group the barrier's thread actions into warps. The memory action for a thread is then added to the appropriate warp memory action.

$$\begin{aligned}
&\mathit{barAct} : \mathit{Barrier} \times \mathit{ProcIdx} \times \mathit{SharedMemOffset} \times \mathit{Warp}^{*} \rightarrow \mathit{Act}_{\mathrm{B},\varepsilon} \\
&\mathit{barAct}((\mathit{bar} \triangleright \mathit{br}_{\varepsilon}, \overrightarrow{\mathit{tact}_{\mathit{bar}}}), \mathit{pid}, \mathit{so}, \vec{\omega}) = \\
&\quad \begin{cases}
\mathit{pid}\ \mathit{so}\ \bigcup_{\omega \in \vec{\omega}} \bigcup_{(\mathit{tact}_{\mathit{bar}} \triangleright \mathit{regAddr}, \mathit{memInf}) \in \overrightarrow{\mathit{tact}_{\mathit{bar}}} \wedge \mathit{memInf}_{\mathit{tid}} \in \{\theta_{\mathit{tid}} \mid \theta \in \omega_{\vec{\theta}}\}} & \\
\qquad\qquad\qquad \mathit{regAddr} \leftarrow \mathit{reduce}(\mathit{br}_{\varepsilon}, \overrightarrow{\mathit{tact}_{\mathit{bar}}})\ \mathit{memInf} & \text{if} \quad \mathit{br}_{\varepsilon} \neq \varepsilon \\
\varepsilon & \text{otherwise}
\end{cases}
\end{aligned}$$

The function *remBar* removes a completed barrier from the warps' lists of barrier indices. If a warp does not wait for other barriers to complete, it can be scheduled again in a future clock cycle. A warp cannot be scheduled if its list of barrier indices is not empty. As soon as some threads of a warp execute a barrier instruction, the barrier indices stored in the thread actions are added to the warp's list of barrier indices. Since the barrier index can be specified by either a constant or by a register, it is possible that all threads, including the inactive ones, of a warp reach several barriers simultaneously and the warp cannot be scheduled until all barriers have completed. We do not know whether this is actually supported by the hardware as the documentation does not mention this situation.

$$\begin{aligned}
&\mathit{remBar} : \mathit{BarrierIdx} \times \mathit{Warp}^{*} \rightarrow \mathit{Warp}^{*} \\
&\qquad\qquad \mathit{remBar}(\mathit{barid}, \varepsilon) = \varepsilon \\
&\mathit{remBar}(\mathit{barid}, (\omega \triangleright \overrightarrow{\mathit{barid}}) \circ \vec{\omega}) = (\omega \triangleleft \overrightarrow{\mathit{barid}} \setminus \mathit{barid}) \circ \mathit{remBar}(\mathit{barid}, \vec{\omega})
\end{aligned}$$

endBar checks whether the arrival count of some barrier is greater or equal to the barrier's thread count. If so, the barrier is completed and the barrier is reset to ε in the barrier environment. Additionally, the barrier is removed from the list of barrier indices of all warps of the thread block. Furthermore, if the barrier should perform a reduction operation on completion, a block action is generated that updates the destination registers of all participating threads.

$$\begin{aligned}
&\mathit{endBar} : \mathit{BarrierEnv} \times \mathit{ProcIdx} \times \mathit{SharedMemOffset} \times \mathit{Warp}^{*} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathit{BarrierEnv} \times \mathit{Act}_{\mathrm{B},\varepsilon} \times \mathit{Warp}^{*} \\
&\mathit{endBar}(\chi, \mathit{pid}, \mathit{so}, \vec{\omega}) = (\chi[\mathit{barid} \mapsto \varepsilon], \mathit{barAct}(\chi(\mathit{barid}), \mathit{pid}, \mathit{so}, \vec{\omega}), \mathit{remBar}(\mathit{barid}, \vec{\omega})) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{if} \quad (\mathit{bar} \triangleright \mathit{tc}, \mathit{ac}) = \chi(\mathit{barid}) \wedge \mathit{ac} \geq \mathit{tc}
\end{aligned}$$

Apparently, when a warp completes program execution, it has implicitly arrived at all barriers. This behavior is not documented, but test programs indicate that warps waiting for a barrier are allowed to continue as soon as warps that never reached the barrier complete. Our formalization does not reflect this behavior. The CUDA specification clearly warns about using barrier instructions that are not reached by all warps or by all threads of a warp.

5.5.2 Thread Block Rules

Thread block actions add the thread block's processor index and shared memory offset to memory actions generated by the block's threads. Conversely, the semantics of thread blocks passes these two values to a thread block register look up function in order to obtain a thread register look up function that can be passed down to the warps.

$$\varrho \in \mathit{BlockRegLookup} = \mathit{BlockIdx} \times \mathit{ProcIdx} \to \mathit{ThreadRegLookup}$$

Typically, a thread block consists of more warps than can be concurrently executed on the thread block's streaming multiprocessor. But as there can be more than one thread block resident on an SM, warp execution cannot be scheduled at the thread block level. The thread block rules are therefore provided with a warp schedule that determines which of the thread block's warps execute a step. The other warps have to remain idle.

$$s_\Omega \in \mathit{WarpSchedule} = \mathit{Warp}^*$$

Transition system $\to^{\mathrm{B}}$ defines the semantics of thread blocks. Given the program environment, a thread block register look up function and a warp schedule, the thread block executes a step, possibly emitting several thread block actions.

$$\begin{aligned} \mathit{Configurations} &: \langle \beta \mid \phi, \varrho, s_\Omega \rangle \in \mathit{ThreadBlock} \times \mathit{ProgEnv} \times \mathit{BlockRegLookup} \times \mathit{WarpSchedule}, \\ &\quad \beta \in \mathit{ThreadBlock} \\ \mathit{Transitions} &: \langle \beta \mid \phi, \varrho, s_\Omega \rangle \xrightarrow[\overrightarrow{bact}]{}{}^{\mathrm{B}} \beta' \end{aligned}$$

The (exec) rule executes a step of an active thread block. The scheduled warps are executed and the generated warp actions are coalesced into a thread block action or used to update the state of the barrier environment. Thread block actions containing an empty set of warp memory actions are later ignored at the device level. Additionally, the default thread count for newly initialized barriers is calculated and passed to the *updBar* function. The default thread count is the number of active warps times the warp size; this ensures that the barrier is not waiting for warps to reach the barrier that have already been completed before the barrier is initialized.

$$(\text{exec}) \quad \frac{\left(\langle \omega_i \mid \phi, \varrho(\mathit{bid}, \mathit{pid}) \rangle \xrightarrow[\mathit{wact}_{\omega_\varepsilon}]{}{}^{\Omega} \omega'_i\right)_{\omega_i \in s_\Omega}}{\langle \beta \triangleright \mathit{bid}, \mathit{pid}, \text{active}, \chi, \mathit{so}, \omega_1 \ldots \omega_n \mid \phi, \varrho, s_\Omega \rangle \xrightarrow[\mathit{pid}\ \mathit{so}\ \bigcup_{\omega \in s_\Omega} \mathit{wact}_{\mathit{mem},\omega_\varepsilon}]{}{}^{\mathrm{B}} \langle \beta \triangleleft \chi', \omega'_1 \ldots \omega'_n \rangle}$$

$$\text{if} \quad \forall \omega_i \in \omega_1 \ldots \omega_n \setminus s_\Omega \,.\, \omega'_i = \omega_i \wedge \chi' = \mathit{updBar}(\chi, \mathit{tc}, \textstyle\bigcup_{\omega \in s_\Omega} \mathit{wact}_{\mathit{bar},\omega_\varepsilon})$$

$$\text{where} \quad \mathit{tc} = |\{(\omega \triangleright \text{active}) \in \omega_1 \ldots \omega_n\}| \cdot \mathit{WarpSize}$$

If there are any completed barriers, the (endBar) rule ends a single barrier after the thread block executed a step. The (endBar) rule can be applied several times during a single step of a thread block, hence several thread block actions might be output by transition system $\to^{\mathrm{B}}$. We ensure that only barriers of active thread blocks can be completed to avoid writing the result of a reduction operation into registers that might already be used by threads of another thread block.

$$\text{(endBar)}\quad \frac{\langle \beta \mid \phi, \varrho, s_\Omega \rangle \xrightarrow[\overrightarrow{bact}]{}^{\mathrm{B}} \langle \beta' \triangleright \text{active}, \chi, \omega_1 \dots \omega_n \rangle}{\langle \beta \triangleright pid, so \mid \phi, \varrho, s_\Omega \rangle \xrightarrow[bact_\varepsilon \circ \overrightarrow{bact}]{}^{\mathrm{B}} \langle \beta' \triangleleft \chi', \omega'_1 \dots \omega'_n \rangle}$$

$$\text{if} \quad (\chi', bact_\varepsilon, \omega'_1 \dots \omega'_n) = endBar(\chi, pid, so, \omega_1 \dots \omega_n) \wedge \chi \neq \chi'$$

After a thread block executed a step, it is checked whether all of the thread block's warps have finished executing the program, i.e. whether their state is set to completed. If so, the thread block's state is set to completed. The resources occupied by the thread block become available again and the Giga Thread scheduler might be able to issue another thread block to the streaming multiprocessor.

$$\text{(compl)}\quad \frac{\langle \beta \mid \phi, \varrho, s_\Omega \rangle \xrightarrow[\overrightarrow{bact}]{}^{\mathrm{B}} \langle \beta' \triangleright \text{active}, \omega_1 \dots \omega_n \rangle}{\langle \beta \mid \phi, \varrho, s_\Omega \rangle \xrightarrow[\overrightarrow{bact}]{}^{\mathrm{B}} \langle \beta' \triangleleft \text{completed} \rangle} \quad \text{if} \quad \forall i \in \{1 \dots n\} \,.\, \omega_{i,state} = \text{completed}$$

Completed and unscheduled thread blocks have to remain idle. This rule is required because grids use transition system $\rightarrow^{\mathrm{B}}$ to advance all of their thread blocks. What a thread block does is decided at the thread block level; for example, warps are executed if there are any scheduled ones and barriers might be completed. However, neither completed nor unscheduled thread blocks are allowed to do any of those things; they have to remain idle whenever their grid performs a step. Without the (idle) rule, the grid semantics would be stuck as soon as there are any unscheduled or completed thread blocks.

$$\text{(idle)} \quad \langle \beta \triangleright blockState \mid \phi, \varrho, s_\Omega \rangle \rightarrow^{\mathrm{B}} \beta \qquad \text{if} \quad blockState \neq \text{active}$$

5.6 Grid Semantics

When a kernel is launched on the GPU, a grid comprising several thread blocks is created. We assign an abstract but unique grid index to each grid.

$$gid \in GridIdx$$

Grids can be either active or completed. Once a grid has completed execution of the kernel, the GPU reports the grid completion to the graphics driver; for the driver, the completion of a grid might be a synchronization event. Driver notification is handled at the context and device level however.

$$GridState = \{\ \text{active},\ \text{completed}\ \}$$

The domain of grids stores the grid's state and index, as well as the program configuration used to execute the program. Among other things, the program configuration selects the entry point of the PTX program to execute, i.e. the first instruction each thread processes, and it determines the number of thread blocks and threads created to execute the kernel.

$$\gamma \in Grid = GridState \times GridIdx \times ProgConf \times ThreadBlock^*$$

Warp scheduling cannot be performed at the grid level, because a streaming multiprocessor might concurrently execute thread blocks and warps of different grids. Therefore, a block schedule is given to the rules of the grid semantics. The block schedule determines for each thread block which of its warps should perform a step.

$$s_{\mathrm{B}} \in \mathit{BlockSchedule} = \mathit{ThreadBlock} \rightarrow \mathit{Warp}^{*}$$

The rules of the grid semantics patch through all generated thread block actions unchanged. However, they are given a grid register look up function that they have to use to obtain a thread block register look up function.

$$\varrho \in \mathit{GridRegLookup} = \mathit{GridSize} \times \mathit{BlockSize} \rightarrow \mathit{BlockRegLookup}$$

Within a grid with thread blocks $\vec{\beta}$, all thread blocks have unique thread block indices. Different grids of the same context, however, share some or all of their thread block indices depending on the dimensions of the grids.

$$\forall \beta_1, \beta_2 \in \vec{\beta} \,.\, \beta_1 \neq \beta_2 \Rightarrow \beta_{1,bid} \neq \beta_{2,bid}$$

Transition system $\rightarrow^{\Gamma}$ defines the semantics of grids. It coalesces the thread block actions output by its thread blocks but does not change them otherwise.

$$\begin{aligned} \mathit{Configurations} : {} & \langle \gamma \mid \phi, \varrho, s_{\mathrm{B}} \rangle \in \mathit{Grid} \times \mathit{ProgEnv} \times \mathit{GridRegLookup} \times \mathit{BlockSchedule}, \\ & \gamma \in \mathit{Grid} \\ \mathit{Transitions} : {} & \langle \gamma \mid \phi, \varrho, s_{\mathrm{B}} \rangle \xrightarrow[\overrightarrow{bact}]{}^{\Gamma} \gamma' \end{aligned}$$

The (exec) rule uses transition system $\rightarrow^{\mathrm{B}}$ to process the thread blocks. Whether a thread block actually does anything is decided at the thread block level. Unscheduled and completed thread blocks do not perform any actions, whereas active thread blocks are allowed to end a barrier and to execute some of their warps in accordance with the warp schedule obtained from the block schedule.

$$(\text{exec}) \quad \frac{\langle \beta_1 \mid \phi, \varrho', s_{\mathrm{B}}(\beta_1) \rangle \xrightarrow[\overrightarrow{bact_1}]{}^{\mathrm{B}} \beta_1' \quad \dots \quad \langle \beta_n \mid \phi, \varrho', s_{\mathrm{B}}(\beta_n) \rangle \xrightarrow[\overrightarrow{bact_n}]{}^{\mathrm{B}} \beta_n'}{\langle \gamma \triangleright \xi, \beta_1 \dots \beta_n \mid \phi, \varrho, s_{\mathrm{B}} \rangle \xrightarrow[\bigcup_{i \in \{1 \dots n\}} \overrightarrow{bact_i}]{}^{\Gamma} \langle \gamma \triangleleft \beta_1' \dots \beta_n' \rangle}$$

$$\text{where} \quad \varrho' = \varrho(\xi_{gridSize}, \xi_{blockSize})$$

Once all of a grid's thread blocks have completed execution, the grid's state is set to completed. The context subsequently removes the grid from its list of grids and informs the graphics driver about the completion of the kernel launch. We ensure that rule (compl) can only be applied once by requiring that the grid is in its active state before it is marked completed.

$$(\text{compl}) \quad \frac{\langle \gamma \mid \phi, \varrho, s_{\mathrm{B}} \rangle \xrightarrow[\overrightarrow{bact}]{}^{\Gamma} \langle \gamma' \triangleright \text{active}, \beta_1 \dots \beta_n \rangle}{\langle \gamma \mid \phi, \varrho, s_{\mathrm{B}} \rangle \xrightarrow[\overrightarrow{bact}]{}^{\Gamma} \langle \gamma' \triangleleft \text{completed} \rangle} \quad \text{if} \quad \forall i \in \{1 \dots n\} \,.\, \beta_{i,blockState} = \text{completed}$$

5.7 Context Semantics

Contexts allow the host program to split an algorithm into several kernels that share global memory and launch those kernels in a fixed order. This approach works around the limitations of CUDA's thread synchronization capabilities. As CUDA does not support global thread synchronization, splitting an algorithm into several kernels emulates that behavior. The performance cost might be relatively high though, because kernel launches are a slow operation. But there is no other way to synchronize access to global memory for threads of different thread blocks.

A PTX program potentially defines several entry points. The program configuration of a grid determines which entry point the threads execute. A grid therefore always executes one entry point of one PTX program. A context, on the other hand, manages several PTX programs. After a program has been loaded into a context, the memory used by the program is reserved and the host program can launch a grid of thread blocks to execute one of the program's entry points. Additionally, the host program might abort grid execution at any time.

Kernels launched within the same context share a virtual address space, hence they are able to access the same memory locations. We define a function to map virtual memory addresses to physical memory addresses. The function is amended to the thread block actions generated by the context's active thread blocks so that the memory environment is able to convert a virtual address to a physical one.

$$\begin{aligned} vms \in VirtMemSpace &= VirtMemAddr \rightarrow PhysMemAddr_{\perp} \\ cact \in Act_{\mathrm{C}} &= VirtMemSpace\ Act_{\mathrm{B}}^{*} \end{aligned}$$

Again, we use unique indices to identify contexts. For this reason, we introduce the abstract domain of context indices.

$$cid \in ContextIdx$$

A context manages a list of program environments. As the program environment executed by a grid is not stored at the grid level, we need a way figure out which program environment to pass to the grid semantics. The program map lets us do that, as it maps grid indices to their corresponding program environment indices.

$$pm \in ProgMap = GridIdx \rightarrow ProgIdx_{\perp}$$

The domain of contexts stores the context's virtual address space function, the program map, a list of loaded programs, the context index, and a list of grids that are currently executing a kernel of one of the context's program environments. Additionally, the context communicates with the graphics driver using context input/output actions defined in section 5.7.2.

$$\zeta \in Context = VirtMemSpace \times ProgMap \times ProgEnv^{*} \times ContextIdx \times ContextIO \times Grid^{*}$$

5.7.1 Formalization of Warp, Thread Block, and Grid Scheduling

At the context level we are finally able to schedule warps, thread blocks, and grids. Fermi-based GPUs are capable of executing up to 16 grids belonging to the same context concurrently. Therefore, the utilization of the streaming multiprocessors is only known at the context level but not at lower levels as mentioned before. The context semantics are responsible for issuing thread blocks to available SMs, for scheduling warps for each SM, and for scheduling grid execution. We define three functions that handle these scheduling tasks; however, as the scheduling algorithms used by the hardware are unknown, we do not give concrete implementations. There are some assumptions that we have to make though. We document them in the remainder of this section.

The $schedule_{Grids}$ function selects a subset of a context's grids that are subsequently allowed to execute a step. As Nvidia has just recently announced that future GPUs will support kernel preemption, we assume that it is currently unsupported.

$$schedule_{Grids} : Grid^* \to Grid^*$$

The purpose of the function *issueBlocks* is to distribute thread blocks to streaming multiprocessors with available execution capacity. As SM utilization is known at the context level because only one context is active at any point in time, *issueBlocks* is able to assign unscheduled thread blocks to available SMs. When it assigns a thread block to a SM, it performs the following steps: The thread block's state is set to active and its processor index is set to the index of the processor the thread block is assigned to. All warps and threads of the thread block are created. The warps' program counters are set to the first address of the entry point which can be obtained from the grid's program configuration. Additionally, the threads' call stacks are initialized with the entry point's local variable environment. Also, the stack offset is set to its initial value; the concrete value depends on the amount and sizes of globally defined local memory variables in the program. If no unscheduled thread blocks are left or all streaming multiprocessors are fully occupied, the function returns $\bot$. Otherwise, it returns all grids passed to it — with the appropriate changes made to the newly scheduled grids and all other grids unchanged.

$$issueBlocks : Grid^* \to Grid^*_\bot$$

A warp contains threads with successive thread indices, i.e. the first warp of a block contains threads 0 to 31, the second warp consists of threads 32 to 63, and so on. We use the one-dimensional representation $(tid)_1$ of a thread index tid to formalize this axiom.

$$\forall \omega \in Warp\ .\ \exists n \in \mathbb{N}\ .\ n \bmod WarpSize = 0 \land \{(\theta_{tid})_1 \mid \theta \in \omega_{\vec{\theta}}\} = \{n, \ldots, n + WarpSize - 1\}$$

Additionally, within a warp, threads are sorted in an ascending order based on their one-dimensional thread index representation.

$$\begin{aligned} &\forall \omega \in Warp\ .\ \forall wl_1, wl_2 \in WarpLane\ .\ wl_1 > wl_2 \Rightarrow \\ &\qquad \exists (\theta_1 \triangleright tid_1, wl_1), (\theta_2 \triangleright tid_2, wl_2) \in \omega_{\vec{\theta}}\ .\ (tid_1)_1 > (tid_2)_1 \end{aligned}$$

If the amount of threads executing a thread block $\beta \triangleright \vec{\omega}$ with program configuration $\xi \triangleright blockSize$ is not a multiple of the warp size, the last warp is filled with dummy threads whose active mask is inactive and whose disable mask is set to exit.

$$\begin{aligned} blockSize \bmod WarpSize \neq 0 \Rightarrow \exists \omega \in \vec{\omega}\ .\ &(blockSize \div WarpSize) \cdot WarpSize \in \{\theta_{tid} \mid \theta \in \omega_{\vec{\theta}}\} \\ &\land \forall wl \in WarpLane\ .\ wl \geq blockSize \bmod WarpSize \\ &\qquad \Rightarrow \omega_\alpha(wl) = \text{inactive} \land \omega_\delta(wl) = \text{exit} \end{aligned}$$

The function *canBeScheduled* determines whether the given warp is in a state that allows it to execute its next instruction. More precisely, the warp's state must be active; the warp does not wait for any barriers to complete, i.e. its barrier index list is empty; no active thread waits for a memory barrier to complete, i.e. the memory barrier levels of all threads of the warp are ε; and all registers of all active threads used by the next instruction are available, i.e. not blocked.

$$\mathit{canBeScheduled} : \mathit{Warp} \to \mathbb{B}$$

Based on the *canBeScheduled* function, we define the scheduling function $\mathit{schedule}_{\mathit{Warps}}$. It determines a schedule for the warps of scheduled grids, using the supplied grid register look up function to check whether any accessed register of a potentially scheduled warp is blocked, i.e. whether $\bot$ is returned for any accessed register. It returns a block schedule for each scheduled grid.

$$\mathit{schedule}_{\mathit{Warps}} : \mathit{Grid}^{*} \times \mathit{GridRegLookup} \times \mathit{ProgMap} \times \mathit{ProgEnv}^{*} \to (\mathit{Grid} \to \mathit{BlockSchedule}_{\bot})$$

5.7.2 Context Input/Output

The graphics driver instructs the GPU to launch kernels, load programs, abort grids, and so on. Some of these actions are handled at the device level, others are processed at the context level. Conversely, the contexts and the device report back to the driver when a grid is completed, a program has been loaded successfully, etc. We define a separate rule system for context I/O actions to avoid cluttering up the rules of the context semantics too much.

The driver can order a context to launch a kernel of some previously loaded program with some program configuration. The program configuration determines the entry point to use, as well as the grid and thread block dimensions among other things. Additionally, the driver may abort a grid that has not yet finished execution. Loading a PTX program into a context is handled at the device level, however, because the memory environment might be modified during the loading process.

$$\mathit{cin} \in \mathit{ContextInput} ::= \texttt{start}\ \mathit{ProgIdx}\ \mathit{ProgConf} \mid \texttt{abort}\ \mathit{GridIdx}$$

The context informs the graphics driver about completed programs and sends the grid index of a newly started grid back to the host program. The host program needs to know the grid index to be able to abort it. It is unclear whether the GPU or the graphics driver are responsible for index assignment; we assume its the GPU's task.

$$\mathit{cout} \in \mathit{ContextOuput} ::= \texttt{started}\ \mathit{GridIdx} \mid \texttt{completed}\ \mathit{GridIdx}$$

The domain of context I/O actions therefore comprises lists of input and output actions. However, we do not enforce any ordering of incoming and outgoing messages.

$$\mathit{cio} \in \mathit{ContextIO} = \mathit{ContextInput}^{*} \times \mathit{ContextOutput}^{*}$$

Transition system $\to^{I/O_C}$ handles the I/O actions of a context. There are three different types of transitions: Either only an input action is handled, only an output action is generated, or an input action is handled and an output action is emitted at the same time. The rules of transition system $\to^{I/O_C}$ are used by the rules of the context semantics later on.

$$\mathit{Configurations} : \mathit{cio} \in \mathit{ContextIO}$$

$$\mathit{Transitions} : \mathit{cio} \xrightarrow[\mathit{cin}]{\mathit{cout}}{}^{I/O_{\mathrm{C}}} \mathit{cio}', \qquad \mathit{cio} \xrightarrow[\mathit{cin}]{}{}^{I/O_{\mathrm{C}}} \mathit{cio}', \qquad \mathit{cio} \xrightarrow{\mathit{cout}}{}^{I/O_{\mathrm{C}}} \mathit{cio}'$$

Once a program has launched a kernel, the corresponding input action is removed from the list of pending input actions. The `started` message containing the grid index of the grid just launched is stored in the output list and submitted to the graphics driver later on.

$$\text{(start)}\ \ \langle(\texttt{start}\ \mathit{progid}\ \xi) \circ \overrightarrow{\mathit{cin}}, \overrightarrow{\mathit{cout}}\rangle \xrightarrow[\texttt{start}\ \mathit{progid}\ \xi]{\texttt{started}\ \mathit{gid}}{}^{I/O_{\mathrm{C}}} \langle\overrightarrow{\mathit{cin}}, (\texttt{started}\ \mathit{gid}) \circ \overrightarrow{\mathit{cout}}\rangle$$

Once an `abort` request is processed, it is removed from the list of pending input actions. We do not output a confirmation to the graphics driver.

$$\text{(abort)}\ \ \langle(\texttt{abort}\ \mathit{gid}) \circ \overrightarrow{\mathit{cin}}, \overrightarrow{\mathit{cout}}\rangle \xrightarrow[\texttt{abort}\ \mathit{gid}]{}{}^{I/O_{\mathrm{C}}} \langle\overrightarrow{\mathit{cin}}, \overrightarrow{\mathit{cout}}\rangle$$

Once a grid completes, we add a `completed` message to the output list. At some later point in time, the message will be sent to the graphics driver.

$$\text{(compl)}\ \ \langle\overrightarrow{\mathit{cin}}, \overrightarrow{\mathit{cout}}\rangle \xrightarrow{\texttt{completed}\ \mathit{gid}}{}^{I/O_{\mathrm{C}}} \langle\overrightarrow{\mathit{cin}}, (\texttt{completed}\ \mathit{gid}) \circ \overrightarrow{\mathit{cout}}\rangle$$

5.7.3 Context Rules

Transition system $\rightarrow^{\mathrm{C}}$ defines the semantics of contexts. In each step, a context can either output several context actions or a context output message. Alternatively, a context input message can be sent to a context.

$$\begin{aligned}
\mathit{Configurations} &: \langle\zeta \mid \varrho\rangle \in \mathit{Context} \times \mathit{GridRegLookup}, \qquad \zeta \in \mathit{Context} \\
\mathit{Transitions} &: \langle\zeta \mid \varrho\rangle \xrightarrow[\overrightarrow{\mathit{cact}}]{}{}^{\mathrm{C}} \zeta', \qquad \langle\zeta \mid \varrho\rangle \xrightarrow[\mathit{cout}]{}{}^{\mathrm{C}} \zeta', \qquad \langle\zeta \mid \varrho\rangle \xrightarrow{\mathit{cin}}{}^{\mathrm{C}} \zeta'
\end{aligned}$$

Rule (exec) executes a step of each scheduled grid and passes down a block schedule to each executing grid. If the emitted context action does not contain any thread block actions, the action is ignored at the device level.

$$\text{(exec)}\ \ \frac{\left(\langle\gamma_i \mid \vec{\phi}_{pm(\gamma_{gid})}, \varrho, s_\Gamma(\gamma)\rangle \xrightarrow[\overrightarrow{\mathit{bact}_\gamma}]{}{}^{\Gamma} \gamma_i'\right)_{\gamma_i \in \vec{\gamma}}}{\langle\zeta \triangleright \mathit{vms}, \mathit{pm}, \vec{\phi}, \gamma_1 \ldots \gamma_n \mid \varrho\rangle \xrightarrow[\mathit{vms}\ \bigcup_{\gamma \in \vec{\gamma}} \overrightarrow{\mathit{bact}_\gamma}]{}{}^{\mathrm{C}} \langle\zeta \triangleleft \gamma_1' \ldots \gamma_n'\rangle}$$

$$\begin{aligned}
\text{if}\quad \forall \gamma_i \in \gamma_1 \ldots \gamma_n \setminus \vec{\gamma}\ .\ \gamma_i' = \gamma_i \wedge \vec{\gamma} &= \mathit{schedule}_{\mathit{Grids}}(\gamma_1 \ldots \gamma_n) \\
\wedge\ s_\Gamma &= \mathit{schedule}_{\mathit{Warps}}(\vec{\gamma}, \varrho, \mathit{pm}, \vec{\phi})
\end{aligned}$$

If there are any unscheduled thread blocks and streaming multiprocessors with available execution capacity, the (issue) rule uses the *issueBlocks* function to assign unscheduled thread blocks to available SMs.

$$\text{(issue)}\ \ \frac{\langle\zeta \triangleleft \mathit{issueBlocks}(\gamma_1 \ldots \gamma_n) \mid \varrho\rangle \xrightarrow[\overrightarrow{\mathit{cact}}]{}{}^{\mathrm{C}} \zeta'}{\langle\zeta \triangleright \gamma_1 \ldots \gamma_n \mid \varrho\rangle \xrightarrow[\overrightarrow{\mathit{cact}}]{}{}^{\mathrm{C}} \zeta'}$$

If one of the context's grids is completed, it is removed from the context and an output message informing the host program about the completion of the kernel call is generated.

$$\text{(compl)}\quad \frac{cio \xrightarrow{\texttt{completed}\ \gamma_{i,gid}}{}^{I/O_{\mathrm{C}}}\ cio' \qquad \langle \zeta \mid \varrho \rangle \xrightarrow[\overrightarrow{cact}]{}{}^{\mathrm{C}} \langle \zeta' \triangleright cio, \gamma_1 \ldots \gamma_{i-1}\ \gamma_i\ \gamma_{i+1} \ldots \gamma_n \rangle}{\langle \zeta \mid \varrho \rangle \xrightarrow[\overrightarrow{cact}]{}{}^{\mathrm{C}} \langle \zeta' \triangleleft cio', \gamma_1 \ldots \gamma_{i-1}\ \gamma_{i+1} \ldots \gamma_n \rangle}$$

$$\text{if} \quad \gamma_{i,gridState} = \text{completed}$$

If a context input message is passed to a context, the message is stored in the context's input/output list for later processing. Incoming messages are not processed in any particular order.

$$\text{(in)}\quad \langle \zeta \triangleright (cio \triangleright \overrightarrow{cin}) \mid \varrho \rangle \xrightarrow{cin}{}^{\mathrm{C}} \langle \zeta \triangleleft (cio \triangleleft cin \circ \overrightarrow{cin}) \rangle$$

Context output messages that have been generated by earlier steps can be output in any order at any point in time.

$$\text{(out)}\quad \langle \zeta \triangleright (cio \triangleright cout \circ \overrightarrow{cout}) \mid \varrho \rangle \xrightarrow[cout]{}{}^{\mathrm{C}} \langle \zeta \triangleleft (cio \triangleleft \overrightarrow{cout}) \rangle$$

A kernel launch command requires that the program the launched entry point belongs to has previously been loaded into the context. A new and unique grid index is generated and subsequently sent back to the host program. The program map is updated so that the correct program environment can be passed to the grid when it executes a step. The abstract function *newGrid* : *GridIdx* × *ProgConf* → *Grid* creates a new grid initialized with the given grid index and program configuration. All of the grid's thread blocks are created but remain in the unscheduled state until *issueBlocks* starts assigning thread blocks to streaming multiprocessors. The newly created grid is stored in the context's list of grids.

$$\text{(start)}\quad \frac{cio \xrightarrow[\texttt{start}\ progid\ \xi]{\texttt{started}\ gid}{}^{I/O_{\mathrm{C}}}\ cio' \qquad \langle \zeta \triangleleft pm[gid \mapsto progid], cio', \gamma_1 \ldots \gamma_n\ \gamma \ \mid \varrho \rangle \xrightarrow[\overrightarrow{cact}]{}{}^{\mathrm{C}} \zeta'}{\langle \zeta \triangleright pm, \vec{\phi}, cio, \gamma_1 \ldots \gamma_n \mid \varrho \rangle \xrightarrow[\overrightarrow{cact}]{}{}^{\mathrm{C}} \zeta'}$$

$$\text{where} \quad \gamma = newGrid(gid, \xi) \wedge \exists \phi \in \vec{\phi} \,.\, \phi_{progid} = progid \wedge \forall \gamma' \in \gamma_1 \ldots \gamma_n \,.\, \gamma'_{gid} \neq gid$$

The host program can abort execution of any kernel at any point in time. The aborted grid is removed from the context's list of grids and from the program map. Pending memory operations issued by the aborted grid are not aborted; as it is undefined when a grid is actually aborted, it is unpredictable which memory operations are completed before the abortion of the grid. Hence, aborting pending memory operations of aborted grids neither improves nor worsens predictability. Additionally, it is unclear whether the hardware contains logic dedicated to aborting outstanding memory operations of aborted grids.

$$\text{(abort)}\quad \frac{cio \xrightarrow[\texttt{abort}\ gid]{}{}^{I/O_{\mathrm{C}}}\ cio' \qquad \langle \zeta \triangleleft pm[gid \mapsto \bot], cio', \gamma_1 \ldots \gamma_{i-1}\ \gamma_{i+1} \ldots \gamma_n \mid \varrho \rangle \xrightarrow[\overrightarrow{cact}]{}{}^{\mathrm{C}} \zeta'}{\langle \zeta \triangleright pm, cio, \gamma_1 \ldots \gamma_{i-1}\ \gamma_i\ \gamma_{i+1} \ldots \gamma_n \mid \varrho \rangle \xrightarrow[\overrightarrow{cact}]{}{}^{\mathrm{C}} \zeta'}$$

$$\text{if} \quad \gamma_{i,gid} = gid$$

5.8 Device Semantics

The device represents the topmost level of the semantics hierarchy. Even though CUDA supports SLI configurations where two or more GPUs can be used by a CUDA application, our formalization focuses on the semantics of a single device only. We also do not take GPU/CPU interactions into account, as this would add yet another level of complexity to the semantics.

The domain of devices manages the list of contexts created on the device. Additionally, the device communicates with the host program using device input/output actions as defined in section 5.8.2. The memory manager establishes the connection between the program semantics and the memory environment. We take a closer look at this connection in the following section.

$$\Delta \in \mathit{Device} = \mathit{MemMgr} \times \mathit{DeviceIO} \times \mathit{Context}^*$$

5.8.1 Linking the Program Semantics to the Memory Environment

Memory operations are issued at the thread level. Higher levels of the hierarchy group thread memory actions and amend additional data. At the device level, memory operations are stored in context actions. The memory environment on the other hand does not know about threads or context actions at all. It operates on memory operations which in turn are defined by a memory program, an implicit program state, and a memory operation priority. Consequently, we need to define a mapping from context actions to memory operations to connect the program semantics to the memory environment.

First, we define the domain of memory managers. A memory manager consists of a memory environment that represents the state of all types of memory supported by our formalization of CUDA's memory model. Additionally, it manages a list of pending memory operations as well as a function that maps a thread index to memory operation indices. The latter allows us to check whether a thread has completed a memory barrier by inspecting the state of the pending memory requests issued by the thread.

$$mm \in \mathit{MemMgr} = \mathit{MemEnv} \times \mathit{MemOp}^* \times (\mathit{ThreadIdx} \rightarrow (\mathit{MemOpIdx}^*)_\perp)$$

The function *memOp* takes a memory manager and a list of context actions and returns an updated memory manager. This function is responsible for converting memory requests issued by some threads into memory operations understood by the memory environment. We leave the function abstract, as most of the conversion process is not sufficiently specified by the CUDA documentation.

$$\mathit{memOp} : \mathit{MemMgr} \times \mathit{ContextAction}^* \rightarrow \mathit{MemMgr}_\perp$$

However, there are several assumptions we have to make about the conversion process. First of all, registers are always big enough to store a complete data word; memory operations, on the other hand, might specify smaller access sizes. This is not a problem for read operations as a smaller value can always be written into a larger register. Though for write operations, some of the register's bits are cut off if the access size is smaller than the size of the data word. But this does not pose a problem because of the preprocessing we perform to convert a PTX program into a program environment: During the preprocessing, the compiler checks whether the program is type safe, i.e. whether all memory operations operate on input operands of matching sizes. For instance, a 64 bit register cannot be the source register of a 32 bit write operation. Type safety guarantees that the source register of a store operation is of equal or smaller size than the memory access size. Consequently, cutting off the highest bits of a register does not result in a loss of data, as these bits are guaranteed to be zero in any case.

The *memOp* function inspects the given context actions. Empty context actions as well as thread block actions with empty warp memory actions are ignored. If the context actions indeed contain valid memory requests, the context actions contain all necessary information for the *memOp* function to select the correct memory program for the given memory operation type and to initialize the implicit memory program state accordingly. Memory operation coalescing is also performed by *memOp*. Furthermore, all registers affected by the operations are blocked and will only become available again once the associated memory program releases them.

memOp also assigns priorities to all generated memory operations. As explained in chapter 4, in most cases we do not know the order in which memory operations are processed. We do know, however, that volatile reads and writes of the same thread are guaranteed to be processed in the order they were issued, hence we assume that *memOp* assigns lower priorities to volatile operations that are issued later during the execution of the program. Non-volatile operations might all be prioritized equally or there might be some unknown prioritization. Additionally, if threads of the same warp write to the same location in memory, it is undefined how many writes are actually processed and which write is processed last. The same is true for several atomic operations on the same address performed by threads of the same warp. If there are n concurrent accesses to the same address, the *memOp* function is free to generate between one to n memory operations with any priority.

In our formalization of the PTX semantics, if a thread executes a `membar` instruction, the requested memory barrier level is stored in the thread's state. As long as the memory barrier level is not reset to ε, the warp scheduler is unable to schedule the thread's warp. The CUDA specification does not explain in detail when a memory barrier completes. Therefore, we define the abstract function *updMemBar* that checks the state of all memory operations associated with a thread waiting for a memory barrier. If all of those operations have advanced as far as necessary, the *updMemBar* function resets the thread's memory barrier to ε, allowing it to be scheduled again. *updMemBar* uses the information stored by the memory manager to retrieve all memory operations associated with a thread. When the *memOp* function creates a new memory operation, the memory managers mapping of thread indices to memory operation indices is updated accordingly. To avoid having to clean up this mapping once a memory operation completes, we assume that the *memOp* function never reuses a memory operation index that it has previously used to identify a memory operation, even after this memory operation has long been completed.

$$\mathit{updMemBar} : \mathit{Context} \times \mathit{MemMgr} \rightarrow \mathit{Context}$$

5.8.2 Device Input/Output

We introduce device input and output messages similar to the context input/output mechanism defined in section 5.7.2. The device has to patch through context messages to the context level. Therefore, the device's input actions comprise messages that contain a context input action and the index of the context the messages should be forwarded to. Additionally, the host program can instruct the device to create or destroy a context as well as to load a program into a specific context. The domain *Prog* represents PTX programs after the application of the code transformations presented in section 5.1.3.

$$\begin{aligned} din \in \mathit{DeviceInput} ::= {} & \texttt{load}\ \mathit{Prog}\ \mathit{ContextIdx} \mid \texttt{create} \mid \texttt{destroy}\ \mathit{ContextIdx} \\ \mid {} & \mathit{ContextInput}\ \mathit{ContextIdx} \end{aligned}$$

Conversely, messages output by the device patch through message emitted by the contexts. Furthermore, messages containing the indices of newly created contexts and recently loaded programs are sent to the host program.

$$\begin{aligned} dout \in \mathit{DeviceOuput} ::= {} & \texttt{created}\ \mathit{ContextIdx} \mid \mathit{ContextOutput}\ \mathit{ContextIdx} \\ \mid {} & \texttt{loaded}\ \mathit{ProgIdx}\ \mathit{ContextIdx} \end{aligned}$$

Analogous to the domain of context I/O actions, we define the domain of device I/O actions that comprises lists of input and output actions. Like for contexts, we do not enforce any ordering of incoming and outgoing messages.

$$dio \in \mathit{DeviceIO} = \mathit{DeviceInput}^{*} \times \mathit{DeviceOutput}^{*}$$

Transition system $\rightarrow^{I/O_{\mathrm{D}}}$ handles the I/O actions of a device. A transition can either generate some output based on some input or silently handle some input without generating any output.

$$\begin{aligned} \mathit{Configurations} : {} & dio \in \mathit{DeviceIO} \\ \mathit{Transitions} : {} & dio \xrightarrow[din]{dout}{}^{I/O_{\mathrm{D}}} dio', \qquad dio \xrightarrow[din]{}{}^{I/O_{\mathrm{D}}} dio' \end{aligned}$$

Once a program has been loaded, the corresponding input action is removed from the device's list of input actions. Moreover, the index of the newly loaded program is returned to the host program later on.

$$\text{(load)}\ \ \langle(\texttt{load}\ prog\ cid) \circ \overrightarrow{din}, \overrightarrow{dout}\rangle \xrightarrow[\texttt{load}\ prog\ cid]{\texttt{loaded}\ progid\ cid}{}^{I/O_{\mathrm{D}}} \langle \overrightarrow{din}, (\texttt{loaded}\ progid\ cid) \circ \overrightarrow{dout}\rangle$$

In a similar fashion, the index of a newly created context is returned to the host program while the corresponding input action is removed.

$$\text{(create)}\ \ \langle(\texttt{create}) \circ \overrightarrow{din}, \overrightarrow{dout}\rangle \xrightarrow[\texttt{create}]{\texttt{created}\ cid}{}^{I/O_{\mathrm{D}}} \langle \overrightarrow{din}, (\texttt{created}\ cid) \circ \overrightarrow{dout}\rangle$$

Destroying a context does not generate any output and removes the corresponding input action from the list.

$$\text{(destroy)}\ \ \langle(\texttt{destroy}\ cid) \circ \overrightarrow{din}, \overrightarrow{dout}\rangle \xrightarrow[\texttt{destroy}\ cid]{}{}^{I/O_{\mathrm{D}}} \langle \overrightarrow{din}, \overrightarrow{dout}\rangle$$

5.8.3 Device Rules

Transition system $\rightarrow^{\mathrm{D}}$ defines the semantics of the device. It does not generate any device actions, because it represents the highest level of the semantics hierarchy. It can, however, receive input messages from the host program or send output messages to the CPU.

$$\begin{aligned}
&\mathit{Configurations} : \Delta \in \mathit{Device}\\
&\quad \mathit{Transitions} : \Delta \rightarrow^{\mathrm{D}} \Delta', \qquad \Delta \xrightarrow[\mathit{dout}]{}^{\mathrm{D}} \Delta', \qquad \Delta \xrightarrow{\mathit{din}}^{\mathrm{D}} \Delta'
\end{aligned}$$

We also finally define how the register look up functions retrieve register values from the memory environment. Given a memory environment, the *regLookup* function returns a grid register look up function.

$$\mathit{regLookup} : \mathit{MemEnv} \rightarrow \mathit{GridRegLookup}$$

We define the behavior of the register look up function as follows. The function parameters are specified at different levels of the hierarchy as explained in the preceding sections. Most importantly, the register look up function does not allow retrieving a value from a blocked register.

$$\begin{aligned}
&\mathit{regLookup}(\eta)(\mathit{gridSize}, \mathit{blockSize})(\mathit{bid}, \mathit{pid})(\mathit{tid}, \mathit{ro})\\
&\qquad (\mathit{regAddr}) = d \qquad\qquad\qquad \text{if} \quad (d, \text{ready}) = \mathit{regFile}(\eta, \mathit{pid})(\mathit{ro} + \mathit{regAddr})\\
&\qquad (\texttt{\%tid.}\mathit{dim}) = \mathit{tid}_{\mathit{dim}}\\
&\qquad (\texttt{\%ntid.}\mathit{dim}) = \mathit{blockSize}_{\mathit{dim}}\\
&\qquad (\texttt{\%ctaid.}\mathit{dim}) = \mathit{bid}_{\mathit{dim}}\\
&\qquad (\texttt{\%nctaid.}\mathit{dim}) = \mathit{gridSize}_{\mathit{dim}}
\end{aligned}$$

The (exec) rule selects a context to execute; contexts cannot be executed concurrently and the contexts scheduling happens implicitly. The *regLookup* function is used to create a register look up function that is passed all the way down to the thread rules and expression evaluation functions to synchronously retrieve a value from an unblocked register. The *updMemBar* function checks whether any of the context's threads have completed a memory barrier. All memory requests issued by the context's threads are converted into memory operations using the *memOp* function. While the selected context executes one step, the memory environment executes a macro step, allowing several memory operations to advance and several silent rules to be invoked. Program execution and memory operation processing are performed in a truly parallel manner.

$$\text{(exec)}\ \frac{\langle \mathit{updMemBar}(\zeta_i, \mathit{mm}) \mid \mathit{regLookup}(\eta)\rangle \xrightarrow[\overrightarrow{\mathit{cact}}]{}^{\mathrm{C}} \zeta'_i \qquad \langle \eta, \overrightarrow{\mathit{op}}\rangle \rightarrow^{\mathrm{M},*} \langle \eta', \overrightarrow{\mathit{op}'}\rangle}{\langle \Delta \triangleright (\mathit{mm} \triangleright \eta, \overrightarrow{\mathit{op}}), \zeta_1 \ldots \zeta_i \ldots \zeta_n\rangle \rightarrow^{\mathrm{D}} \langle \Delta \triangleleft \mathit{memOp}((\mathit{mm} \triangleleft \eta', \overrightarrow{\mathit{op}'}), \overrightarrow{\mathit{cact}}), \zeta_1 \ldots \zeta'_i \ldots \zeta_n\rangle}$$

Input messages are inserted into the device's input list and are serviced at a later execution step. There is no guaranteed ordering of input messages.

$$\text{(in)}\quad \langle \Delta \triangleright (\mathit{dio} \triangleright \overrightarrow{\mathit{din}})\rangle \xrightarrow{\mathit{din}}^{\mathrm{D}} \langle \Delta \triangleleft (\mathit{dio} \triangleleft \mathit{din} \circ \overrightarrow{\mathit{din}})\rangle$$

Conversely, output messages created during an earlier execution step are sent to the host program in an unpredictable order.

$$\text{(out)}\quad \langle \Delta \triangleright (dio \triangleright dout \circ \overrightarrow{dout})\rangle \xrightarrow[dout]{}{}^{\mathrm{D}} \langle \Delta \triangleleft (dio \triangleleft \overrightarrow{dout})\rangle$$

Input messages addressed to a specific context are forwarded to that context using the (cin) rule.

$$\text{(cin)}\quad \frac{\langle \zeta_i \mid regLookup(\eta)\rangle \xrightarrow{cin}{}^{\mathrm{C}} \zeta_i'}{\langle \Delta \triangleright (mm \triangleright \eta), (dio \triangleright (cin\ cid) \circ \overrightarrow{din}), \zeta_1 \ldots \zeta_i \ldots \zeta_n\rangle \rightarrow^{\mathrm{D}} \langle \Delta \triangleleft (dio \triangleleft \overrightarrow{din}), \zeta_1 \ldots \zeta_i' \ldots \zeta_n\rangle}$$

$$\text{if}\quad \zeta_{i,cid} = cid$$

Output actions emitted by a context are added to the device's output list and will be sent to the host program at a later point in time.

$$\text{(cout)}\quad \frac{\langle \zeta_i \triangleright cid \mid regLookup(\eta)\rangle \xrightarrow[cout]{}{}^{\mathrm{C}} \zeta_i'}{\langle \Delta \triangleright (mm \triangleright \eta), (din \triangleright \overrightarrow{dout}), \zeta_1 \ldots \zeta_i \ldots \zeta_n\rangle \rightarrow^{\mathrm{D}} \langle \Delta \triangleleft (dio \triangleleft (cout\ cid) \circ \overrightarrow{dout}), \zeta_1 \ldots \zeta_i' \ldots \zeta_n\rangle}$$

The abstract function *newVM* : *MemEnv* × *Context** → *VirtMemSpace* creates a new virtual memory space for a newly created context. Different contexts to do not share physical memory addresses, hence *newVM* does not map any virtual addresses to a physical address that is already mapped by another context. Other than that, the newly created context returned by *newCtx* : *VirtMemSpace* × *ContextIdx* → *Context* is empty, i.e. it does not yet have any grids, programs, I/O messages, or a program map. The index assigned to the new context must be unique.

$$\text{(create)}\quad \frac{dio \xrightarrow[\texttt{create}]{\texttt{created}\ cid}{}^{I/O_{\mathrm{D}}}\ dio' \qquad \langle \Delta \triangleleft dio', \zeta_1 \ldots \zeta_n\ newCtx(vms, cid)\rangle \rightarrow^{\mathrm{D}} \Delta'}{\langle \Delta \triangleright (mm \triangleright \eta), dio, \zeta_1 \ldots \zeta_n\rangle \rightarrow^{\mathrm{D}} \Delta'}$$

$$\text{if}\quad vms = newVM(\eta, \zeta_1 \ldots \zeta_n) \wedge \forall i \in \{1 \ldots n\}\ .\ \zeta_{i,cid} \neq cid$$

A context can be destroyed at any time, implicitly aborting all of the context's grids. As already discussed in section 5.7.3 for individual grid abortion, pending memory operations issued by the aborted grids are not canceled.

$$\text{(destroy)}\quad \frac{dio \xrightarrow[\texttt{destroy}\ cid]{}{}^{I/O_{\mathrm{D}}}\ dio' \qquad \langle \Delta \triangleleft dio', \zeta_1 \ldots \zeta_{i-1}\ \zeta_{i+1} \ldots \zeta_n\rangle \rightarrow^{\mathrm{D}} \Delta'}{\langle \Delta \triangleright dio, \zeta_1 \ldots \zeta_{i-1}\ \zeta_i\ \zeta_{i+1} \ldots \zeta_n\rangle \rightarrow^{\mathrm{D}} \Delta'} \quad \text{if}\quad \zeta_{i,cid} = cid$$

Loading a program into a context relies on the *load* function already defined in section 5.1.3. The newly loaded program is added to the context's program list and the program is assigned a context-wide unique program index. The *load* function returns a memory environment that guarantees that all threads read the correct values of all global and constant variables for which the PTX program specifies initial values.

$$\text{(load)}\quad \frac{dio \xrightarrow[\texttt{load}\ prog\ \zeta_{cid}]{\texttt{loaded}\ progid\ \zeta_{cid}}{}^{I/O_{\mathrm{D}}}\ dio' \qquad \langle \Delta \triangleleft (mm \triangleleft \eta'), dio', (\zeta \triangleleft \phi \circ \zeta_{\vec{\phi}}) \circ \vec{\zeta}\rangle \rightarrow^{\mathrm{D}} \Delta'}{\langle \Delta \triangleright (mm \triangleright \eta), dio, \zeta \circ \vec{\zeta}\rangle \rightarrow^{\mathrm{D}} \Delta'}$$

$$\text{if}\quad (\eta', \phi) = load(\eta, prog, progid) \wedge \forall \phi \in \zeta_{\vec{\phi}}\ .\ \phi_{progid} \neq progid$$

5.9 Summary

Our formalization of CUDA's model of computation closely reflects the thread hierarchy established by CUDA for a single GPU. We do not consider multi-GPU setups or the interactions of the GPU and the CPU. While it is possible to execute CUDA programs on a single GPU only, a CUDA program always consists of device code and host code. We completely omit any discussion of the host code semantics; in that sense the formalization of the CUDA program semantics presented in this report is incomplete. But as this report shows, proving a property of a CUDA program will not be an easy task considering the amount and the complexity of the rules making up the semantics of CUDA programs and the memory model. Future work should definitely strive to employ tools to check the soundness of the formalization and to enable tool-supported proofs.

Just like there are many open questions that remain unanswered by our formalization of the memory model, the formalized program semantics are also based on many assumptions whenever the CUDA specification falls short of details. We circumvent this issue for the warp level branching algorithm by basing our formalization on a patent filed by Nvidia, as the official documentation gives almost no hints as to the workings of the algorithm. In other cases, particularly thread block barrier synchronization, there are also many undefined situations where we either try to make reasonable assumptions or use $\perp$ to indicate that the behavior of the program is unpredictable as far as we are aware.

Many CUDA programs are written in CUDA-C instead of PTX. While there might be performance reasons to write a program in PTX, the CUDA-C compiler usually does a good job in optimizing the generated PTX code. Nvidia's compiler developers have a lot of experience in writing highly optimized compilers for C-like languages, as GPU drivers have optimized shader code written in HLSL or GLSL for almost a decade now. Nevertheless, defining the semantics for PTX instead of CUDA-C gives us more precise knowledge of the actual hardware operations used. During the formalization of the semantics it becomes clear that even more low level details are required, so the semantics lies somewhere in between PTX and the actual instruction set of the underlying hardware.

6 Correctness of the Branching Algorithm

Only algorithms that are inherently data-parallel can fully utilize the GPU's functional units. The ratio of memory operations to floating point operations and whether memory accesses can be coalesced also greatly influence the execution efficiency of CUDA programs. Consequently, programmers devote most of their time to optimizing the aforementioned issues. By contrast, less thought is given to the behavior of basic instructions like `add`, `setp`, or `return`; their semantics are expected to be intuitively clear, i.e. they are expected to behave just like their CPU counterparts.

The semantics of control flow instructions, however, are not based on individual thread branching that is commonplace on x86 CPUs. Instead, when a warp's threads execute control flow instructions, the branching happens at the warp level as already outlined in section 5.4.1. Nvidia suggests that developers can "essentially ignore" [3, 4.1] this difference when thinking about program correctness, so we should be able to prove the correctness of CUDA's branching algorithm for all valid programs.

Informally, correctness means that whenever a thread encounters a control flow instruction, the path taken by the thread must be the same whether the decision is made by the thread individually or by the warp. Additionally, all instructions must be executed in the correct order, the thread must not skip an instruction that it should have executed, and the thread must not execute any further instructions after having executed an `exit` statement.

As we show in the remainder of this section, CUDA's branching algorithm affects program correctness in some cases. This becomes more clear when we decompose the meaning of correctness into a safety and a liveness property:

- *Safety*: All threads only execute the instructions they are allowed to execute, in the correct order, and without skipping any instructions. The branching algorithm satisfies the safety property.
- *Liveness*: All threads eventually execute all instructions they must execute. In general, the branching algorithm does not satisfy the liveness property.

Even though the liveness property does not hold in the general case — as shown in the next section — it is still possible to prove the safety property for all programs which fulfill the conditions imposed on them in section 6.2. Furthermore, we narrow down the scope of the semantics to a single warp in section 6.3, as the semantics for thread blocks, grids, contexts, and so on are of no interest for the proof. We then formalize and prove the safety property and give a formal definition of the liveness property in sections 6.4 and 6.5, respectively.

6.1 Implications of Warp Level Branching

The most obvious effects of warp level branching concern program performance. When all threads execute different code paths, the warp has to serially execute all threads and cannot fully utilize all of the streaming processor's parallel execution capabilities. In the worst case, program performance decreases by a factor proportional to the GPU's warp size and the branching algorithm itself is also likely to cause some overhead. Although these performance considerations are generally of high importance for general purpose GPU programming, we focus exclusively on the semantic implications of CUDA's branching algorithm.

As already mentioned above, we can decompose the meaning of correctness into a safety and a liveness property. Whereas the safety property holds for all programs including the following one, the liveness property might be violated as subsequently illustrated for program 6.1. Thus, the following example illustrates how the behavior of a CUDA application can be affected by the warp level branching algorithm.

Program 6.1 tries to manually establish a critical section that can only be entered by one single thread at any given point in time; critical sections are not directly supported by CUDA. In line 1, the variable `lock` is declared that is used to indicate whether there currently is a thread executing the critical section. The variable lives in global memory and is therefore accessible by all threads. Inside the kernel, an atomic compare-and-swap operation within a while-loop is used to check whether a thread is allowed to access the critical section. `atomicCAS` returns `1` for one single thread only — which one is undefined, the memory model is free to process the atomic memory requests for the location of `lock` in any order. If `atomicCAS` does not return `1`, meaning that there is already another thread executing the critical section, the thread continues to execute the while-loop until it is finally able to obtain the lock. A thread that is allowed to enter the critical section resets the value of `lock` to indicate that it has completed the execution of the critical section and that another thread may enter it. These are the semantics a developer might have in mind when writing program 6.1. And if the program were run on a x86 CPU where threads are allowed to branch individually, this is precisely what would happen.

```
 1 __device__ int lock = 1;
 2
 3 __global__ void kernel()
 4 {
 5   while (true)
 6   {
 7     if (atomicCAS(&lock, 1, 0) == 1) // Acquire lock
 8     {
 9       break;
10     }
11   }
12
13   // Inside critical section
14   lock = 1; // Release lock
15 }
```

Listing 6.1: A program written in CUDA-C that deadlocks due to warp level branching

CUDA's warp level branching algorithm changes the behavior of the program: When executed on the GPU, the program deadlocks and never terminates if it is executed by more than one thread concurrently. To explain this, let us suppose two threads θ_1 and θ_2 run program 6.2, which is the PTX version of program 6.1.

Kernel execution starts at line 8 where the threads encounter the `preBrk` instruction and the warp pushes a break token onto the stack. As the condition of the while-loop in line

```
1 .global .u32 lock = 1;
2
3 .entry kernel ()
4 {
5   .reg .u32 r;
6   .reg .pred p;
7
8   preBrk ENDLOOP;
9
10   BEGINLOOP:
11     atom .global .cas .u32 r, lock, 1, 0;  // Acquire lock
12     setp .u32 .eq p, r, 1;
13     @p break;
14     bra BEGINLOOP;
15
16   ENDLOOP:
17     // Inside critical section
18     st .global .u32 .cg lock, 1; // Release lock
19     exit;
20 }
```

Listing 6.2: Program 6.1 compiled into PTX; `preBrk` instruction included for clarity

5 of program 6.1 is `true`, no condition needs to be checked and the loop's body is entered immediately at line 11. There, the threads load the address of the global variable `lock` and execute the atomic compare-and-swap operation. In line 12, the threads check whether the atomic memory operation returned `1` and store the result of the comparison in the predicate register `p`.

Subsequently, the threads execute the conditional `break` statement in line 13. Suppose the guard `p` is true for θ_1; consequently `p` is false for θ_2 because of the semantics of the `atomicCAS` instruction. Thus, θ_1 executes the `break` instruction and rule (brk_{div}) of transition system $\rightarrow^{\text{CRS}}$ sets θ_1's disable mask to break. θ_2, on the other hand, is not disabled and continues to execute program 6.2. In this situation, the program is already deadlocked: θ_1 is reactivated only once the break token is popped from the stack, but this cannot happen as long as θ_2 is still active. θ_2, however, cannot execute the `break` instruction and become disabled, because it has to wait for the atomic operation to return `1`, which in turn can only happen if θ_1 continues execution.

Obviously, all terminating programs fulfill the liveness property, as no deadlock can occur if all threads eventually execute an `exit` instruction. Also, a thread that reaches an `exit` statement has processed all instructions it must execute, assuming that the safety property holds.

6.2 Control Flow Consistency

The branching algorithm depends on certain instructions that the compiler injects into PTX programs. If it does so incorrectly, the behavior of a program is unpredictable. Since we do not know the algorithm the compiler uses to inject control flow instructions into PTX programs, we make the injection explicit in the definition of the semantics in chapter 5 by having the instruction environment explicitly contain the required branching instructions. The conversion process that translates a PTX program into a program environment is represented by the abstract function *load* in section 5.1.3, which, among other things, correctly injects the necessary control flow operations into the program. For the proof though, we have to know more about the program's control flow, the location of control flow instructions within the program, and the layout of the warp's stack. As giving a concrete implementation of the abstract *load* function is outside the scope of this report, we axiomatically define the class of control flow consistent programs instead. Control flow consistent programs allow us to reason about the occurrence of control flow instructions and possible stack layouts, which in turn enables us to prove the safety property without relying on implementation details of the compiler. Future work might render the following axiomatic definition of control flow consistent programs unnecessary by giving a concrete implementation of the *load* function's branching instructions injection algorithm.

Control flow consistent programs must satisfy two properties which we subsequently define. The first one is stack consistency $\textit{stack-cons} : \textit{ProgEnv} \to \mathbb{B}$, whose definition depends on the following set operations on active masks: For two active masks α_1, α_2, we write $\alpha_1 \subseteq \alpha_2$ to mean $\forall wl \in \textit{WarpLane} \,.\, \alpha_1(wl) = \text{active} \Rightarrow \alpha_2(wl) = \text{active}$. Furthermore, $\alpha = \alpha_1 \cup \alpha_2$ stands for $\forall wl \in \textit{WarpLane} \,.\, \alpha(wl) = \text{active} \Leftrightarrow \alpha_1(wl) = \text{active} \vee \alpha_2(wl) = \text{active}$. Moreover, $\textit{execState}_\phi$ denotes the set of all execution states that can occur when a warp executes program ϕ.

Definition (Stack Consistency)**:** A program ϕ is stack consistent if and only if the following holds for all execution states $(\varsigma \triangleright pc, \alpha, \delta, \vec{\tau}) \in \textit{execState}_\phi$:

$$
\begin{array}{ll}
(1) & \ \phi_{\textit{instrEnv}}(pc) = \texttt{break} \Rightarrow \exists \tau \in \vec{\tau} \,.\, \tau_{\textit{tokenType}} = \text{break} \\
(2) & \wedge\ \phi_{\textit{instrEnv}}(pc) = \texttt{return} \Rightarrow \exists \tau \in \vec{\tau} \,.\, \tau_{\textit{tokenType}} = \text{call} \\
(3) & \wedge\ \phi_{\textit{instrEnv}}(pc) = \texttt{sync} \Rightarrow \vec{\tau}_{0,\textit{tokenType}} = \text{sync} \vee \vec{\tau}_{0,\textit{tokenType}} = \text{diverge} \\
(4) & \wedge\ \phi_{\textit{instrEnv}}(pc) = \texttt{sync} \wedge \vec{\tau}_{0,\textit{tokenType}} = \text{diverge} \\
 & \qquad\qquad \Rightarrow \vec{\tau}_{1,\textit{tokenType}} = \text{sync} \wedge \vec{\tau}_{1,pc} = pc + 1 \wedge \vec{\tau}_{0,\alpha} \cup \alpha \subseteq \vec{\tau}_{1,\alpha} \\
(5) & \wedge\ \phi_{\textit{instrEnv}}(pc) = \texttt{sync} \wedge \vec{\tau}_{0,\textit{tokenType}} = \text{sync} \Rightarrow \vec{\tau}_{0,pc} = pc + 1 \\
(6) & \wedge\ \phi_{\textit{instrEnv}}(pc) = \texttt{call} \Rightarrow \exists \tau \in \vec{\tau} \,.\, \tau_{\textit{tokenType}} = \text{call} \wedge \tau_{pc} = pc + 1
\end{array}
$$

Definition (Program Address Consistency)**:** A program ϕ is program address consistent if and only if all control flow instructions reference valid program addresses.

We merely give an informal definition of the meaning of "valid program addresses", because even thread level branching would become totally unpredictable if a control flow instruction ever referenced an invalid program address. For the proof, we never explicitly depend on program address consistency, so the following informal definition is fully adequate: Whenever a thread executes a `bra` instruction, the address denoted by the operation is a valid program address within the same function. This applies to direct branches, in which case this property can be and is enforced by the compiler, as well as indirect branches, in which case it is not enforceable by the compiler. When a `call` instruction is encountered, the address specified by the operation is the address of a function's first instruction. Again, for direct function calls this is enforced by the compiler, for indirect function calls the compiler is generally unable to prove it. Additionally, the compiler makes sure that the labels of `sync`, `preBrk`, and `preRet` instructions are valid, i.e. are defined in the same function.

Definition (Control Flow Consistency)**:** A program ϕ is control flow consistent if and only if it is both stack and program address consistent.

$ProgEnv_{cfc} \subseteq ProgEnv$ denotes the class of control flow consistent programs. We write $\hat{\phi}$ to distinguish control flow consistent programs $\hat{\phi} \in ProgEnv_{cfc}$ from non control flow consistent programs $\phi \in ProgEnv \setminus ProgEnv_{cfc}$.

6.3 Single Warp Semantics

Since two distinct warps execute independently and are free to branch individually without affecting one another, we narrow down the scope of the semantics to a single warp. This allows us to focus on what the warp and its threads do when a control flow instruction is executed without having to consider all the complexities of thread blocks, thread synchronization, grids, contexts, and so on. Doing so does not limit the proof's generality however. We reuse most of the rules and domains of the thread and warp semantics defined in sections 5.3 and 5.4 for reasons of brevity; only some minor changes are necessary. Furthermore, we reuse the memory model completely unchanged from its definition in chapter 4.

6.3.1 Modified Thread Rules

For the proof, a thread has to know the program counter of the next instruction it should execute. Since the program counter of the next instruction is normally only known to the warp, we define a new domain $Thread_{pc}$ for threads ϑ which keeps track of the next program address:

$$\vartheta \in Thread_{pc} = ProgAddr_{\varepsilon} \times Thread$$

Reusing the original domain *Thread* allows us to also reuse transition system $\rightarrow^{\Theta}$. The rules of $\rightarrow^{\Theta}$ defined in section 5.3.2 specify how a thread executes PTX instructions and which thread action each instruction generates. We do not have to make any changes to the rules of transition system $\rightarrow^{\Theta}$, even though we are actually only interested in control flow instructions for the purpose of the proof, in which case $\rightarrow^{\Theta}$ outputs a $FlowAct_{\varepsilon}$. Reusing all of $\rightarrow^{\Theta}$'s rules and only looking at thread flow actions enables us to abstract from all non control flow instructions while at the same time supporting all previously defined thread operations and interactions between threads with minimal effort.

On the other hand, the transition system $\rightarrow^{\Theta_e}$ cannot be reused. It is replaced by transition system $\rightarrow^{\Theta_{pc}}$ defined below that, in addition to checking the thread's active mask and guard, also updates the thread's program counter.

After a thread ϑ has executed its current instruction, ϑ must be updated with the program address of the next instruction that should be executed according to the program's control flow. This is the responsibility of the function $branch_{\Theta}$.

$$branch_{\Theta} : ProgAddr \times ExecToken^{*} \times Act_{\Theta,\varepsilon} \rightarrow ProgAddr_{\varepsilon,\perp}$$

If the thread does not output a thread action, or if the thread action is a communication or memory action, the program counter is incremented by 1, as the next instruction that should be executed is the next instruction of the program. This is also true if the thread encounters a `ssy`, `preRet`, `preBrk`, or `sync` instruction which have no meaning at the thread level. On the other hand, if the thread performed an `exit`, there is no next instruction, because the thread terminates. In this case, $branch_\Theta$ returns an invalid program address, ε, to indicate that the thread must not be allowed to execute any further instructions.

In the case of `break`, `return`, `bra`, and `call` instructions, the program counter is set to the address denoted by the control flow operation. For branches and function calls, this is the address specified by the operation. Breaks and returns, however, do not specify the target address directly. Instead, the target address is stored in the topmost break or call token on the warp's stack of execution tokens. We pass the warp's current stack $\vec{\tau}$ to $branch_\Theta$ instead of managing a stack of break and call tokens per thread for reasons of brevity. Moreover, the stack of any thread could always be constructed from the warp's stack by removing all diverge and sync tokens as well as all break and call tokens in which the thread's active mask is set to inactive. Hence, per-thread stacks are redundant and we rather use the function *topmostToken* to get the topmost token of a given type from the warp's stack instead. When this function is used, it is assumed that such a token exists on the stack. Thus, not finding the specified token is an error and $\perp$ is returned.

$$\begin{aligned}
&\mathit{topmostToken} : \mathit{ExecToken}^* \times \mathit{TokenType} \to \mathit{ExecToken}_\perp \\
\mathit{topmostToken}(\varepsilon, \mathit{tokenType}) &= \perp \\
\mathit{topmostToken}(\tau :: \vec{\tau}, \mathit{tokenType}) &= \begin{cases} \tau & \text{if} \quad \tau_{\mathit{tokenType}} = \mathit{tokenType} \\ \mathit{topmostToken}(\vec{\tau}, \mathit{tokenType}) & \text{otherwise} \end{cases}
\end{aligned}$$

We define the function $branch_\Theta$ as follows, using _ to leave out irrelevant values of an action:

$$\begin{aligned}
branch_\Theta(pc, \vec{\tau}, \varepsilon) &= pc + 1 \\
branch_\Theta(pc, \vec{\tau}, tact_{comm}) &= pc + 1 \\
branch_\Theta(pc, \vec{\tau}, tact_{mem}) &= pc + 1 \\
branch_\Theta(pc, \vec{\tau}, \texttt{ssy}\ \_) &= pc + 1 \\
branch_\Theta(pc, \vec{\tau}, \texttt{preRet}\ \_) &= pc + 1 \\
branch_\Theta(pc, \vec{\tau}, \texttt{preBrk}\ \_) &= pc + 1 \\
branch_\Theta(pc, \vec{\tau}, \texttt{sync}) &= pc + 1 \\
branch_\Theta(pc, \vec{\tau}, \texttt{break}\ \_) &= \mathit{topmostToken}(\vec{\tau}, \mathrm{break})_{pc} \\
branch_\Theta(pc, \vec{\tau}, \texttt{return}\ \_) &= \mathit{topmostToken}(\vec{\tau}, \mathrm{call})_{pc} \\
branch_\Theta(pc, \vec{\tau}, \texttt{exit}\ \_) &= \varepsilon \\
branch_\Theta(pc, \vec{\tau}, \texttt{bra}\ wl\ pc') &= pc' \\
branch_\Theta(pc, \vec{\tau}, \texttt{call}\ wl\ pc') &= pc'
\end{aligned}$$

In section 5.3.3, the rules of the transition system $\to^{\Theta_e}$ only need to know the warp's current program counter and active mask. By contrast, $\to^{\Theta_{pc}}$ depends on the warp's entire execution state ς:

$$\begin{aligned}
\mathit{Configurations} &: \langle \vartheta \mid \phi, \varsigma, \varrho \rangle \in \mathit{Thread}_{pc} \times \mathit{ProgEnv} \times \mathit{ExecState} \times \mathit{ThreadRegLookup}, \\
&\quad \vartheta \in \mathit{Thread}_{pc} \\
\mathit{Transitions} &: \langle \vartheta \mid \phi, \varsigma, \varrho \rangle \xrightarrow[tact_\varepsilon]{}^{\Theta_{pc}} \vartheta'
\end{aligned}$$

If the thread's active mask is set to active and the guard is true, $\rightarrow^{\Theta_{pc}}$ uses $\rightarrow^{\Theta}$ to execute the current instruction, just as $\rightarrow^{\Theta_e}$ does. Additionally, the generated thread action and the $branch_\Theta$ function determine the next program address of the thread:

$$(\text{Exec}_{\text{tt}}) \quad \frac{\langle \theta \mid \phi_{instrEnv}(pc), \phi, \varrho(tid, ro)\rangle \xrightarrow[tact_\varepsilon]{}^{\Theta} \theta'}{\langle \vartheta \triangleright pc, \theta \triangleright \kappa, ro, tid, wl \mid \phi, (\varsigma \triangleright \alpha, \vec{\tau}), \varrho\rangle \xrightarrow[tact_\varepsilon]{}^{\Theta_{pc}} \langle \vartheta \triangleleft branch_\Theta(pc, \vec{\tau}, tact_\varepsilon), \theta'\rangle}$$

$$\text{if} \quad \alpha(wl) = \text{active} \wedge \mathcal{B}[\![\phi_{guardEnv}(pc)]\!]\phi\varrho(tid, ro)\kappa$$

We have to define two distinct rules for the cases when either the guard is false or the thread is disabled according to the active mask. These two cases cannot be handled in a single rule as done for $\rightarrow^{\Theta_e}$ because of the program counter updates: When the thread is active but the guard is false, we have to increment the thread's program counter, since the thread has to skip the current instruction and execute the next one in the next step. On the other hand, if the thread's active mask is set to inactive, the thread really does not do anything and the program counter has to remain unchanged. Once the warp activates the thread again, this is the instruction the thread executes next.

$$(\text{Exec}_{\text{ff}_1}) \quad \langle \vartheta \triangleright pc, \theta \triangleright \kappa, ro, tid, wl \mid \phi, (\varsigma \triangleright \alpha, \vec{\tau}), \varrho\rangle \rightarrow^{\Theta_{pc}} \langle \vartheta \triangleleft pc + 1\rangle$$

$$\text{if} \quad \alpha(wl) = \text{active} \wedge \neg\mathcal{B}[\![\phi_{guardEnv}(pc)]\!]\phi\varrho(tid, ro)\kappa$$

$$(\text{Exec}_{\text{ff}_2}) \quad \langle \vartheta \triangleright \theta \triangleright wl \mid \phi, \varsigma \triangleright \alpha, \varrho\rangle \rightarrow^{\Theta_{pc}} \vartheta \qquad \text{if} \quad \alpha(wl) \neq \text{active}$$

6.3.2 Modified Warp Rules

Warps now manage threads of type $Thread_{pc}$ instead of $Thread$. Like for threads, we introduce a new domain $Warp_\vartheta$ for warps w in conjunction with the new transition system $\rightarrow^{\Omega_\vartheta}$:

$$w \in Warp_\vartheta = WarpState \times ExecState \times BarrierIdx^* \times Thread_{pc}^{WarpSize}$$

$$\begin{aligned} Configurations &: \langle w \mid \phi, \varrho\rangle \in Warp_\vartheta \times ProgEnv \times ThreadRegLookup, \qquad \omega \in Warp_\vartheta \\ Transitions &: \langle w \mid \phi, \varrho\rangle \xrightarrow[wact_\varepsilon]{}^{\Omega_\vartheta} w' \end{aligned}$$

If the threads execute a control flow instruction, their program counters are updated according to the rules of the $\rightarrow^{\Theta_{pc}}$ transition system. The warp's execution state, i.e. program counter, active mask, disable mask, and stack, is updated by $\rightarrow^{\text{CRS}}$ depending on the actions output by the threads.

There is one complication though: When we formalize the safety property, we basically say that the program counters of all active threads are equal to the warp's program counter. This ensures that the warp and the threads always agree on the instruction being executed and consequently guarantees that the threads are always asked to execute the correct instruction. Even though it is an intuitive and easy to understand formalization of the safety property, it does not hold in the case of indirect branching instructions and indirect calls with more than one target address. Suppose a thread ϑ executes an indirect `bra` instruction and the rule ($\text{bra}_{\text{target}}$) is used at the warp level to handle the divergent control flow. This means that there are at least two different target addresses and a diverge token with the current program counter is pushed onto the stack. If ϑ's active mask is active in the next step, ϑ executes the instruction specified by the branch instruction, which is the program address stored by both the warp and the thread. The problem occurs if ϑ is not allowed to continue execution immediately. ϑ remains inactive until the diverge token is popped off the stack, but then ϑ's program counter is no longer equal to the warp's program counter: The warp wants all threads that were deactivated during the first encounter of the indirect branch to execute the `bra` instruction again, whereas ϑ wants to immediately jump to the target address specified by the branch operation.

There are three possible solutions to this problem, all of which have certain advantages and disadvantages:

- Use another formalization of the safety property which avoids this issue.
- Change the behavior of indirect branches and calls such that the instruction is not revisited and an appropriate amount of diverge tokens is pushed onto the stack to ensure equivalent behavior.
- Reset the program counters of all deactivated threads that performed the indirect branch or call.

We take the latter approach because this is what the hardware actually does and it allows us to use an intuitive formalization of the safety property. This decision comes at the cost of making $\rightarrow^{\Omega_\vartheta}$ less intuitive, as we have to define and use the function *reset* which detects the described situation and resets the program counters of all affected threads. The *reset* function resets a thread's program counter to the warp's previous program counter if the thread is not allowed to immediately jump to the target address of an indirect branch or call. If all threads participating in an indirect branch or call agree on one common target address, the aforementioned issue does not occur and *reset* does not change the threads' program counters. In all other cases besides indirect branches or calls, the threads' program counters remain unchanged as well.

$$
\begin{aligned}
&\mathit{reset} : \mathit{Thread}_{pc}^{\mathit{WarpSize}} \times \mathit{FlowAct}^{*} \times \mathit{ProgAddr} \times \mathit{ActiveMask} \times \mathit{ActiveMask} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathit{Thread}_{pc}^{\mathit{WarpSize}} \\
&\mathit{reset}(\vartheta_1 \ldots \vartheta_n, \texttt{bra}\ wl_1\ pc_1 \ldots \texttt{bra}\ wl_n\ pc_n, pc, \alpha, \alpha') = \vartheta'_1 \ldots \vartheta'_n \\
&\qquad\qquad\qquad\text{where} \quad \vartheta'_i = \begin{cases} \vartheta_i \triangleleft pc & \text{if} \quad \alpha(wl) = \text{active} \wedge \alpha'(wl) = \text{inactive} \\ \vartheta_i & \text{otherwise} \end{cases} \\
&\qquad\qquad\qquad\text{if} \quad \exists i, j \in \{1 \ldots n\} \,.\, pc_i \neq pc_j \\
&\mathit{reset}(\vartheta_1 \ldots \vartheta_n, \texttt{call}\ wl_1\ pc_1 \ldots \texttt{call}\ wl_n\ pc_n, pc, \alpha, \alpha') = \vartheta'_1 \ldots \vartheta'_n \\
&\qquad\qquad\qquad\text{where} \quad \vartheta'_i = \begin{cases} \vartheta_i \triangleleft pc & \text{if} \quad \alpha(wl) = \text{active} \wedge \alpha'(wl) = \text{inactive} \\ \vartheta_i & \text{otherwise} \end{cases} \\
&\qquad\qquad\qquad\text{if} \quad \exists i, j \in \{1 \ldots n\} \,.\, pc_i \neq pc_j \\
&\mathit{reset}(\vartheta_1 \ldots \vartheta_n, \overrightarrow{\mathit{tact}_{\mathit{flow}}}, pc, \alpha, \alpha') = \vartheta_1 \ldots \vartheta_n \qquad \text{otherwise}
\end{aligned}
$$

We can define $\rightarrow^{\Omega_\vartheta}$'s rules based on $\rightarrow^{\Omega}$'s ones. The only differences are the call to the *reset* function in (bra) and the replacement of $\rightarrow^{\Theta_e}$ by $\rightarrow^{\Theta_{pc}}$. For the proof, the (bra) rule is the most interesting one.

$$\text{(bra)}\quad \frac{\langle \vartheta_1 \mid \phi, \varsigma, \varrho \rangle \xrightarrow[tact_{flow,1_\varepsilon}]{}{}^{\Theta_{pc}} \vartheta'_1 \quad \dots \quad \langle \vartheta_n \mid \phi, \varsigma, \varrho \rangle \xrightarrow[tact_{flow,n_\varepsilon}]{}{}^{\Theta_{pc}} \vartheta'_n}{\langle w \triangleright \varsigma, \vartheta_1 \dots \vartheta_n \mid \phi, \varrho \rangle \rightarrow^{\Omega_\vartheta} \langle w \triangleleft state', \varsigma', \vartheta''_1 \dots \vartheta''_n \rangle}$$

$$\text{if} \quad \varsigma \xrightarrow{\bigcup_{i \in \{1 \dots n\}} tact_{flow,i_\varepsilon}}{}^{\mathrm{CRS}} \langle state', \varsigma' \rangle$$

$$\wedge\ \vartheta''_1 \dots \vartheta''_n = reset(\vartheta'_1 \dots \vartheta'_n, \bigcup_{i \in \{1 \dots n\}} tact_{flow,i_\varepsilon}, \varsigma_{pc}, \varsigma_\alpha, \varsigma'_\alpha)$$

$$\text{(comm)}\quad \frac{\langle \vartheta_1 \mid \phi, \varsigma, \varrho \rangle \xrightarrow[tact_{comm,1_\varepsilon}]{}{}^{\Theta_{pc}} \vartheta'_1 \quad \dots \quad \langle \vartheta_n \mid \phi, \varsigma, \varrho \rangle \xrightarrow[tact_{comm,n_\varepsilon}]{}{}^{\Theta_{pc}} \vartheta'_n}{\langle w \triangleright (\varsigma \triangleright pc), \vartheta_1 \dots \vartheta_n \mid \phi, \varrho \rangle \xrightarrow[wact_\varepsilon]{}{}^{\Omega_\vartheta} \langle w \triangleleft (\varsigma \triangleleft pc + 1), \overrightarrow{barid}, \vartheta'_1 \dots \vartheta'_n \rangle}$$

$$\text{if} \quad (\overrightarrow{barid}, wact_\varepsilon) = comm(\phi_{instrEnv}(pc), \bigcup_{i \in \{1 \dots n\}} tact_{comm,i_\varepsilon})$$

$$\wedge\ \exists i \in \{1 \dots n\} \;.\; tact_{comm,i_\varepsilon} \neq \varepsilon$$

$$\text{(mem)}\quad \frac{\langle \vartheta_1 \mid \phi, \varsigma, \varrho \rangle \xrightarrow[tact_{mem,1_\varepsilon}]{}{}^{\Theta_{pc}} \vartheta'_1 \quad \dots \quad \langle \vartheta_n \mid \phi, \varsigma, \varrho \rangle \xrightarrow[tact_{mem,n_\varepsilon}]{}{}^{\Theta_{pc}} \vartheta'_n}{\langle w \triangleright (\varsigma \triangleright pc), \vartheta_1 \dots \vartheta_n \mid \phi, \varrho \rangle \xrightarrow[\bigcup_{i \in \{1 \dots n\}} tact_{mem,i_\varepsilon}]{}{}^{\Omega_\vartheta} \langle w \triangleleft (\varsigma \triangleleft pc + 1), \vartheta'_1 \dots \vartheta'_n \rangle}$$

$$\text{if} \quad \exists i \in \{1 \dots n\} \;.\; tact_{mem,i_\varepsilon} \neq \varepsilon$$

We need one more transition system, $\rightarrow^{\Omega_s}$, to complete the definition of the single warp semantics. The purpose of $\rightarrow^{\Omega_s}$ is to combine the $\rightarrow^{\Omega_\vartheta}$ transition system with the memory environment. The warp actions output by the rules above contain communication and memory actions generated by the warp's threads. Communication actions occur when the threads execute a `vote` or `bar` instruction; the former is handled at the warp level by the *comm* function, the latter has no meaning in the context of a single warp: All threads of the same warp always execute a `bar` instruction at the same time as explained in section 5.5.1, regardless of how many threads are actually active. Subsequently, the warp is disabled until the required amount of other threads in the thread block reached the barrier as well. Thus, the only effect of a barrier in the context of a single warp is its associated memory barrier.

Memory actions have to be passed to the memory environment. For the purpose of the proof we do not care about the specifics of the memory model and its supported operations, so we reuse the memory environment formalized in section 4 for reasons of brevity. We also reuse the the *MemMgr* domain and the *memOp* function defined in section 5.8.1 to hook up the memory environment to the warp's rule system. To be able to do this, we need a function that packages up a warp action as a context action.

$$at \in ActTranslator = Act_{\Omega,\varepsilon} \rightarrow Act_{\mathrm{C},\varepsilon}$$

We do not explicitly state how the process of translating a warp action into a context action works, i.e. what parameters are used for the processor index, grid index, and so on. We just expect a *ActTranslator* function be passed to $\rightarrow^{\Omega_s}$ which uses parameters consistent with those used by the thread register lookup function ϱ also passed to $\rightarrow^{\Omega_s}$. We do require, however, that the context action returned by the translator is ε if the warp action passed to the translator function is a barrier action.

We encapsulate the translator and the register lookup function as well as the the memory manager in a new domain *Mem*.

$$m \in \mathit{Mem} = \mathit{MemMgr} \times \mathit{ThreadRegLookup} \times \mathit{ActTranslator}$$

We also need an *updMemBar* function similar to the one defined for devices in section 5.8.1 with the only difference being that it operates on warps instead of contexts. This function resets the memory barrier level of the warp's threads once all of the threads' memory operations are no longer affected by the barrier.

$$\mathit{updMemBar} : \mathit{Warp} \times \mathit{MemMgr} \to \mathit{Warp}$$

We can then specify the two rules of the final transition system $\to^{\Omega_s}$. The first one executes one step of the warp and allows the memory environment to advance an arbitrary amount of steps at the same time, similar to how the device's (exec) rule works in section 5.8.3. The rule then uses the action translator function to send any generated memory actions to the memory environment. The second rule handles the case that the warp cannot perform a step because either all of its threads have executed an `exit` instruction, one of its active threads has to wait for registers to be released, or one of its threads has to wait for a memory barrier to be completed. The memory environment can execute an arbitrary amount of steps nevertheless. The function *canBeScheduled* introduced in section 5.7.1 is used to decide whether the warp is allowed to execute the next step.

$$\begin{aligned} \mathit{Configurations} &: \langle w \mid \phi, m\rangle \in \mathit{Warp}_{\vartheta} \times \mathit{ProgEnv} \times \mathit{Mem} \\ \mathit{Transitions} &: \langle w \mid \phi, m\rangle \to^{\Omega_s} \langle w' \mid \phi, m'\rangle \end{aligned}$$

$$\text{(step)} \quad \frac{\langle w \mid \phi, \varrho\rangle \xrightarrow[\mathit{wact}_{\varepsilon}]{}^{\Omega_{\vartheta}} w' \qquad \langle \eta, \overrightarrow{op}\rangle \to^{\mathrm{M},*} \langle \eta', \overrightarrow{op'}\rangle}{\langle w \mid \phi, (m \triangleright (mm \triangleright \eta, \overrightarrow{op}), \varrho, at)\rangle \to^{\Omega_s} \langle w' \mid \phi, (m \triangleleft \mathit{memOp}((mm \triangleleft \eta', \overrightarrow{op'}), at(\mathit{wact}_{\varepsilon})))\rangle} \quad \text{if} \quad \mathit{canBeScheduled}(w)$$

$$\text{(idle)} \quad \frac{\langle \eta, \overrightarrow{op}\rangle \to^{\mathrm{M},*} \langle \eta', \overrightarrow{op'}\rangle}{\langle w \mid \phi, m \triangleright (mm \triangleright \eta, \overrightarrow{op})\rangle \to^{\Omega_s} \langle \mathit{updMemBar}(w, mm) \mid \phi, m \triangleleft (mm \triangleleft \eta', \overrightarrow{op'})\rangle} \quad \text{if} \quad \neg\mathit{canBeScheduled}(w)$$

6.4 Proof of the Safety Property

The preceding sections have already referred to the safety property several times, albeit in an informal way; we now give a formal definition and a proof. To recapitulate, the safety property demands that warp level branching respects the control flow of each individual thread: A thread that has executed an `exit` statement is never again activated by the warp, no instructions that should be executed are skipped, the correct instruction order is maintained, and the thread does not execute any instructions it should not execute.

We now have to find a formalization for all of these points. To do this, we first define two properties and an invariant that we need for the proof of the safety property. Subsequently, we prove two lemma, a theorem, and a corollary before we finally prove the safety property itself.

First of all, to avoid the execution of instructions that a thread should not execute, we require that the thread's and the warp's program counters are equal whenever the thread is active. We call this the active consistency property $\mathit{act\text{-}cons} : \mathit{ExecState} \times \mathit{Thread}_{pc} \rightarrow \mathbb{B}$.

Definition (Active Consistency)**:** A warp $w \triangleright (\varsigma \triangleright pc_w, \alpha)$ is active consistent if and only if the following holds for all of its threads $(\vartheta \triangleright pc_\vartheta, \theta \triangleright wl) \in w_{\vec{\vartheta}}$:

$$\mathit{act\text{-}cons}(\varsigma, \vartheta) \Leftrightarrow \alpha(wl) = \text{active} \Rightarrow pc_w = pc_\vartheta$$

Active consistency also ensures that a thread remains disabled for the remainder of a program after it has executed an `exit` instruction: The branch_Θ function sets the thread's program counter to ε if the thread executes an `exit`, which is not a program address that can ever be stored in the warp's execution state. Therefore, if we prove active consistency, we solve the exit issue for free.

Additionally, active consistency also enforces the correct instruction order and makes skipping of instructions that should be executed impossible as long as the thread remains active. However, once a thread is deactivated by the warp and resumes execution at a later point in time, active consistency is not enough. When the warp deactivates a thread, and the thread's last instruction was not an `exit`, there must be a token on the stack that activates the thread once the token is popped off the stack. We call this token the thread's activation token and define the function *actToken* that finds a thread's activation token on the stack:

$$\mathit{actToken} : \mathit{ActiveMask} \times \mathit{DisableMask} \times \mathit{ExecState}^* \times \mathit{WarpLane} \rightarrow \mathit{ExecToken}_\perp$$

A token is a thread's activation token if the thread is currently inactive but has not yet completed execution, if the thread is active according to the token's active mask, and, if the thread is waiting for a call or break token, the token has the matching type. Only the topmost token on the stack fulfilling these requirements is the activation token. Therefore, the activation token is well-defined for all inactive but not yet completed threads. There is no activation token for active or completed threads.

$$\begin{aligned}
\mathit{actToken}(\alpha, \delta, \varepsilon, wl) &= \perp \\
\mathit{actToken}(\alpha, \delta, \tau :: \vec{\tau}, wl) &= \begin{cases} \tau & \text{if} \quad \alpha(wl) = \text{inactive} \wedge \delta(wl) \neq \text{exit} \\ & \quad \wedge\ \tau_\alpha(wl) = \text{active} \\ & \quad \wedge\ (\delta(wl) = \text{break} \Rightarrow \tau_{\mathit{tokenType}} = \text{break}) \\ & \quad \wedge\ (\delta(wl) = \text{return} \Rightarrow \tau_{\mathit{tokenType}} = \text{call}) \\ \mathit{actToken}(\alpha, \delta, \vec{\tau}, wl) & \text{otherwise} \end{cases}
\end{aligned}$$

When a warp deactivates a thread due to a branch, function call, `return` or `break`, we know there must be an activation token on the stack that reactivates the thread. When a thread is reactivated, we have to enforce that the warp asks the thread to execute the correct instruction, i.e. the next instruction the thread should execute, ensuring that no instructions are skipped and that the thread is allowed to execute the instruction. This means that the program address stored in the activation token must be the address of the instruction the thread wants to execute next. We call this the inactive consistency property $\mathit{inact\text{-}cons} : \mathit{ExecState} \times \mathit{Thread}_{pc} \rightarrow \mathbb{B}$.

Definition (Inactive Consistency)**:** A warp $w \triangleright (\varsigma \triangleright \alpha, \delta, \vec{\tau})$ is inactive consistent if and only if the following holds for all of its threads $(\vartheta \triangleright pc_{\vartheta}, \theta \triangleright wl) \in w_{\vec{\vartheta}}$:

$$\mathit{inact\text{-}cons}(\varsigma, \vartheta) \Leftrightarrow \alpha(wl) = \text{inactive} \wedge \delta(wl) \neq \text{exit} \Rightarrow pc_{\vartheta} = \mathit{actToken}(\alpha, \delta, \vec{\tau}, wl)_{pc}$$

Together, active and inactive consistency form the safety property. Additionally, the proof of the safety property also relies on the following invariant $\mathit{inv} : \mathit{ExecState} \to \mathbb{B}$ for some execution state $\varsigma \triangleright \alpha, \delta, \vec{\tau}$:

$$\begin{array}{lll}
(1) & & \forall wl \in \mathit{WarpLane} \,.\, \alpha(wl) = \text{inactive} \wedge \delta(wl) \neq \text{exit} \Rightarrow \mathit{actToken}(\alpha, \delta, \vec{\tau}, wl) \neq \bot \\
(2) & \wedge & \exists wl \in \mathit{WarpLane} \,.\, \alpha(wl) = \text{active} \vee \forall wl \in \mathit{WarpLane} \,.\, \delta(wl) = \text{exit} \\
(3) & \wedge & \forall wl \in \mathit{WarpLane} \,.\, \alpha(wl) = \text{active} \\
 & & \qquad \Rightarrow \forall \tau \in \vec{\tau} \,.\, \tau_{\mathit{tokenType}} \neq \text{diverge} \Rightarrow \tau_{\alpha}(wl) = \text{active} \\
(4) & \wedge & \forall wl \in \mathit{WarpLane} \,.\, \alpha(wl) = \text{inactive} \wedge \delta(wl) \neq \text{exit} \\
 & & \qquad \wedge\, \vec{\tau} = \vec{\tau_1} :: \mathit{actToken}(\alpha, \delta, \vec{\tau}, wl) :: \vec{\tau_2} \\
 & & \qquad \Rightarrow \forall \tau \in \vec{\tau_2} \,.\, \tau_{\mathit{tokenType}} \neq \text{diverge} \Rightarrow \tau_{\alpha}(wl) = \text{active} \\
(5) & \wedge & \forall wl \in \mathit{WarpLane} \,.\, \alpha(wl) = \text{active} \Rightarrow \delta(wl) = \text{enabled} \\
 & & \qquad \wedge\, \mathit{actToken}(\alpha, \delta, \vec{\tau}, wl) \neq \bot \Rightarrow \alpha(wl) = \text{inactive} \\
(6) & \wedge & \forall \tau \in \vec{\tau} \,.\, \tau_{\alpha} \neq \alpha^{\mathit{inact}}
\end{array}$$

Furthermore, the following observation is helpful for the proof of the safety property: When the warp uses the *unwind* function to pop tokens from the stack, it is unclear what happens exactly in the general case. Still, if *unwind* is called and all threads are inactive, we can prove the following lemma and the resulting corollary:

Lemma 1: Let $\varsigma \triangleright \alpha, \delta, \vec{\tau}$ and $\varsigma' \triangleright \alpha', \delta', \vec{\tau}'$ be two execution states where $\varsigma' = \mathit{unwind}(\varsigma)$, and $\vec{\vartheta}$ be a set of threads. Then:

$$\begin{aligned}
&\left|\vec{\tau}\right| = \left|\vec{\tau}'\right| + 1 \wedge \mathit{inv}_{1,4,6}(\varsigma) \wedge \forall \vartheta \in \vec{\vartheta} \,.\, \mathit{inact\text{-}cons}(\varsigma, \vartheta) \\
&\qquad \wedge \forall wl \in \mathit{WarpLane} \,.\, \alpha(wl) = \text{inactive} \wedge \exists wl \in \mathit{WarpLane} \,.\, \delta(wl) \neq \text{exit} \\
&\qquad \Rightarrow \mathit{inv}(\varsigma') \wedge \forall \vartheta \in \vec{\vartheta} \,.\, \mathit{act\text{-}cons}(\varsigma', \vartheta) \wedge \mathit{inact\text{-}cons}(\varsigma', \vartheta)
\end{aligned}$$

Proof. For $\vec{\tau} = \tau :: \vec{\tau}''$, we know that $\vec{\tau}' = \vec{\tau}''$, because *unwind* removes tokens from the stack in the opposite order they were pushed onto the stack. Thus, if the call to *unwind* shrinks the stack by one token, the topmost token τ is popped off the stack. The active and disable masks change depending on the token type of τ as is specified by the *updExecState* function. We know that τ is an activation token for at least one thread — otherwise *unwind* would remove more than one token from the stack — and thus at least one thread is active according to the active mask α' and enabled according to the disable mask δ'.

We prove active and inactive consistency for some thread $(\vartheta \triangleright \theta \triangleright wl) \in \vec{\vartheta}$. The function *updExecState* sets the warp's program counter to the program counter of the token. Additionally, it enables ϑ's warp lane in the active mask depending on whether τ is ϑ's activation token. If $\alpha'(wl) = \text{active}$, then $\tau = \mathit{actToken}(\alpha, \delta, \vec{\tau}, wl)$ and $\delta'(wl) \neq \text{exit}$. So $\vartheta_{pc} \overset{\mathit{inact\text{-}cons}(\varsigma,\vartheta)}{=} \tau_{pc} = \varsigma'_{pc}$ and thus $\mathit{act\text{-}cons}(\varsigma', \vartheta)$ holds. If $\alpha'(wl) = \text{inactive}$, $\mathit{inact\text{-}cons}(\varsigma', \vartheta)$ holds because $\tau \neq \mathit{actToken}(\alpha, \delta, \vec{\tau}, wl)$ and $\mathit{inact\text{-}cons}(\varsigma, \vartheta)$.

It remains to be shown that the invariant holds for ς'.

- $inv_1(\varsigma')$: Trivial for all activated threads and for all threads that are still inactive, the activation token has not changed thus $inv_1(\varsigma')$ follows from $inv_1(\varsigma)$.
- $inv_2(\varsigma')$: At least one thread is activated as mentioned above.
- $inv_3(\varsigma')$: Follows from $inv_4(\varsigma)$ for all newly activated threads.
- $inv_4(\varsigma')$: Follows from $inv_4(\varsigma)$ for all threads that remained inactive.
- $inv_5(\varsigma')$: If the function *updExecState* sets a thread's active mask to active, it also sets the thread's disable mask to enabled. Also, if τ is not the activation token of a thread, the thread remains inactive.
- $inv_6(\varsigma')$: No new tokens are pushed onto the stack and $inv_6(\varsigma')$ holds because of $inv_6(\varsigma)$.

□

If there are still threads that have not yet finished execution of the program, the function *unwind* either removes only one token from the stack or several ones, depending on where the topmost activation token is located on the stack. Lemma 1 takes care of the former case, lemma 2 of the latter one:

Lemma 2: Let $\varsigma \triangleright \alpha, \delta, \vec{\tau}$ and $\varsigma' \triangleright \alpha', \delta', \vec{\tau}'$ be two execution states where $\varsigma' = unwind(\varsigma)$, and $\vec{\vartheta}$ be a set of threads. Then:

$$\begin{aligned}\left|\vec{\tau}\right| > \left|\vec{\tau}'\right| + 1 \land inv_{1,4,6}(\varsigma) \land \forall \vartheta \in \vec{\vartheta} \,.\, \textit{inact-cons}(\varsigma, \vartheta)&\\ \land\, \forall wl \in \textit{WarpLane} \,.\, \alpha(wl) = \text{inactive} \land \exists wl \in \textit{WarpLane} \,.\, \delta(wl) \neq \text{exit}&\\ \Rightarrow inv(\varsigma') \land \forall \vartheta \in \vec{\vartheta} \,.\, \textit{act-cons}(\varsigma', \vartheta) \land \textit{inact-cons}(\varsigma', \vartheta)&\end{aligned}$$

Proof. More than one token is popped off the stack and we know that the last token popped off the stack must be an activation token, because there are still threads that have not yet finished execution and $inv_1(\varsigma)$ holds. Thus, $\vec{\tau} = \tau_1 :: \ldots :: \tau_n :: \vec{\tau}''$ and τ_n is an activation token for some thread ϑ whereas all the other τ_i are not an activation token for any of the threads. So popping $\tau_1 \ldots \tau_{n-1}$ from the stack does not change the active mask. Moreover, if the active mask remains inactive for all threads, the disable mask cannot change either: Looking at the definition of *updExecState* makes this obvious for sync and diverge tokens. For call and break tokens on the other hand, we know that the resulting active mask is inactive for all threads. So the active mask stored in the token for some thread is either inactive, and inactive threads remain inactive according to the definition of the $. \setminus .$ function, or active, in which case the final value of the active mask for the thread depends on the value of the disable mask for the thread. If the value of the disable mask is different from the token type, the disable mask remains unchanged and the thread remains inactive. If it is not different, the disable mask is set to enabled and the thread is activated, a contradiction to the assumption that τ is not an activation token for any thread. The same contradiction occurs when a thread's active mask is active according to the token and the disable mask is enabled.

We can now construct an intermediate execution state $\varsigma'' = \varsigma \triangleleft \tau_n :: \vec{\tau}$; the actual value of the program counter of ς'' does not matter as it is overwritten anyway when the activation token τ_n is popped off the stack. For all threads, $\textit{inact-cons}(\varsigma'', \vartheta)$ holds because of $\textit{inact-cons}(\varsigma, \vartheta)$, ς'' still contains the same activation tokens as ς, and ς''_α is inactive for all warp lanes. Also, $inv_{1,4,6}$ still hold for ς'' because of $inv_{1,4,6}(\varsigma)$ and the fact that no activation tokens are popped off the stack and all threads remain inactive. We can now apply lemma 1 to ς'' and $\varsigma' = \mathit{unwind}(\varsigma'')$. As popping tokens 1 to $n-1$ has absolutely no effect on the execution state as shown above, it follows that $\mathit{unwind}(\varsigma) = \mathit{unwind}(\varsigma'') = \varsigma'$ and thus the proof is completed. □

Using lemma 1 and 2 we can prove the general implications a call to *unwind* has on an execution state with all threads disabled.

Corollary 1: Let $\varsigma \triangleright \alpha$ and ς' be two execution states where $\varsigma' = \mathit{unwind}(\varsigma)$, and $\vec{\vartheta}$ be a set of threads. Then:

$$\mathit{inv}_{1,4,6}(\varsigma) \wedge \forall \vartheta \in \vec{\vartheta} \,.\, \mathit{inact\text{-}cons}(\varsigma, \vartheta) \wedge \forall \mathit{wl} \in \mathit{WarpLane} \,.\, \alpha(\mathit{wl}) = \text{inactive}$$
$$\Rightarrow \mathit{inv}(\varsigma') \wedge \forall \vartheta \in \vec{\vartheta} \,.\, \mathit{act\text{-}cons}(\varsigma', \vartheta) \wedge \mathit{inact\text{-}cons}(\varsigma', \vartheta)$$

Proof. First of all, $\varsigma' \neq \bot$. If there is some wl with $\varsigma_\delta(\mathit{wl}) \neq \text{exit}$, then there is an activation token for wl on the stack according to $\mathit{inv}_1(\varsigma)$. Hence $\varsigma_{\vec{\tau}} \neq \varepsilon$ and *unwind* could not possibly return $\bot$.

If all threads have already executed an `exit` instruction, we know that $\varsigma'_{\vec{\tau}} = \varepsilon$, $\varsigma'_\alpha(\mathit{wl}) =$ inactive, and $\varsigma'_\delta(\mathit{wl}) = \text{exit}$ for all warp lanes wl. $\mathit{inv}(\varsigma')$, $\mathit{act\text{-}cons}(\varsigma', \vartheta)$, and $\mathit{inact\text{-}cons}(\varsigma', \vartheta)$ follow trivially for all $\vartheta \in \varsigma'_{\vec{\vartheta}}$.

Otherwise, one or more tokens are popped off the stack, the last of which is an activation token. Lemma 1 and 2 cover these remaining cases. □

We now have everything we need to formalize and prove theorem 1 which we subsequently use to prove the safety property.

Theorem 1 (Consistency Preservation)**:** Let $\hat{\phi}$ be a control flow consistent program, w be a warp and m be a memory configuration. Then:

$$\langle w, \hat{\phi}, m \rangle \to^{\Omega_s} \langle w', \hat{\phi}, m' \rangle \wedge \mathit{inv}(w_\varsigma) \wedge \forall \vartheta \in w_{\vec{\vartheta}} \,.\, \mathit{act\text{-}cons}(w_\varsigma, \vartheta) \wedge \mathit{inact\text{-}cons}(w_\varsigma, \vartheta)$$
$$\Rightarrow \mathit{inv}(w'_\varsigma) \wedge \forall \vartheta \in w'_{\vec{\vartheta}} \,.\, \mathit{act\text{-}cons}(w'_\varsigma, \vartheta) \wedge \mathit{inact\text{-}cons}(w'_\varsigma, \vartheta)$$

Proof. In the following proof, $\varsigma \triangleright pc_w, \alpha, \delta, \vec{\tau}$ denotes the execution state of the warp w before the execution of the step, whereas $\varsigma' \triangleright pc'_w, \alpha', \delta', \vec{\tau}'$ denotes the execution state of the warp w' afterwards. We use some thread $(\vartheta \triangleright pc_\vartheta, \theta \triangleright \mathit{wl}) \in w_{\vec{\vartheta}}$ to prove active and inactive consistency. $(\vartheta' \triangleright pc'_\vartheta) \in w'_{\vec{\vartheta}}$ denotes ϑ's state after the execution of the step, so $\mathit{wl} = \vartheta'_{\theta,\mathit{wl}}$. We assume that $\mathit{inv}(\varsigma)$, $\mathit{act\text{-}cons}(\varsigma, \vartheta)$, and $\mathit{inact\text{-}cons}(\varsigma, \vartheta)$ hold. Additionally, we can also assume stack and program address consistency, because $\hat{\phi} \in \mathit{ProgEnv}_{\mathit{cfc}}$ is a control flow consistent program.

There are two rules for transition system $\to^{\Omega_s}$. If rule (idle) is used, we know that changes to w are restricted to the removal of some memory barriers of some threads. Other than that, the warp's execution state and the threads' program counters remain the same. As the memory environment is irrelevant for the proof, it follows that $\mathit{inv}(\varsigma')$, $\mathit{act\text{-}cons}(\varsigma', \vartheta')$, and $\mathit{inact\text{-}cons}(\varsigma', \vartheta')$ hold.

The more interesting case is rule (step), where the warp and its threads change their state in accordance with the rules of transition system $\to^{\Omega_\vartheta}$. Another case distinction is necessary to handle the three rules of $\to^{\Omega_\vartheta}$. However, we can prove the cases (mem) and (comm) together, as these two rules do not deal with control flow instructions. In both rules, pc_w is incremented by one, so $pc'_w = pc_w + 1$. Furthermore, the warp's execution state remains unchanged except for the program counter, i.e. $\alpha = \alpha'$, $\delta = \delta'$, and $\vec{\tau} = \vec{\tau}'$ and thus $\mathit{inv}(\varsigma')$

holds. If ϑ is inactive, rule (Exec$_{\text{ff},2}$) does not change its program counter and therefore $pc_\vartheta = pc'_\vartheta$. Then $\textit{inact-cons}(\varsigma', \vartheta')$ follows from $\textit{inact-cons}(\varsigma, \vartheta)$. If ϑ is active, either rule (Exec$_{\text{tt}}$) or rule (Exec$_{\text{ff},1}$) is used. In both cases, the thread's program counter is incremented by one according to the definition of rule (Exec$_{\text{ff},1}$) and the function $\textit{branch}_\Theta$ for memory and communication actions. Consequently, $pc'_\vartheta = pc_\vartheta + 1 \overset{\textit{act-cons}(\varsigma,\vartheta)}{=} pc_w + 1 = pc'_w$ and therefore $\textit{act-cons}(\varsigma', \vartheta')$.

By far the largest case is rule (bra) of transition system $\to^{\Omega_\vartheta}$. The threads emit control flow actions $\textit{tact}_{\textit{flow},1_\varepsilon} \ldots \textit{tact}_{\textit{flow},n_\varepsilon}$ that we use to further subdivide this case. We ignore empty control flow actions by declaring $\overrightarrow{\textit{tact}_{\textit{flow}}} = \bigcup_{i\in\{1\ldots n\}} \textit{tact}_{\textit{flow},i_\varepsilon}$.

- $\overrightarrow{\textit{tact}_{\textit{flow}}} = \varepsilon$. In this case, the threads either execute an instruction that does not generate any thread action, the guards of all active threads are false, or a combination thereof. Either way, rule (next) of transition system $\to^{\text{CRS}}$ only increments the warp's program counter, leaving the rest of the execution state unchanged. Moreover, if ϑ is active, rules (Exec$_{\text{tt}}$) or (Exec$_{\text{ff},1}$) both increment ϑ's program counter by one. $\textit{inv}(\varsigma')$, $\textit{act-cons}(\varsigma', \vartheta')$, and $\textit{inact-cons}(\varsigma', \vartheta')$ follow from the exact same reasoning as for the (comm) and (mem) case above.

- $\overrightarrow{\textit{tact}_{\textit{flow}}} = \texttt{ssy}\ pc_1 \ldots \texttt{ssy}\ pc_n$. The (ssy) rule of transition system $\to^{\text{CRS}}$ updates the execution state to $\alpha' = \alpha$, $\delta' = \delta$, $\vec{\tau}' = \tau :: \vec{\tau}$ with $\tau = (\text{sync}, \alpha, pc_1)$, and $pc'_w = pc_w + 1$.
  - $\alpha(wl) = \alpha'(wl) = \text{active}$. As above, the thread executes either rule (Exec$_{\text{tt}}$) or (Exec$_{\text{ff},1}$); both rules set pc'_ϑ to $pc_\vartheta + 1$. Thus $\textit{act-cons}(\varsigma', \vartheta)$ follows from $\textit{act-cons}(\varsigma, \vartheta)$.
  - $\alpha(wl) = \alpha'(wl) = \text{inactive}$. The thread is also inactive in τ's active mask, hence τ is not the activation token for ϑ. If $\delta(wl) \neq \text{exit}$, there must be another token on the stack that is ϑ's activation token because of $\textit{inv}_1(\varsigma)$ and $\textit{inact-cons}(\varsigma, \vartheta)$. Since (Exec$_{\text{ff},2}$) does not change ϑ's program counter, $\textit{inact-cons}(\varsigma', \vartheta)$ holds.

  $\textit{inv}(\varsigma')$ follows from $\textit{inv}(\varsigma)$ as no threads are deactivated or reactivated in this step, τ is not an activation token for any of the threads, and $\tau_\alpha = \alpha = \alpha' \neq \alpha^{\textit{inact}}$.

- $\overrightarrow{\textit{tact}_{\textit{flow}}} = \texttt{preBrk}\ pc_1 \ldots \texttt{preBrk}\ pc_n$. In an analogous manner to the `ssy` case.

- $\overrightarrow{\textit{tact}_{\textit{flow}}} = \texttt{preRet}\ pc_1 \ldots \texttt{preRet}\ pc_n$. In an analogous manner to the `ssy` case.

- $\overrightarrow{\textit{tact}_{\textit{flow}}} = \texttt{sync} \ldots \texttt{sync}$. When the threads execute a `sync` instruction, transition system $\to^{\text{CRS}}$ unwinds the stack. Since program $\hat{\phi}$ is control flow consistent, $\textit{stack-cons}_3$ ensures that the first token τ is removed from the stack $\vec{\tau} = \tau :: \vec{\tau}'$ and that τ is either of type sync or diverge. The next instruction the warp executes is the one the token's program counter points to, thus $pc'_w = \tau_{pc}$.
  - $\tau_{\textit{tokenType}} = \text{diverge}$. At least one thread is active in τ's active mask because of $\textit{inv}_6$. This thread was active when the diverge token was pushed onto the stack but was subsequently deactivated as the taken path was executed. The disable mask for this thread is still enabled because no rule of transition system $\to^{\text{CRS}}$ changes the disable mask of an inactive thread. Thus, τ is the activation token for this thread.
    - $\alpha(wl) = \text{active}$. By inspection of the thread rules, we know that $pc'_\vartheta = pc_\vartheta + 1$.

        - $\alpha'(wl) = \text{active}$. This cannot happen, because when τ was pushed onto the stack, ϑ could not have been active in both the taken and not taken mask at the same time by the definition of the *taken* and *notTaken* functions. If ϑ was active in the taken mask, it cannot be active in τ's active mask and is therefore deactivated by τ. On the other hand, if ϑ was active in the not taken mask, ϑ's activation token would be τ, therefore it cannot be active in α because of $inv_5(\varsigma)$.
        - $\alpha'(wl) = \text{inactive}$. As specified by $stack\text{-}cons_4$, the next token τ' on the stack is a sync token. τ' is ϑ's activation token: $\delta(wl) = \text{enabled}$ because of $inv_5(\varsigma)$ and $\tau'_\alpha(wl) = \text{active}$ because of $stack\text{-}cons_4$. Moreover, $pc'_\vartheta = pc_\vartheta + 1 \overset{act\text{-}cons(\varsigma,\vartheta)}{=} pc_w + 1 \overset{stack\text{-}cons_4}{=} \tau'_{pc}$, thus $inact\text{-}cons(\varsigma', \vartheta)$ holds.
    - $\alpha(wl) = \text{inactive}$. $pc'_\vartheta = pc_\vartheta$. If $\delta(wl) = \text{exit}$, the thread is not reactivated by popping τ.
        - $\alpha'(wl) = \text{active}$. τ is ϑ's activation token, so $act\text{-}cons(\varsigma', \vartheta)$ follows from $pc'_\vartheta = pc_\vartheta \overset{inact\text{-}cons(\varsigma,\vartheta)}{=} \tau_{pc} = pc'_w$.
        - $\alpha'(wl) = \text{inactive}$. As τ is not ϑ's activation token, $inv_1(\varsigma)$ guarantees that there is another token on the stack which is ϑ's activation token if $\delta(wl) \neq \text{exit}$. Because of $inact\text{-}cons(\varsigma, \vartheta)$ and $pc'_\vartheta = pc_\vartheta$, $inact\text{-}cons(\varsigma', \vartheta)$ holds.
- $\tau_{tokenType} = \text{sync}$. Popping τ has the effect that at least all threads that are active in α remain active in α' as guaranteed by $inv_6(\varsigma)$, $inv_3(\varsigma)$, and $inv_5(\varsigma)$. Further threads might be activated in this step.
    - $\alpha(wl) = \text{active}$. By inspection of the thread rules, we know that $pc'_\vartheta = pc_\vartheta + 1$.
        - $\alpha'(wl) = \text{active}$. $pc'_\vartheta = pc_\vartheta + 1 \overset{act\text{-}cons(\varsigma,\vartheta)}{=} pc_w + 1 \overset{stack\text{-}cons_5}{=} \tau_{pc} = pc'_w$, thus $act\text{-}cons(\varsigma', \vartheta)$ follows.
        - $\alpha'(wl) = \text{inactive}$. As explained above, all active threads remain active, so this cannot happen.
    - $\alpha(wl) = \text{inactive}$. In an analogous manner to the diverge token case.

It remains to be shown that the invariant still holds after the execution of the step:

- $inv_1(\varsigma')$: All threads for which τ is not the activation token remain inactive and because of $inv_1(\varsigma)$ there still is an activation token on the stack for them. For all newly deactivated threads, we have shown that the activation token exists.
- $inv_2(\varsigma')$: As τ is an activation token for at least one thread, there is at least one active thread after the execution of this step.
- $inv_3(\varsigma')$: Follows from $inv_3(\varsigma)$ and $inv_4(\varsigma)$.
- $inv_4(\varsigma')$: Follows from $inv_3(\varsigma)$ and $inv_4(\varsigma)$.
- $inv_5(\varsigma')$: If the function *updExecState* sets a thread's active mask to active, it also sets the thread's disable mask to enabled. Also, if τ is not the activation token of a thread, the thread remains inactive.
- $inv_6(\varsigma')$: Follows from $inv_6(\varsigma)$, because no new tokens are pushed onto the stack during this step.

- $\overrightarrow{\mathit{tact}_{\mathit{flow}}} = \texttt{break}\ wl_1 \ldots \texttt{break}\ wl_n$. Breaks can be uniform or divergent, handled by $\rightarrow^{\mathrm{CRS}}$'s rules ($\mathrm{brk}_{\mathrm{uni}}$) and ($\mathrm{brk}_{\mathrm{div}}$) respectively.
    - ($\mathrm{brk}_{\mathrm{uni}}$). We construct an intermediate execution state $\varsigma'' \triangleright \alpha^{\mathit{inact}}, \delta'', \vec{\tau}$ with $\delta'' = \delta[wl_1 \ldots wl_n \mapsto \text{break}]$ and $\varsigma' = \mathit{unwind}(\varsigma'')$. To apply corollary 1, we first observe that all threads are inactive in the active mask of ς'' by the definition of rule ($\mathrm{brk}_{\mathrm{uni}}$). We have to show that ς'' is inactive consistent for all threads and that parts one, four, and six of the invariant hold.
        - $\alpha(wl) = \text{active}$. The thread is disabled according to the active mask of ς'' and waits for a break token according to δ''. The branch_{Θ} function sets pc'_{ϑ} to $\mathit{topmostToken}(\vec{\tau}, \text{break})_{pc}$. $\mathit{stack\text{-}cons}_1$ guarantees that there is a break token on the stack and from $\mathit{inv}_3(\varsigma)$ it follows that the topmost break token is ϑ's activation token. Therefore, $pc'_{\vartheta} = \mathit{topmostToken}(\vec{\tau}, \text{break})_{pc} = \mathit{actToken}(\alpha'', \delta'', \vec{\tau}, wl)_{pc}$ and $\mathit{inact\text{-}cons}(\varsigma'', \vartheta)$ holds.
        - $\alpha(wl) = \text{inactive}$. As the stack remains unchanged, $\mathit{inact\text{-}cons}(\varsigma'', \vartheta)$ follows directly from $\mathit{inact\text{-}cons}(\varsigma, \vartheta)$.

      $\mathit{inv}_1(\varsigma'')$ holds because of $\mathit{inv}_1(\varsigma)$ and because there is an activation token on the stack for all newly deactivated threads. $\mathit{inv}_4(\varsigma'')$ follows from $\mathit{inv}_4(\varsigma)$ for all previously inactive threads and from $\mathit{inv}_3(\varsigma)$ for all previously active threads. inv_6 holds because there are no new tokens pushed onto the stack. We can now complete this case by applying corollary 1 to $\varsigma' = \mathit{unwind}(\varsigma'')$.
    - ($\mathrm{brk}_{\mathrm{div}}$). $pc'_w = pc_w + 1$, $\alpha' = \alpha[wl_1 \ldots wl_n \mapsto \text{inactive}]$, $\delta' = \delta[wl_1 \ldots wl_n \mapsto \text{break}]$, and $\vec{\tau}' = \vec{\tau}$.
        - $\alpha(wl) = \text{active}$. If ϑ remains active after the execution of this step, ϑ does not execute the break instruction meaning that ϑ's guard evaluates to false. Consequently, $\mathit{act\text{-}cons}(\varsigma', \vartheta)$ holds since $pc'_{\vartheta} \overset{(\mathrm{Exec}_{\mathrm{ff},1})}{=} pc_{\vartheta} + 1 \overset{\mathit{act\text{-}cons}(\varsigma, \vartheta)}{=} pc_w + 1 = pc'_w$. On the other hand, if ϑ is deactivated in this step, we can reason analogously to the corresponding case of uniform breaks. Therefore, there is an activation token on the stack that reactivates ϑ if it is popped and $\mathit{inact\text{-}cons}(\varsigma', \vartheta)$ holds.
        - $\alpha(wl) = \text{inactive}$. ϑ cannot become active in this step and $pc'_{\vartheta} = pc_{\vartheta}$. As the stack remains unchanged, $\mathit{inact\text{-}cons}(\varsigma', \vartheta)$ holds because of $\mathit{inact\text{-}cons}(\varsigma, \vartheta)$.

      It remains to be shown that the invariant still holds after the execution of the step:
        - $\mathit{inv}_1(\varsigma')$: Follows from $\mathit{inv}_1(\varsigma)$ and because there is an activation token on the stack for all newly deactivated threads.
        - $\mathit{inv}_2(\varsigma')$: At least one thread must be active, otherwise $\neg \mathit{uni}(\alpha, wl_1 \ldots wl_n)$ would not be true and rule ($\mathrm{brk}_{\mathrm{uni}}$) should have been used instead.
        - $\mathit{inv}_3(\varsigma')$: Follows from $\mathit{inv}_3(\varsigma)$ and because $\alpha' \subseteq \alpha$.
        - $\mathit{inv}_4(\varsigma')$: Follows from $\mathit{inv}_4(\varsigma)$ for all previously inactive threads and from $\mathit{inv}_3(\varsigma)$ for all previously active threads.
        - $\mathit{inv}_5(\varsigma')$: Follows from $\mathit{inv}_5(\varsigma)$ and for all deactivated threads both the active mask is set to inactive and the disable mask is set to break.

    - $inv_6(\varsigma')$: Follows from $inv_6(\varsigma)$, because no new tokens are pushed onto the stack during this step.

- $\overrightarrow{tact_{flow}} = \texttt{return}\ wl_1 \ldots \texttt{return}\ wl_n$. In an analogous manner to the `break` case.

- $\overrightarrow{tact_{flow}} = \texttt{exit}\ wl_1 \ldots \texttt{exit}\ wl_n$. This case is similar to the `break` case, but there is a difference worth mentioning: Active threads that are deactivated in this step do not have an activation token on the stack. Furthermore, their program counter is set to ε, ensuring that the threads cannot be activated again; $pc_w = \varepsilon$ is impossible, so activating a completed thread would be a violation of active consistency. Since inactive consistency and most parts of the invariant do not talk about completed threads, these properties trivially hold.

- $\overrightarrow{tact_{flow}} = \texttt{bra}\ wl_1\ pc_1 \ldots \texttt{bra}\ wl_n\ pc_n$. The rules ($\text{bra}_{\text{uni}}$), ($\text{bra}_{\text{guard}}$), and ($\text{bra}_{\text{target}}$) of transition system $\rightarrow^{\text{CRS}}$ handle branch thread actions depending on whether all active threads take the branch and how many different target addresses there are. The case that no thread takes the branch, resulting in empty thread actions, is already proven above.
  - (bra_{uni}). $pc'_w = pc_1$, $\alpha' = \alpha$, $\delta' = \delta$, $\vec{\tau}' = \vec{\tau}$. $inv(\varsigma')$ trivially holds as the relevant parts of the execution state remain unchanged and no threads are deactivated or reactivated in this step. Thus, $\textit{inact-cons}(\varsigma', \vartheta)$ follows directly from $\textit{inact-cons}(\varsigma, \vartheta)$ if ϑ is inactive. If ϑ is active, function $branch_\Theta$ sets pc'_ϑ to pc_i, if $wl_i = wl$. Since all program addresses $pc_1 \ldots pc_n$ are the same, $\textit{act-cons}(\varsigma', \vartheta)$ follows.
  - ($\text{bra}_{\text{guard}}$). $pc'_w = pc_1$, $\tau = (\text{diverge}, notTaken(\alpha, wl_1\ pc_1 \ldots wl_n\ pc_n, pc_1), pc_w + 1)$, $\vec{\tau}' = \tau :: \vec{\tau}$, $\alpha' = taken(wl_1\ pc_1 \ldots wl_n\ pc_n, pc_1)$, $\delta' = \delta$.
    - $\alpha(wl) = \text{active}$.
      - $\alpha'(wl) = \text{active}$. Since all program addresses $pc_1 \ldots pc_n$ are the same, $pc'_\vartheta \overset{branch_\Theta}{=} pc_1 = pc'_w$ and $\textit{act-cons}(\varsigma', \vartheta)$ holds.
      - $\alpha'(wl) = \text{inactive}$. $pc'_\vartheta = pc_\vartheta + 1$ according to the definition of rule ($\text{Exec}_{\text{ff},1}$). Furthermore, τ is ϑ's activation token because $\tau_\alpha(wl) = \text{active}$ and $\delta(wl) = \delta'(wl) = \text{enabled}$. Consequently, $pc'_\vartheta = pc_\vartheta + 1 \overset{\textit{act-cons}(\varsigma,\vartheta)}{=} pc_w + 1 = \tau_{pc}$ and for this reason $\textit{inact-cons}(\varsigma', \vartheta)$ holds.
    - $\alpha(wl) = \text{inactive}$. τ is not the activation token of a previously inactive thread, because $\tau_\alpha(wl) = \text{inactive}$. $\textit{inact-cons}(\varsigma', \vartheta)$ therefore follows from $\textit{inact-cons}(\varsigma, \vartheta)$.

  It remains to be shown that the invariant still holds after the execution of the step:
    - $inv_1(\varsigma')$: Follows from $inv_1(\varsigma)$ and because there is an activation token on the stack for all newly deactivated threads.
    - $inv_2(\varsigma')$: At least one thread must be active, otherwise $\neg uni(\alpha, wl_1 \ldots wl_n)$ would not be true and rule (bra_{uni}) should have been used instead.
    - $inv_3(\varsigma')$: Follows from $inv_3(\varsigma)$ and because $\tau_{tokenType} = \text{diverge}$.
    - $inv_4(\varsigma')$: Follows from $inv_4(\varsigma)$ and because $\tau_{tokenType} = \text{diverge}$.

        - $\mathit{inv}_5(\varsigma')$: Follows from $\mathit{inv}_5(\varsigma)$, $\delta' = \delta$, and τ is the activation token for all newly deactivated threads.
        - $\mathit{inv}_6(\varsigma')$: Follows from $\mathit{inv}_6(\varsigma)$ and at least one thread must be active in the not-taken mask, otherwise $\neg \mathit{uni}(\alpha, wl_1 \dots wl_n)$ would not be true and rule (bra_{uni}) should have been used instead.
    - ($\text{bra}_{\text{target}}$). $\tau = (\text{diverge}, \mathit{notTaken}(\alpha, wl_1\ pc_1 \dots wl_n\ pc_n, pc_k), pc_w)$, $\vec{\tau}' = \tau :: \vec{\tau}$, $\alpha' = \mathit{taken}(wl_1\ pc_1 \dots wl_n\ pc_n, pc_k)$, $\delta' = \delta$, and $pc'_w = pc_k$ for some $k \in \{1 \dots n\}$.
        - $\alpha(wl) = \text{active}$.
            - $\alpha'(wl) = \text{active}$. $pc'_\vartheta \overset{\mathit{branch}_\ominus}{=} pc_i$ if $wl_i = wl$. Since ϑ remains active, $k = i$ and thus $pc'_\vartheta = pc_i = pc_k = pc'_w$ and $\mathit{act\text{-}cons}(\varsigma', \vartheta)$ holds.
            - $\alpha'(wl) = \text{inactive}$. $pc'_\vartheta = pc_\vartheta$ because of function *reset* used by rule (bra). Furthermore, τ is ϑ's activation token because $\tau_\alpha(wl) = \text{active}$ and $\delta(wl) = \delta'(wl) = \text{enabled}$. Consequently, $pc'_\vartheta = pc_\vartheta \overset{\mathit{act\text{-}cons}(\varsigma, \vartheta)}{=} pc_w = \tau_{pc}$ and for this reason, $\mathit{inact\text{-}cons}(\varsigma', \vartheta)$ holds.
        - $\alpha(wl) = \text{inactive}$. See corresponding ($\text{bra}_{\text{guard}}$) case.

    The invariant holds for the same reasons as shown above for the ($\text{bra}_{\text{guard}}$) case.

- $\overrightarrow{\mathit{tact}_{\mathit{flow}}} = \texttt{call}\ wl_1\ pc_1 \dots \texttt{call}\ wl_n\ pc_n$. The rules ($\text{call}_{\text{uni}}$), ($\text{call}_{\text{guard}}$), and ($\text{call}_{\text{target}}$) of transition system $\rightarrow^{\text{CRS}}$ handle call thread actions. The two cases (call_{uni}) and ($\text{call}_{\text{target}}$) can be proven analogously to the corresponding branch cases above, only the proof of rule ($\text{call}_{\text{guard}}$) works differently.

    Rule ($\text{call}_{\text{guard}}$) modifies the execution state such that $\alpha' = \mathit{taken}(wl_1\ pc_1 \dots wl_n\ pc_n, pc_1)$, $\delta' = \delta$, $\vec{\tau}' = \vec{\tau}$, and $pc'_w = pc_1$.

    - $\alpha(wl) = \text{active}$.
        - $\alpha'(wl) = \text{active}$. Since all program addresses $pc_1 \dots pc_n$ are the same, $pc'_\vartheta \overset{\mathit{branch}_\ominus}{=} pc_1 = pc'_w$ and $\mathit{act\text{-}cons}(\varsigma', \vartheta)$ holds.
        - $\alpha'(wl) = \text{inactive}$. $pc'_\vartheta = pc_\vartheta + 1$ according to the definition of rule ($\text{Exec}_{\text{ff},1}$). $\mathit{stack\text{-}cons}_6$ guarantees that there is a token τ on the stack of type call with $\tau_{pc} = pc_w + 1$. τ is ϑ's activation token because of $\mathit{inv}_3(\varsigma)$. Hence, $pc'_\vartheta = pc_\vartheta + 1 \overset{\mathit{act\text{-}cons}(\varsigma, \vartheta)}{=} pc_w + 1 \overset{\mathit{stack\text{-}cons}_6}{=} \tau_{pc}$ and for this reason $\mathit{inact\text{-}cons}(\varsigma', \vartheta)$ holds.
    - $\alpha(wl) = \text{inactive}$. See corresponding ($\text{bra}_{\text{guard}}$) case.

    It remains to be shown that the invariant still holds after the execution of the step:

    - $\mathit{inv}_1(\varsigma')$: Follows from $\mathit{inv}_1(\varsigma)$ and because there is an activation token on the stack for all newly deactivated threads.
    - $\mathit{inv}_2(\varsigma')$: At least one thread must be active, otherwise $\neg \mathit{uni}(\alpha, wl_1 \dots wl_n)$ would not be true and rule (call_{uni}) should have been used instead.
    - $\mathit{inv}_3(\varsigma')$: Follows from $\mathit{inv}_3(\varsigma)$ and because the stack is not changed.
    - $\mathit{inv}_4(\varsigma')$: Follows from $\mathit{inv}_4(\varsigma)$ and because the stack is not changed.
    - $\mathit{inv}_5(\varsigma')$: Follows from $\mathit{inv}_5(\varsigma)$, $\delta' = \delta$, and τ is the activation token for all newly deactivated threads.

- $inv_6(\varsigma')$: Follows from $inv_6(\varsigma)$ and at least one thread must remain active in the not-taken mask, otherwise $\neg uni(\alpha, wl_1 \ldots wl_n)$ would not be true and rule (call_{uni}) should have been used instead.

This concludes all the cases and hence the proof. □

We formulate the safety property over infinite sequences of execution steps. For this, let Γ_{Ω_s} be the set of all runs of transition system $\rightarrow^{\Omega_s}$, i.e. $\langle w_1, \phi, m_1 \rangle \rightarrow^{\Omega_s} \langle w_2, \phi, m_2 \rangle \rightarrow^{\Omega_s} \ldots \in \Gamma_{\Omega_s}$. Although such a sequence is infinite, finite runs of terminating programs are also in Γ_{Ω_s}; for terminating programs, there is a step i where the warp completes the execution of the program, therefore $w_i = w_j$ for all $j > i$. The memory might still change after step i if there are any unfinished memory operations left.

Corollary 2 (Safety)**:** Let $\langle w_1, \hat{\phi}, m_1 \rangle \rightarrow^{\Omega_s} \langle w_2, \hat{\phi}, m_2 \rangle \rightarrow^{\Omega_s} \ldots \in \Gamma_{\Omega_s}$ be a run of transition system $\rightarrow^{\Omega_s}$ for a control flow consistent program $\hat{\phi}$, an initial warp w_1, and an initial memory configuration m_1. Then:

$$\begin{aligned} &inv(w_{1,\varsigma}) \land \forall \vartheta \in w_{1,\vec{\vartheta}} \,.\, act\text{-}cons(w_{1,\varsigma}, \vartheta) \land inact\text{-}cons(w_{1,\varsigma}, \vartheta) \\ &\qquad\qquad \Rightarrow \forall w_i, \vartheta \in w_{i,\vec{\vartheta}} \,.\, act\text{-}cons(w_{i,\varsigma}, \vartheta) \land inact\text{-}cons(w_{i,\varsigma}, \vartheta) \end{aligned}$$

Proof. Assume that w_1 is active and inactive consistent and that the invariant holds for w_1. Using induction, corollary 2 follows directly from theorem 1. □

6.5 Formalization of the Liveness Property

Program 6.1 clearly shows that the liveness property cannot be proven in the general case. CUDA's branching algorithm has a problem that causes the violation of the liveness property, but investigating possible fixes is outside the scope of this report. In this section, we only give the formal definition of the liveness property and leave the fix to future work.

The safety property already ensures that whenever a thread is active or becomes active, the instruction it executes is the correct one. On the other hand, the safety property does not guarantee that a thread executes all instructions; a thread might only execute all instructions up to some point and then never execute the remaining ones. This might happen when the warp deactivates the thread because of a `bra`, `call`, `break`, or `return` instruction. The safety property guarantees that there is an activation token that reactivates the thread once the token is popped off the stack. However, the safety property does not guarantee that this will ever happen. But if the liveness property were to hold, all tokens pushed onto the stack would eventually be popped from the stack. Referring back to the deadlock problem of program 6.1, the liveness property is violated because the break token pushed onto the stack by the `preBrk` instruction at the beginning of the program is never popped off the stack. Before the token can be popped, all threads must execute the `break` instruction which in this situation is impossible.

Liveness Property: Let $\langle w_1, \phi, m_1 \rangle \rightarrow^{\Omega_s} \langle w_2, \phi, m_2 \rangle \rightarrow^{\Omega_s} \ldots \in \Gamma_{\Omega_s}$ be a run of transition system $\rightarrow^{\Omega_s}$ for a program ϕ, an initial warp w_1, and an initial memory configuration m_1. Then:

$$\forall i \in \mathbb{N} \,.\, \left|w_{i,\varsigma,\vec{\tau}}\right| \neq 0 \Rightarrow \exists j > i \in \mathbb{N} \,.\, \left|w_{j,\varsigma,\vec{\tau}}\right| < \left|w_{i,\varsigma,\vec{\tau}}\right|$$

The liveness property compares the stack sizes of two execution states to enforce that eventually all tokens are popped off the stack. This is possible because in a single step either one token is pushed onto the stack or an arbitrary amount of tokens is popped off the stack. More precisely, it is impossible that tokens are pushed onto the stack and popped off the stack in the same step. Therefore, it is enough to compare the stack sizes; it is not necessary to keep track of the individual tokens on the stack.

In general, the liveness property does not hold; it is violated by CUDA's current branching algorithm as program 6.1 demonstrates. Whether the branching algorithm can be fixed such that it no longer violates the liveness property remains to be seen; even if the branching algorithm can be fixed, however, liveness will still not be guaranteed in the general case: If a program contains a `while (true) {}` statement or infinite recursion, the liveness property is obviously also violated. We are therefore currently preparing a paper where we classify the branching algorithm as an unfair scheduling strategy, as divergent threads are scheduled in an unfair way. Hence, warp execution might not terminate because of unfairness, whereas all fair schedules of the individual thread semantics would terminate.

The liveness property trivially holds for terminating programs. Suppose all threads finish the execution of the program at or before step i. Consequently, the stack size at step i is 0 because the last thread to execute the `exit` instruction causes an unwinding process that removes all tokens from the stack. But if the stack size is 0 for step i, the implication of the liveness property is trivially true.

6.6 Summary

CUDA employs a warp level branching algorithm to handle control flow instructions instead of allowing individual thread execution. While this allows for a more efficient hardware implementation, it can also cause significant performance problems if control flow instructions are used unwisely. More importantly, a program might even deadlock because of the branching algorithm.

It is possible to prove the correctness of the branching algorithm for terminating programs. Even for programs that do not terminate, it can be proven that up to the point where a thread becomes inactive and is never activated again, it executes correctly according to its individual control flow. These two different propositions translate into a safety and a liveness property over sequences of execution steps of a warp. If a program fulfills certain restrictions regarding the placement of control flow instructions within its functions, it is possible to prove the safety property. On the other hand, there are programs that violate the liveness property such that liveness only holds for terminating programs.

Safety and liveness together result in warp level branching semantics equivalent to that of thread level branching. Warp level branching guarantees that "if a thread executes all instructions it must execute (liveness), then it does so correctly (safety)". Thread level branching says that "a thread executes all instructions it must execute and it does so correctly". At least for terminating programs, these two statements are provably equivalent.

7 Conclusion

In this report, we formalized the memory model and the model of computation of Nvidia's CUDA infrastructure for general purpose GPU programming. Our formalization has to be close to the hardware as design decisions made at the hardware level influence the semantics of CUDA programs. Exploring ways to formalize the semantics in a more high-level fashion was outside the scope of this report but should be considered for the future. Also, the complexity of both CUDA and the formalization suggests that tool support is needed in order to be able to prove any non-trivial properties of a CUDA program. For that, future work might strive to formalize the semantics in a tool like KIV[1] and at the same time verify the proof of the warp level branching algorithm presented in this report.

The memory model comprises several different types of memories and caches that have unique performance characteristics and sizes and consequently are used in vastly different scenarios. As the CUDA specification is not very clear about the precise semantics of the memory model, our formalization gives almost no guarantees about the result of a read operation after a write operation to the same location of memory. Additionally, we did not formalize volatile memory requests as well as texture memory because of a lack of proper documentation. We reduced the complexity of the memory model by defining the semantics of a declarative programming language that we in turn used to define the actual semantics of the memory operations.

CUDA's model of computation is considerably different from the traditional programming model of x86 CPUs. This is due to the different architectures of modern GPUs and CPUs which affect the respective programming models. CUDA's thread hierarchy closely reflects the architecture of the hardware in the sense that a thread is executed by a CUDA core, a warp is executed by an execution block of a streaming multiprocessor, a thread block runs on one streaming multiprocessor, a grid runs on a device, and a context allows its grids to communicate using DRAM memory. This close mapping of software concepts to the hardware is one of the reasons that CUDA programs often outperform their CPU counterparts if only the problem at hand is sufficiently parallelizable. Furthermore, by looking at this mapping it becomes apparent why only thread blocks support a synchronization primitive: Threads of the same thread block are executed by the same piece of hardware, hence a synchronization mechanism can be implemented locally causing only little overhead; thus, the incurred performance penalty is kept within limits. Global thread synchronization, however, would require more logic and would most likely be considerably slower. Consequently, global thread synchronization is currently not supported by the hardware; Nvidia has not yet announced whether it will be supported by future GPU generations.

Even though it is not part of the official CUDA specification, we formalized the warp level branching algorithm, i.e. the hardware's SIMT architecture. Threads of the same warp are executed in lockstep for reasons of performance, but after a data-dependant branch instruction control flow of the threads within a warp might diverge. Our formalization

---

[1] `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/`, last access 2010-10-29

explained how a warp handles divergent control flow and how it reliably resynchronizes thread execution. We presented a formal proof that the warp level branching algorithm used by the underlying hardware is correct for terminating programs but also showed that it is not equivalent to individual thread branching in the general case. Hence, CUDA developers have to be careful when writing code that contains control flow instructions whose outcome depends on values computed by other threads. If there are any circular dependencies, threads might deadlock because they are not allowed to branch individually. It remains to be seen whether there is a way to fix CUDA's warp level branching algorithm such that deadlocks are avoided and the algorithm is provably correct under all circumstances.

Future work might add the constant caches to the formalization of the memory model. It might also be worth taking a closer look at the special optimizations the hardware employs to handle textures. As general purpose GPU programming is often used in the context of graphics applications like raytracing or post-processing effects of games, it might be desirable to add support for texture operations to our formalization of the semantics of CUDA programs. In addition to texture fetching, the only other important feature missing in our formalization is generic addressing and all instructions associated with that feature. Other than that, we have included all important PTX features so adding the remaining ones should not take too much effort. However, Nvidia has already released the release candidate of the next version of CUDA which adds dynamic memory allocation and deallocation from within kernels. Some changes might be required for the memory model to support the management of allocated and unused memory. Furthermore, Nvidia stated that the Fermi architecture supports C++ exceptions. As of now, it is unknown how exceptions are handled by the hardware and by PTX, but it will be interesting to see if and how exceptions fit into the formalization presented in this report. The documentation of the next version of CUDA also outlines how warp scheduling of second generation Fermi GPUs differs from our formalization. Apparently, warps can dual issue instructions, i.e. a warp can execute two successive instructions in parallel if there are no dependencies between the two. Since we have abstracted from the scheduling algorithm anyway, it should be possible to adapt our semantics to incorporate the new scheduling behavior. The abstract scheduling function then not only returns the warps that should execute, but also the amount of instructions each scheduled warp should execute. In order to make the most efficient use of the available hardware resources, the PTX compiler has to reorder instructions so that instructions can be dual issued as often as possible. Those compiler optimizations are completely transparent to our formalization however.

There is a lot of interest in CUDA both academically and in the industry. However, with the recent introduction of cross-vendor frameworks like OpenCL and Direct Compute, it seems likely that CUDA's predominance will vanish or at least diminish in favor of the non-vendor specific frameworks. For instance, Folding@Home's recently released GPU3 client already supports OpenCL in addition to CUDA[2]. As Nvidia translates OpenCL programs into PTX before execution, it might be worth exploring if the semantics of OpenCL can be defined upon the formalization of the PTX semantics presented in this report. It will be interesting to see how OpenCL's ignorance of warp level branching affects the formalization of its semantics. Officially, CUDA does not know about warp level branching either, but we chose to circumvent this issue by explicitly handling control flow instructions at the warp level. It might be necessary to do the same in order to formalize the semantics of OpenCL.

---

[2] `http://folding.stanford.edu/English/FAQ-NVIDIA-GPU3`, last access 2010-10-29

Moreover, it should be figured out how a higher-level semantics for CUDA could be defined. It should be possible to define the semantics at the CUDA-C level instead of the PTX level if the memory model abstracts from caches. We only included the L1 caches in our formalization because of the inconsistencies that can arise due to the fact that the L1 caches of different streaming multiprocessors are not kept coherent. Consequently, we needed to know whether a memory operation accessing global or local memory uses the L1 cache — which is only known at the PTX level. However, with the very few guarantees given by the memory model, one approach might be to assume that all reads after writes to the same location return undefined data anyway. Then, the L1 caches have no effect on the formalization and can be omitted, making it possible to define the semantics for CUDA-C. There are some situations, however, where the result is indeed not unpredictable, i.e. no read-after-write hazard occurs: After a synchronization barrier, threads of the same thread block are guaranteed to read the latest data from shared memory. Additionally, after a global memory barrier followed by a synchronization instruction, threads of the same thread block are guaranteed to read the latest data from global memory if the corresponding writes were performed by threads of the same thread block. Writes by threads of other streaming multiprocessors might still be missed because of incoherent L1 cache states. These guarantees might be sufficient to develop a more high-level version of the semantics of CUDA programs. At that point, it might also become viable to define a semantics that also takes the host program and SLI configurations into account. As the CPU and the GPUs are independent processors that operate in parallel, CPU/GPU interactions might affect the correctness of CUDA programs; for instance, the CPU could write to global memory addresses that are currently being used by a kernel executed by one of the GPUs. Additionally, compute capabilities could be formalized too. Since devices of higher compute capability support a strict superset of the features and instructions of devices of lower compute capability, it should be possible to define the semantics of older CUDA versions by removing the unsupported features from the formalization presented in this report. That way, it might be possible to define the formal semantics of previous CUDA versions with little effort.

List of Symbols

$\alpha \in \mathit{ActiveMask}$, 84
$\beta \in \mathit{ThreadBlock}$, 102
$\gamma \in \mathit{Grid}$, 108
$\Delta \in \mathit{Device}$, 115
$\delta \in \mathit{DisableMask}$, 87
$\zeta \in \mathit{Context}$, 110
$\eta \in \mathit{MemEnv}$, 43
$\theta \in \mathit{Thread}$, 79
$\vartheta \in \mathit{Thread}_{pc}$, 125
$\kappa \in \mathit{CallStack}$, 68
$\nu \in \mathit{VarEnv}$, 68
$\xi \in \mathit{ProgConf}$, 69
$\pi \in \mathit{Priority}$, 61
$\varrho \in \mathit{BlockRegLookup}$, 107
$\varrho \in \mathit{ExprRegLookup}$, 76
$\varrho \in \mathit{GridRegLookup}$, 109
$\varrho \in \mathit{ThreadRegLookup}$, 85
$\varsigma \in \mathit{ExecState}$, 88
$\sigma \in \Sigma$, 46
$\sigma_a \in \Sigma_{atomic}$, 48
$\sigma_d \in \Sigma_{device}$, 48
$\sigma_r \in \Sigma_{reg}$, 48
$\sigma_s \in \Sigma_{shared}$, 48
$\tau \in \mathit{ExecToken}$, 87
$\phi \in \mathit{ProgEnv}$, 69
$\chi \in \mathit{BarrierEnv}$, 103
$\omega \in \mathit{Warp}$, 86

$\mathit{ac} \in \mathit{ArrCnt}$, 103
$\mathit{aop} \in \mathit{AtomicOp}$, 67
$\mathit{at} \in \mathit{ActTranslator}$, 129

$b \in \mathit{Byte}$, 33
$\mathit{bact} \in \mathit{Act}_{\mathrm{B}}$, 102
$\mathit{bar} \in \mathit{Barrier}$, 103
$\mathit{barid} \in \mathit{BarrierIdx}$, 80
$\mathit{bid} \in \mathit{BlockIdx}$, 102
$\mathit{br} \in \mathit{BarrierRed}$, 67
$\mathit{bt} \in \mathit{BarrierType}$, 67

$c \in \mathit{Const}$, 64
$\mathit{cact} \in \mathit{Act}_{\mathrm{C}}$, 110
$\mathit{cf} \in \mathit{CacheFunc}$, 53
$\mathit{cid} \in \mathit{ContextIdx}$, 110
$\mathit{cin} \in \mathit{ContextInput}$, 112
$\mathit{cio} \in \mathit{ContextIO}$, 112
$\mathit{cl} \in \mathit{CacheLine}$, 37
$\mathit{comp} \in \mathit{Comparison}$, 66
$\mathit{cop} \in \mathit{CacheOp}$, 67
$\mathit{cout} \in \mathit{ContextOuput}$, 112
$\mathit{cw} \in \mathit{CacheWord}$, 37

$d \in \mathit{DataWord}$, 33
$\mathit{df} \in \mathit{DecompFunc}$, 47
$\mathit{din} \in \mathit{DeviceInput}$, 117
$\mathit{dio} \in \mathit{DeviceIO}$, 117
$\mathit{dout} \in \mathit{DeviceOuput}$, 117

$e \in \mathit{Expr}$, 76
$\mathit{ec} \in \mathit{EvClass}$, 37

$f \in \mathit{FuncName}$, 64

$\mathit{gid} \in \mathit{GridIdx}$, 108

$i \in \mathit{BarInfo}$, 103

$l \in \mathit{Label}$, 64
$\mathit{lo} \in \mathit{LocalMemOffset}$, 78

$m \in \mathit{Mem}$, 130
$\mathit{mbl} \in \mathit{MemBarLevel}$, 66
$\mathit{mid} \in \mathit{MemOpIdx}$, 33
$\mathit{mm} \in \mathit{MemMgr}$, 115

$\mathit{op} \in \mathit{MemOp}$, 61

$p \in \mathit{Predicate}$, 80
$\mathit{pAddr} \in \mathit{PhysMemAddr}$, 33
$\mathit{pc} \in \mathit{ProgAddr}$, 65
$\mathit{pid} \in \mathit{ProcIdx}$, 44

$\mathit{pm} \in \mathit{ProgMap}$, 110
$\mathit{progid} \in \mathit{ProgIdx}$, 69

$r \in \mathit{Reg}$, 64
$\mathit{ro} \in \mathit{RegOffset}$, 78

$s_{\mathrm{B}} \in \mathit{BlockSchedule}$, 109
$s_{\Omega} \in \mathit{WarpSchedule}$, 107
$\mathit{size} \in \mathit{MemOpSize}$, 33
$\mathit{so} \in \mathit{SharedMemOffset}$, 102
$\mathit{ssa} \in \mathit{StateSpace}_{atomic}$, 42
$\mathit{ssd} \in \mathit{StateSpace}_{device}$, 42
$\mathit{state} \in \mathit{WarpState}$, 86
$\mathit{stm} \in \mathit{MemStm}$, 54

$\mathit{tact} \in \mathit{Act}_{\Theta}$, 79
$\mathit{tact}_{bar} \in \mathit{BarrierAct}_{\Theta}$, 81
$\mathit{tact}_{comm} \in \mathit{CommAct}_{\Theta}$, 81
$\mathit{tact}_{flow} \in \mathit{FlowAct}$, 79
$\mathit{tact}_{mem} \in \mathit{MemAct}_{\Theta}$, 80
$\mathit{tact}_{vote} \in \mathit{VoteAct}$, 81
$\mathit{tc} \in \mathit{ThreadCnt}$, 80
$\mathit{tid} \in \mathit{ThreadIdx}$, 78

$v \in \mathit{Var}$, 64
$v_{i/o} \in \mathit{Value}_{in/out}$, 47
$v_{in} \in \mathit{Value}_{in}$, 47
$v_{out} \in \mathit{Value}_{out}$, 47
$\mathit{vm} \in \mathit{VoteMode}$, 66
$\mathit{vms} \in \mathit{VirtMemSpace}$, 110
$\mathit{volatile} \in \mathit{Volatility}$, 67

$w \in \mathit{Warp}_{\vartheta}$, 127
$\mathit{wact} \in \mathit{Act}_{\Omega}$, 86
$\mathit{wact}_{bar} \in \mathit{BarrierAct}_{\Omega}$, 86
$\mathit{wact}_{mem} \in \mathit{MemAct}_{\Omega}$, 86
$\mathit{wl} \in \mathit{WarpLane}$, 78

$\mathit{ya} \in \mathit{YieldAction}$, 46

List of Listings

List of Figures

Bibliography

[1] Mark S. Friedrichs, Peter Eastman, Vishal Vaidyanathan, Mike Houston, Scott Legrand, Adam L. Beberg, Daniel L. Ensign, Christopher M. Bruns, and Vijay S. Pande. Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry*, 30(6):864–872, April 2009.

[2] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors — A Hands-on Approach*. Morgen Kaufmann Publishers, 2010.

[3] Nvidia. *Nvidia CUDA C Programming Guide*, 2010. Version 3.1.1.

[4] Microsoft. *DirectX SDK*, June 2010. `http://www.microsoft.com/downloads/en/details.aspx?FamilyID=3021d52b-514e-41d3-ad02-438a3ba730ba`, last access 2010-10-05.

[5] Nvidia. *Nvidia's Next Generation CUDA Compute Architecture: Fermi*, 2009. Version 1.1, `http://www.nvidia.com/object/IO_89570.html`, last access 2010-9-27, Whitepaper.

[6] Peter N. Glaskowsky. *Nvidia's Fermi: The First Complete GPU Computing Architecture*, 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf`, last access 2010-09-27, Whitepaper.

[7] Nvidia. *Nvidia GF100*, 2010. Version 1.4, `http://www.nvidia.com/object/IO_89569.html`, last access 2010-09-27, Whitepaper.

[8] Nvidia. *Nvidia CUDA Best Practices Guide*, 2010. Version 3.1 Beta.

[9] Nvidia. *PTX: Parallel Thread Execution ISA Version 2.1*, 2010.

[10] Nvidia. *Nvidia CUDA C Programming Guide*, 2010. Version 3.2 Beta.

[11] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *Performance Analysis of Systems and Software (ISPASS), 2010 IEEE International Symposium*, pages 235–246. IEEE Computer Society, 2010.

[12] AMD. *ATI Stream Computing Compute Abstraction Layer*, 2010. Revision 2.01.

[13] Khronos OpenCL Working Group. *The OpenCL Specification*, 2010. OpenCL Version 1.1, Document Revision 36, `http://www.khronos.org/registry/cl/`, last access 2010-10-15.

[14] Nvidia. *OpenCL Programming Guide for the CUDA Architecture*, 2010. Version 3.1 Beta.

[15] Nvidia. *Nvidia OpenCL JumpStart Guide*, 2010. Version 1.0.

[16] Nvidia. *OpenCL Programming Guide for the CUDA Architecture*, 2010. Version 3.1.

[17] AMD. *ATI Stream Computing OpenCL*, 2010. Revision 1.05.

[18] Tianyun Ni. *Direct Compute — Bring GPU Computing to the Mainstream*. Nvidia, 2009. `http://www.nvidia.com/content/GTC/documents/1015_GTC09.pdf`, last access 2010-10-14, Presentation.

[19] Matt Sandy. *DirectCompute Memory Patterns*. Microsoft, 2010. `http://tiny.cc/1ugpc`, last access 2010-10-14, Presentation.

[20] Johan Andersson. *Parallel Graphics in Frostbite — Current and Future*. DICE, 2009. `http://s09.idav.ucdavis.edu/talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf`, last access 2010-10-14, Presentation.

[21] Ian A. Buck, John R. Nickolls, Michael C. Shebanow, and Lars S. Nyland. *Atomic memory operators in a parallel processor*, December 2009. Patent US7627723.

[22] Peter B. Holmqvist, George R. Lynch, Patrick R. Marchand, David B. Glasco, and James Roberts. *Multi-class data cache policies*, May 2010. Patent GB2465474.

[23] Christopher D. S. Donham, John S. Montrym, and Patrick R. Marchand. *Out of order graphics L2 cache*, July 2009. Patent US7565490.

[24] G. Berry. *The Constructive Semantics of Esterel*, 2002. Version 3.0, `http://www-sop.inria.fr/members/Gerard.Berry/`, last access 2010-09-13, Draft Book.

[25] Nvidia. *Nvidia CUDA Reference Manual*, 2010. Version 3.1 Beta.

[26] Brett W. Coon, John R. Nickolls, Lars Nyland, Peter C. Mills, and John Erik Lindholm. *Indirect Function Call Instructions in a Synchronous Parallel Thread Processor*, September 2009. Patents US20090240931 and GB2458556.

[27] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley and Sons, 1999.

[28] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. University of Edinburgh, 2004.

[29] Sarita V. Adve and Kourosh Gharachorloo. *Shared Memory Consistency Models: A Tutorial*. Western Research Laboratory, 1995.

[30] Nvidia. *cuobjdump disassembler User Guide*, August 2010. Not yet publicly released.

[31] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420. IEEE Computer Society, 2007.