On the Correctness of the SIMT Execution Model of GPUs

Axel Habermaier and Alexander Knapp

Institute for Software and Systems Engineering, University of Augsburg
{habermaier,knapp}@informatik.uni-augsburg.de

Abstract. GPUs are becoming a primary resource of computing power. They use a single instruction, multiple threads (SIMT) execution model that executes batches of threads in lockstep. If the control flow of threads within the same batch diverges, the different execution paths are scheduled sequentially; once the control flows reconverge, all threads are executed in lockstep again. Several thread batching mechanisms have been proposed, albeit without establishing their semantic validity or their scheduling properties. To increase the level of confidence in the correctness of GPU-accelerated programs, we formalize the SIMT execution model for a stack-based reconvergence mechanism in an operational semantics and prove its correctness by constructing a simulation between the SIMT semantics and a standard interleaved multi-thread semantics. We also demonstrate that the SIMT execution model produces unfair schedules in some cases. We discuss the problem of unfairness for different batching mechanisms like dynamic warp formation and a stack-less reconvergence strategy.

1 Introduction

Since the introduction of general purpose programming frameworks for *graphics processing units* (GPUs) a few years ago, GPUs are capable of accelerating many other kinds of data-parallel algorithms aside from graphics computations. Speedups of one or two orders of magnitude compared to CPU-based implementations have been achieved for applications in the fields of molecular dynamics, medical imaging, seismic imaging, fluid dynamics, and many others [8, 21]. GPUs are well suited for such massively parallel problems because of their high computational power and memory bandwidth. CPUs, on the other hand, are more optimized for sequential code containing many data-dependent branch instructions. The model of computation of GPUs is therefore unlike the traditional one of CPUs, even though they are converging [17].

GPUs typically launch thousands of threads to execute a data-parallel program. Each thread executes the program on different input data akin to the *single program, multiple data* (SPMD) principle. NVIDIA GPUs based on the FERMI architecture process up to 512 threads in parallel, with thousands more being idle and waiting to be scheduled later on [21]. Instead of executing each thread individually, however, the hardware transparently batches several threads together for improved efficiency. The threads of a batch always execute the same instruction in *lockstep* on a *single instruction, multiple data* (SIMD) unit, i.e., in parallel on different operands [24]. As threads are batched dynamically at runtime, GPUs based on NVIDIA's FERMI architecture or

H. Seidl (Ed.): ESOP 2012, LNCS 7211, pp. 316–335, 2012.
© Springer-Verlag Berlin Heidelberg 2012

on AMD's GRAPHICS CORE NEXT design have no explicit support for SIMD vectors in their instruction sets [16, 23], which distinguishes them from classical SIMD architectures [14]. NVIDIA uses the term *single instruction, multiple threads* (SIMT) to describe the SPMD- and SIMD-like execution model of their hardware [24].

A batch of threads that is executed in lockstep is called a *warp* or *wavefront* by NVIDIA and AMD, respectively. Current NVIDIA hardware always uses a warp size of 32 threads, whereas AMD's GRAPHICS CORE NEXT architecture has a wavefront size of 64 [16, 24]. The SIMT design promises significant performance benefits especially for graphics applications [14], but is in general not well-suited for code that heavily relies on data-dependent control flow instructions: If the control flow of threads of the same warp diverges, execution of the warp is serialized for each unique path, disabling the threads that did not take the path. The hardware must therefore track the threads' activation states and schedule all paths for execution one after another. Once all paths complete, the threads reconverge and proceed in lockstep again. It is desirable to reconverge threads as soon as possible for performance reasons, as otherwise many hardware units remain unused [24]. There are several different mechanisms to handle divergent control flow on a SIMT architecture [4], which differ in thread scheduling and performance; in particular, FERMI uses a *stack-based reconvergence mechanism* based on *immediate post-dominators*, enabling full C++ support on the GPU [5, 20].

The growing complexity of GPU-based software potentially increases the number of software errors, while at the same time the more wide-spread adoption requires a higher level of confidence in program correctness. For formal proofs of correctness, however, not only a precise understanding but also a formal foundation of the underlying SIMT execution model is required. We therefore provide a formal semantics for NVIDIA's stack-based reconvergence mechanism. To keep the formalization concise, it is based on a high-level programming language, disregarding much of NVIDIA's technical environment (a full discussion based on the assembly-level language PTX including the memory model can be found in [9]). We analyze the semantic validity of this execution model by comparing it to a standard interleaved multi-thread semantics: We construct and prove a simulation that demonstrates that each warp execution including serializations on control-flow divergences corresponds to an interleaved multi-thread execution. In this sense, the SIMT execution model is *correct* with respect to the interleaving semantics. However, there is a difference between the two models regarding fairness of execution. In the interleaving model, generally weak fairness is assumed such that every enabled thread will eventually take a step. In contrast, FERMI's stack-based reconvergence mechanism does not guarantee such a fairness condition, rather its *unfair scheduling* of divergent threads prevents some otherwise valid programs from terminating in certain corner cases. We discuss the issue of unfairness for both NVIDIA's mechanism and for alternative SIMT implementations and provide a sufficient criterion for detecting such situations.

Overview. Section 2 gives a brief overview of the hardware architecture and introduces NVIDIA's SIMT execution model with the help of an example. Section 3 presents the formalization of this execution model. The interleaved multi-thread semantics and their simulation of the SIMT behavior are given in Sect. 4. Section 5 then discusses the issue of unfairness and Sect. 6 summarizes our findings and gives an outlook to future work.

2 SIMT Hardware Model

To set the context for the remainder of the paper, we give an overview of the FERMI architecture and its programming model, omitting all graphics-related details. We focus on NVIDIA's *Compute Unified Device Architecture* (CUDA – particularly, batches of threads are referred to as *warps*), as it is representative for other GPU programming frameworks and hardware designs: The underlying principles also apply to *DirectCompute*, the *Open Computing Language* (OPENCL), and AMD's GPUs [1, 13, 16, 22, 24]. Hennessy and Patterson [11] provide a more detailed introduction to CUDA and FERMI and contrast FERMI's design with traditional vector and SIMD architectures.

2.1 Hardware Architecture and Programming Model

CUDA programs launch parallel *compute kernels* on the GPU. Kernels are typically executed by thousands of threads in parallel, organized into a hierarchy: A *grid* represents a set of threads executing the same kernel in a data-parallel fashion similar to the SPMD principle. Grids are divided into *thread blocks*, each of which is assigned to a particular *SIMT core* by the hardware. Threads belonging to the same block communicate via a special on-chip memory on the SIMT core. Threads of different thread blocks share data using the GPU's global memory, which is not guaranteed to be consistent. The SIMT core splits thread blocks into *warps*, which comprise a fixed number of *scalar threads* that are executed in lockstep by an array of *scalar processors* on the SIMT core.

GPUs have multiple SIMT cores that execute warps as SIMD groups on the scalar processors. Typically, there are more warps allocated on a SIMT core than can be executed in parallel. This enables the GPU to hide high-latency memory operations: Instead of using advanced hardware mechanism such as prediction and out-of-order execution, GPUs exploit the massive parallelism of the thread hierarchy by interleaving the execution of different warps; switching between warps incurs no overhead. Different warps are executed independently, hence there is no performance gain or penalty when they are executing common or disjoint code paths.

The focus of this paper is divergence and reconvergence within a warp, so we mainly consider one single warp only; Sect. 5.2 discusses alternative SIMT implementations, some of which take multiple warps into consideration to improve efficiency.

2.2 SIMT Control Flow

SIMT cores execute warps in lockstep, that is, each thread of a warp executes the same instruction in parallel on different operands. In this way, instruction fetch and decoding costs are amortized over all threads of a warp and memory operations performed by threads of the same warp can often be coalesced into fewer memory accesses, which is likely to result in significant performance benefits [14, 24]. To support a wide variety of programs, however, the SIMT cores must be able to deal with diverging control flow that occurs when at least two threads of the same warp take different paths due to a data-dependent control flow instruction. Earlier GPU designs serialized the execution of the remainder of the program once the control flow diverges. Serialization without a reconvergence mechanism results in a low utilization of processing resources for

branch-heavy programs and performance drops by a factor proportional to the warp size in the worst case [19]. Consequently, today's GPU architectures implement mechanisms for reconverging threads once their control flows return to a common path. Our focus in this paper is the formalization of an operational semantics for NVIDIA's stack-based reconvergence mechanism as outlined by Hennessy and Patterson [11] and explained in detail in one of NVIDIA's US patent applications [5]. Neither AMD nor NVIDIA provide any official information on their SIMT implementations.

2.3 Stack-Based SIMT Reconvergence

The main idea of a stack-based reconvergence mechanism is to store information about control flow divergence and reconvergence in a *reconvergence stack*. Possible causes for divergence are branches and loops with conditions depending on thread-specific data as well as loops and function calls with bodies containing data-dependent `break` or `return` statements. Whenever the control flow might diverge, a *token* is pushed onto the reconvergence stack. Tokens store both the *continuation* of a potentially divergent instruction and the threads participating in its execution. Once the control flows reconverge (or the execution of the branch, loop, or function call completes without causing any divergence at all), the topmost token is popped off the stack and the SIMT core uses the information contained in the token to continue the execution of the program.

The reconvergence stack belongs to the *execution state* of a warp, also comprising a *program counter* (PC), an *active mask*, and a *disable mask*. The active mask indicates which threads are active and participate in the execution of the instruction referred to by the warp's PC. Inactive threads do not execute the instruction and their respective scalar processors remain idle. The disable mask records the *disable state* of each thread: b or r indicate that the thread executed a `break` or `return` instruction, respectively, whereas 0 means that the thread is not disabled. Only threads with a disable state of 0 can be reactivated when a token is popped off the stack, thereby guaranteeing the correct handling of nested control flow instructions within a compute kernel.

Each token on the reconvergence stack is of a specific type and comprises an active mask and a program counter. Once a token is popped off the stack, the token's PC indicates the next instruction to be executed by the warp, the active mask determines which threads are activated or deactivated, and the token type affects the update operations performed on the warp's current active and disable masks. The type of a token is either div or sync for branches, brk for loops, or call for function calls. A div token stores all the information required for the execution of the second path of a branch after the first path terminates, while sync and brk tokens mark the *reconvergence points* of branches and `while` loops (i.e., their continuations), respectively. It seems that the reconvergence point of a control flow statement always corresponds to the statement's *immediate post-dominator* [5, 20], which denotes the first instruction that must be executed by all divergent (and still active) threads before they return from the current function. The return address of a function call is stored in the corresponding call token on the reconvergence stack, hence no function call stack is required to store return addresses (such a stack would only be necessary to push and pop function arguments). Tokens of types brk and call are also used to determine all instructions a thread must skip after executing a `break` or `return`.

The execution state of a warp is stored directly on the SIMT core, but older entries of the stack may be spilled to global memory if necessary, making operations on the stack potentially time-consuming. In fact, the disable mask is merely an optimization that eliminates the need to modify the stack when threads are deactivated [5].

Example 1. We demonstrate how NVIDIA's SIMT implementation handles nested control flow instructions by means of Prog. 1's compute kernel `main`; the kernel serves no real purpose other than an illustrative one. Note that when Prog. 1 is run on a GPU, the actual execution is likely to differ from what is described below, as the compiler performs (semantics preserving) control flow optimizations to minimize the overhead incurred by operations on the reconvergence stack; particularly, the compiler will most certainly inline the call to `func`.

Table 1 depicts the evolution of the warp's execution state as the kernel is being executed. The active mask is shown as a bit field with a value of 1 at position n indicating that the thread with id n is active; the disable mask is also given in a similar bit field like fashion. The rightmost column represents the reconvergence stack with the topmost token on the left. Each consecutive pair of rows shows how the instruction pointed to by the PC in between them manipulates the warp's execution state. Figure 1 shows the control flow graph of the `while` loop in function `func` of Prog. 1, highlighting the immediate post-dominators of the loop at line 4 and the branch at line 5.

We assume a warp size of four for illustration purposes, with all four threads being active initially (in situations where the number of threads executing a kernel is not a multiple of the warp size, there are underpopulated warps with idle scalar processors). The global pointer variables `a` and `b` are assumed to be shared arrays of length four, acting as the kernel's input and output parameters; we avoid function parameters and return values to simplify matters, even though they are supported by CUDA, OPENCL, and DirectCompute. We assume that the values of the arrays `a` and `b` are $0, 1, 1, 1$ and $0, 1, 1, 3$ for indices 0 to 3, respectively. For each thread executing the kernel, the thread-local variable `tid` contains the unique id of the thread (0 for the first thread, 1 for the second one, and so on), which is used to index into the arrays.

Execution of the compute kernel begins at line 14 where all threads of the warp call `func` in lockstep. A call token is pushed onto the stack, with its program counter set to the return address of the call, i.e., 15. The token's active mask marks all four threads as active, meaning that all four threads should eventually execute the `return` at line 15.

Execution continues at line 3 where all threads initialize their local variables `i` and `j`. Subsequently, the `while` loop is encountered and a brk token is pushed onto the stack. The loop condition evaluates to false for thread 0, hence the corresponding bit in the active mask is set to 0 and the thread does not participate in the execution of the loop. Next, the remaining threads push a sync token onto the stack because of the `if` instruction at line 5. The token's PC is set to the instruction at line 9, which is the reconvergence point where all threads taking one of the two paths of the branch should be executed in lockstep again. As thread 0 is not active when the `if` instruction is encountered, its bit in the sync token's active mask is not set. The branch statement also causes a div token to be pushed onto the stack, as threads 1 and 2 take the else-path and thread 3 takes the then-path; if no divergence had occurred, the stack would remain unchanged. Execution of both paths is serialized, with current NVIDIA hardware

```
1  __shared int *a, *b;
2  void func() {
3      int i = a[tid], j = b[tid];
4      while (i > 0) {
5          if (j > 2 * i)
6              b[tid] += i;
7          else
8              break;
9          --i;
10     }
11     return;
12 }
13 void main() {
14     func();
15     return;
16 }
```

Program 1. A compute kernel with nested control flow instructions

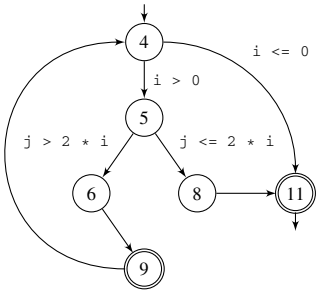

Fig. 1. Control flow graph of the `while` statement in Prog. 1; immediate post-dominators are highlighted

Table 1. Evolution of the warp's execution state during the execution of Prog. 1

PC	Active Mask	Disabled Mask	Top of Stack		
14	1111	0000	–	–	
3	1111	0000	(call, 1111, 15)	–	
4	1111	0000	(call, 1111, 15)	–	
5	0111	0000	(brk, 1111, 11)	(call, 1111, 15)	
8	0110	0000	(div, 0001, 6)	(sync, 0111, 9)	...
6	0001	0bb0	(sync, 0111, 9)	(brk, 1111, 11)	...
9	0001	0bb0	(brk, 1111, 11)	(call, 1111, 15)	
4	0001	0bb0	(brk, 1111, 11)	(call, 1111, 15)	
11	1111	0000	(call, 1111, 15)	–	
15	1111	0000	–	–	
	1111	0000	–	–	

executing the else-path first. Thus, thread 3 is disabled and threads 1 and 2 continue execution at line 8. The div token stores the information required to execute the then-path once the else-path is completed. For this reason, the token's PC is set to point to the instruction of the then-path at line 6 and its active mask has only thread 3 activated.

Threads 1 and 2 execute the `break` statement at line 8. Their bits in the active mask are cleared and their disable states are set to b. Consequently, there no longer are any active threads, so the warp pops the div token off the stack that causes thread 3 to execute the then-path of the branch at line 6. After the execution of the assignment, the end of the then-path is reached, causing the sync token to be popped off the stack. Execution now resumes at line 9; however, the warp cannot just use the token's active mask, as threads 1 and 2 executed a `break` and should therefore not participate in the execution of the loop anymore. To support such arbitrary nested control flow instructions within a compute kernel, the token's active mask is combined with the warp's disable mask, resulting in threads 1 and 2 to remain disabled in this case because their disable states have not yet been reset to 0.

Thread 3 returns to the beginning of the loop at line 4. As the condition evaluates to false, there no longer are any active threads and the warp subsequently pops the brk token off the stack. A token of type brk resets a disable state of b of all threads that are active in the token's active mask. Hence, all four threads execute the `return` instruction at line 11 that causes the threads to jump to the function's return address by popping the call token off the stack. They execute the next `return` statement at line 15, which completes the execution of the compute kernel. □

During the compilation of a compute kernel written in a structured programming language like CUDA-C, branches and loops are replaced by unstructured conditional jumps. The compiler preserves the structural information by adding special flags and instructions into the assembly-level code, allowing the hardware to efficiently determine the immediate post-dominators [5, 23]. Our formalization of the SIMT execution model disregards these low-level implementation details and focuses on a structured programming language instead. A formal semantics of the SIMT behavior based on NVIDIA's assembly-level language PTX can be found in [9].

3 Formalization of the SIMT Execution Model

We formalize the SIMT behavior as discussed in the preceding section in an operational semantics for a C-like high-level language. We focus on the main ideas of the mechanism and omit other hardware supported features such as indirect branches and function calls, as these can be reduced to sequences of their direct counterparts [5]. Due to the stack-based reconvergence mechanism, we base our formalization of the SIMT semantics on an instruction stack that unifies the treatment of statements from the structured programming language and the warp management tokens.

3.1 Basic Domains and Language Grammar

We assume a syntactic category Var of *variable identifiers* with typical element x (all metavariables may occur in an arbitrarily adorned form) and a syntactic category $Func$ of *function identifiers*, of which f will be a typical element. We also assume a syntactic category $Expr$ of (side effect-free arithmetical) *expressions* over Var, ranged over by e, which we do not specify more precisely.

Our programming language is a simple while-language with function calls; we distinguish between *statements*, *statement lists*, and *programs*. The grammar of (C-like) statements and statement lists is as follows:

$$
\begin{aligned}
Stm \ni s ::= {} & \texttt{;} \mid x \texttt{ = } e\texttt{;} \\
& \mid \texttt{if (}e\texttt{) } S_1 \texttt{ else } S_2 \\
& \mid \texttt{while (}e\texttt{) } S \mid \underline{\texttt{while}}\texttt{ (}e\texttt{) } S \mid \texttt{break;} \\
& \mid f\texttt{();} \mid \texttt{return;} \\
Stms \ni S ::= {} & s \mid s\ S
\end{aligned}
$$

Stm does not include a separate sequential composition but relies on statement lists instead. The statements `while` and `while` are required to distinguish between the first iteration of a loop and subsequent iterations, which do not generate further brk tokens. The domain $Prog \subseteq Stms$ of programs, ranged over by P, singles out those statement lists that do not contain `while` and have `break;` only within `while` loops. A *function environment* $\varphi \in FuncEnv = Func \to Prog$ maps each function identifier to a program.

Statement lists are executed by threads θ taken from a domain of *thread identifiers* $Thread$; the domain of all subsets of $Thread$ is ranged over by Θ. Threads can share variables or have their own local copies depending on the initialization of the threads' *variable environments*. A variable environment $\nu \in VarEnv = Var \to Addr$ of a thread assigns an *address* from the domain $Addr$ to each variable in a program. A *thread environment* $\eta \in ThreadEnv = Thread \to VarEnv$ maps a thread to its variable environment. We assume a *memory* type $Mem = Addr \to Val$, ranged over by μ, holding integer values $Val = \mathbb{Z}$ that is accessible by all threads (we disregard caching).

3.2 Warp Configurations and Transitions

The execution state of a warp consists of an *instruction stack* comprising statements and *tokens*, an *active mask*, and a *disable mask*. In contrast to the informal description in Sect. 2.3, our formal model does not maintain a separate reconvergence stack.

A thread is considered active if it is contained in the warp's active mask $\Theta \subseteq Thread$; we write Θ^+ for an active mask with at least one active thread, i.e., $\Theta^+ \neq \emptyset$. A disable mask Δ is defined as a function in $DisableMask = Thread \to DisableState$, which maps a thread to its disable state δ that is either 0, b, or r. *Token types* are denoted by t and have a value of either div, sync, brk, or call. A token $\tau = t_\Theta$ comprises a token type t and an active mask Θ. Warp instruction stacks are given by

$$WStack \ni W ::= \varepsilon \mid s\ W \mid \tau\ W\ ,$$

combining tokens and statements. In contrast to Sect. 2.3, a token's continuation is not given by a program counter but rather as the remainder of the instruction stack.

A *warp configuration* $\varphi, \eta \rhd W, \Theta, \Delta, \mu$ consists of a static and a dynamic part, separated by $\rhd$. The former comprises a function environment φ and a thread environment η, whereas the latter contains a warp instruction stack W, an active mask Θ, a disable mask Δ, and a memory μ. An *initial* warp configuration is of the form $\varphi, \eta \rhd P\ \ \mathsf{call}_\Theta, \Theta, \{\Theta \mapsto 0\}, \mu$ where P is a program, Θ is an arbitrary subset of threads and μ is an arbitrary memory. For reasons of uniformity, we assume that there always is a call token at the bottom of the stack which corresponds to the invocation of the compute kernel. A *warp transition* $\varphi, \eta \rhd W, \Theta, \Delta, \mu \Rightarrow^{\mathrm{w}} W', \Theta', \Delta', \mu'$ describes a step transforming a warp configuration into another warp configuration, not repeating the static parts of the configurations.

3.3 Warp Operational Semantics

The *warp operational semantics* is the smallest binary relation $\Rightarrow^{\mathrm{w}}$ on warp configurations which is closed under the rules in Table 2. These are, in fact, rule schemes where

Table 2. Operational semantics of a warp

$$(\text{skip}^{\text{w}}) \quad ;, \Theta^+ \Rightarrow^{\text{w}} \varepsilon, \Theta^+$$

$$(\text{assign}^{\text{w}}) \quad \eta \rhd x \ = \ e;, \Theta^+, \mu \Rightarrow^{\text{w}} \varepsilon, \Theta^+, \mu\{\eta(\Theta^+)(x) \mapsto \mathcal{E}[\![e]\!]\,\eta(\Theta^+)\,\mu\}$$

$$(\text{if}^{\text{w}}) \quad \eta \rhd \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2, \Theta^+, \mu \Rightarrow^{\text{w}} S_2\ \mathsf{div}_{act(\Theta^+,e,\eta,\mu)}\ S_1\ \mathsf{sync}_{\Theta^+}, \Theta^+ \setminus act(\Theta^+,e,\eta,\mu), \mu$$

$$(\text{while}^{\text{w}}) \quad \eta \rhd \texttt{while (}e\texttt{)}\ S, \Theta^+, \mu \Rightarrow^{\text{w}} S\ \underline{\texttt{while}}\ \texttt{(}e\texttt{)}\ S\ \mathsf{brk}_{\Theta^+}, act(\Theta^+,e,\eta,\mu), \mu$$

$$(\underline{\text{while}}^{\text{w}}) \quad \eta \rhd \underline{\texttt{while}}\ \texttt{(}e\texttt{)}\ S, \Theta^+, \mu \Rightarrow^{\text{w}} S\ \underline{\texttt{while}}\ \texttt{(}e\texttt{)}\ S, act(\Theta^+,e,\eta,\mu), \mu$$

$$(\text{break}^{\text{w}}) \quad \texttt{break;}\ [S]\ \tau, \Theta^+, \Delta \Rightarrow^{\text{w}} \tau, \emptyset, \Delta\{\Theta^+ \mapsto \mathsf{b}\}$$

$$(\text{call}^{\text{w}}) \quad \varphi \rhd f\texttt{();}, \Theta^+ \Rightarrow^{\text{w}} \varphi(f)\ \mathsf{call}_{\Theta^+}, \Theta^+$$

$$(\text{return}^{\text{w}}) \quad \texttt{return;}\ [S]\ \tau, \Theta^+, \Delta \Rightarrow^{\text{w}} \tau, \emptyset, \Delta\{\Theta^+ \mapsto \mathsf{r}\}$$

$$(\text{token}^{\text{w}}) \quad \tau, \Theta_1, \Delta_1 \Rightarrow^{\text{w}} \varepsilon, \Theta_2, \Delta_2 \quad \text{where } (\Theta_2, \Delta_2) = enable(\Delta_1, \tau)$$

$$(\text{inact}^{\text{w}}) \quad S\ \tau, \emptyset \Rightarrow^{\text{w}} \tau, \emptyset$$

warp transitions are obtained by replacing the metavariables with suitable instances. We use the following notational conventions: We only give the initial segment of the instruction stack W that is of relevance for the given rule; the remainder of W is omitted and remains unchanged. Similarly, we drop all other irrelevant parts of an operational judgement that remain unchanged. For example, the rule

$$\varphi, \eta \rhd f\texttt{();}\ W, \Theta^+, \Delta, \mu \Rightarrow^{\text{w}} \varphi(f)\ \mathsf{call}_{\Theta^+}\ W, \Theta^+, \Delta, \mu$$

is abbreviated as follows, where Θ^+ is not omitted (even though it remains unchanged) as it has to be stored in the call token and the rule may only be applied to warp configurations with a non-empty set of active threads:

$$\varphi \rhd f\texttt{();}, \Theta^+ \Rightarrow^{\text{w}} \varphi(f)\ \mathsf{call}_{\Theta^+}, \Theta^+\ .$$

The operational rules in Table 2 process the instruction stack of a warp configuration disregarding all possible compiler or hardware optimizations. The skip operation ; is simply popped off the instruction stack; like all other rules except for (token$^{\text{w}}$) and (inact$^{\text{w}}$), it can only be applied to warp configurations with at least one active thread.

To execute an assignment $x = e;$, all active threads use the function $\mathcal{E}[\![-]\!] : \mathit{Expr} \to \mathit{VarEnv} \times \mathit{Mem} \to \mathit{Val}$ to evaluate e before any of them write to x, thereby avoiding potential nondeterminism. However, the order of conflicting writes (that is, $\eta(\theta_1)(x) = \eta(\theta_2)(x)$ but $\mathcal{E}[\![e]\!]\,\eta(\theta_1)\,\mu \neq \mathcal{E}[\![e]\!]\,\eta(\theta_2)\,\mu$ for two active threads $\theta_1, \theta_2 \in \Theta$) is undefined [24]; this is modeled by the nondeterminism in the instantiation of the rule (assign$^{\text{w}}$): $\mu\{\eta(\Theta^+)(x) \mapsto \mathcal{E}[\![e]\!]\,\eta(\Theta^+)\,\mu\}$ abbreviates the memory update

$$\mu\{\eta(\theta_1)(x) \ldots \eta(\theta_n)(x) \mapsto \mathcal{E}[\![e]\!]\,\eta(\theta_1)\,\mu \ldots \mathcal{E}[\![e]\!]\,\eta(\theta_n)\,\mu\}$$

for some arbitrary order of threads $\theta_i \in \Theta^+$.

An `if` statement pushes two tokens onto the stack: The sync token marks the reconvergence point where all active threads Θ^+ are reactivated again; provided that they

do not execute a `break` or `return` instruction in the meantime. The div token stores the information needed to execute the then-path of the branch once the execution of the else-path is completed. The rule (if$^{\mathrm{w}}$) uses the function

$$act(\Theta, e, \eta, \mu) = \{\theta \in \Theta \mid \mathcal{E}[\![e]\!]\,\eta(\theta)\,\mu \neq 0\}$$

to determine the set of threads for which expression e evaluates to true (i.e. non-zero).

A `while` statement at the top of the instruction stack is replaced by a sequence of instructions: First, a brk token storing the continuation for all threads exiting the loop is pushed onto the stack. Second, a corresponding `while` statement is created, which does not generate another brk token once it is encountered. Finally, the statement list forming the `while` statement's body is pushed onto the stack. Whether there are any threads for which the loop condition holds is again determined by the function act.

A `break` or `return` statement deactivates all active threads and sets their disable state to b or r, respectively. $[S]$ denotes a possibly empty statement list, so either `break` and `return` are directly followed by a token τ or all statements up to the next token on the stack are skipped.

A call to a function f places the program $\varphi(f)$ on the top of the stack. The call token that is pushed onto the stack beforehand stores the currently active threads Θ^+, all of which are reactivated once the warp begins to execute the continuation of the call token.

When a token is popped off the instruction stack, the warp's active and disable masks are updated. The reactivation of an inactive thread depends on the type of the token and the thread's disable state. For instance, a thread with a disable state of r may only be reactivated if the token is of type call. The predicate

$$awaits(\delta, t) \leftrightarrow (\delta = \mathsf{b} \rightarrow t = \mathsf{brk}) \wedge (\delta = \mathsf{r} \rightarrow t = \mathsf{call})$$

establishes this relationship between disable states and token types. The function $enable$ clears the disable states of all threads for which $awaits$ holds. Furthermore, it replaces the warp's active mask with the one of the token, removing all threads with a disable state other than 0. Formally, we define $enable$ as

$$\begin{aligned} enable(\Delta_1, t_\Theta) &= (\{\theta \in \Theta \mid \Delta_2(\theta) = 0\}, \Delta_2) \\ &\quad \text{where } \Delta_2 = \Delta_1\{\{\theta \in \Theta \mid awaits(\Delta_1(\theta), t)\} \mapsto 0\}\,. \end{aligned}$$

Particularly, $enable$ only changes the disable states of threads that are contained in the token's active mask, otherwise threads would be reactivated too soon; this has already been illustrated in Ex. 1. For another example, consider a thread θ that executes a `return` statement in the else-path of a branch, while the other threads Θ of the warp call another function f' when they later execute the if-path of the branch. Once the control flow returns from f', θ's disable state remains unchanged because θ is not active in the call token corresponding to the invocation of f'. Hence, when the sync token of the branch is subsequently popped, θ is removed from the token's active mask before it is copied into the warp's execution state because its disable state is still set to r.

The rule (inact$^{\mathrm{w}}$) is used to skip all statements up to the topmost token on the stack if the warp's active mask is empty. Such a situation typically arises when the condition of a `while` statement evaluates to false for all active threads or when a token does not

activate any threads at all. The latter occurs when all threads return from a function call within an `if`-statement, for example. In that case, the rule $(\text{inact}^{\text{w}})$ skips all statements up to the next token, which is then dealt with by the rule $(\text{token}^{\text{w}})$. Once the last token is popped of the stack, the compute kernel terminates, as the last token on the instruction stack is the call token corresponding to the invocation of the compute kernel.

Example 2. We apply our formal semantics of the SIMT execution model to the compute kernel `main` of Prog. 1. As in Ex. 1, we illustrate active and disable masks as bit fields and omit the memory as well as the function and thread environments for reasons of brevity (observe that our formal model treats indexed array accesses such as `b[tid]` as regular variables; we assume that the thread environment is initialized such that no variables are shared). The following shows the derivation sequence of Prog. 1 beginning with the `while` statement at line 4 and ending just before the execution of the `return` statement at line 11. Again considering only four threads as in Ex. 1, the first warp configuration is given as `while (i > 0) ... return;` $W, 1111, 0000$ with $W = \mathsf{call}_{1111}$ `return;` call_{1111} denoting the remainder of the instruction stack; W contains two call tokens that correspond to the invocations of `func` and `main`. Recall that the values of the shared arrays `a` and `b` are $0, 1, 1, 1$ and $0, 1, 1, 3$ for indices 0 to 3, respectively.

$$
\begin{array}{ll}
 & \texttt{while (i > 0) ... return;}\ W, \quad 1111, 0000 \\
\xrightarrow{\text{while}}^{\text{w}} & \texttt{if (j > 2 * i) b[tid] += i; else break; --i;} \\
 & \quad \underline{\texttt{while}}\ \texttt{(i > 0) ...}\ \mathsf{brk}_{1111}\ \texttt{return;}\ W, \quad 0111, 0000 \\
\xrightarrow{\text{if}}^{\text{w}} & \texttt{break;}\ \mathsf{div}_{0001}\ \texttt{b[tid] += i;}\ \mathsf{sync}_{0111}\ \texttt{--i;} \\
 & \quad \underline{\texttt{while}}\ \texttt{(i > 0) ...}\ \mathsf{brk}_{1111}\ \texttt{return;}\ W, \quad 0110, 0000 \\
\xrightarrow{\text{break}}^{\text{w}} & \mathsf{div}_{0001}\ \texttt{b[tid] += i;}\ \mathsf{sync}_{0111}\ \texttt{--i;} \\
 & \quad \underline{\texttt{while}}\ \texttt{(i > 0) ...}\ \mathsf{brk}_{1111}\ \texttt{return;}\ W, \quad 0000, 0\mathsf{bb}0 \\
\xrightarrow{\text{token}}^{\text{w}} & \texttt{b[tid] += i;}\ \mathsf{sync}_{0111}\ \texttt{--i;} \\
 & \quad \underline{\texttt{while}}\ \texttt{(i > 0) ...}\ \mathsf{brk}_{1111}\ \texttt{return;}\ W, \quad 0001, 0\mathsf{bb}0 \\
\xrightarrow{\text{assign}}^{\text{w}} & \mathsf{sync}_{0111}\ \texttt{--i;}\ \underline{\texttt{while}}\ \texttt{(i > 0) ...}\ \mathsf{brk}_{1111}\ \texttt{return;}\ W, \quad 0001, 0\mathsf{bb}0 \\
\xrightarrow{\text{token}}^{\text{w}} & \texttt{--i;}\ \underline{\texttt{while}}\ \texttt{(i > 0) ...}\ \mathsf{brk}_{1111}\ \texttt{return;}\ W, \quad 0001, 0\mathsf{bb}0 \\
\xrightarrow{\text{assign}}^{\text{w}} & \underline{\texttt{while}}\ \texttt{(i > 0) ...}\ \mathsf{brk}_{1111}\ \texttt{return;}\ W, \quad 0001, 0\mathsf{bb}0 \\
\xrightarrow{\text{while}}^{\text{w}} & \texttt{if (j > 2 * i) b[tid] += i; else break; --i;} \\
 & \quad \underline{\texttt{while}}\ \texttt{(i > 0) ...}\ \mathsf{brk}_{1111}\ \texttt{return;}\ W, \quad 0000, 0\mathsf{bb}0 \\
\xrightarrow{\text{inact}}^{\text{w}} & \mathsf{brk}_{1111}\ \texttt{return;}\ W, \quad 0000, 0\mathsf{bb}0 \\
\xrightarrow{\text{token}}^{\text{w}} & \texttt{return;}\ W, \quad 1111, 0000
\end{array}
$$

The application of the rule $(\underline{\text{while}}^{\text{w}})$ pushes the entire body of the `while` statement onto the instruction stack again, even though there no longer are any active threads. However, the body of a `while` statement consists of statements only, hence no new tokens are pushed onto the stack. The rule $(\text{inact}^{\text{w}})$ is therefore able to skip all statements on the stack up to the brk token. When the rule $(\text{token}^{\text{w}})$ pops the brk token off the stack, the function *enable* reactivates all threads with a disable state of b. In contrast, the threads remain disabled when the div and sync tokens are encountered. □

The following lemma summarizes a few invariants of *reachable* warp configurations. A warp configuration w is reachable if a finite sequence of warp transitions transforms some initial warp configuration into w; a warp transition is reachable if the warp configuration on its left hand side is reachable.

As can be seen by inspecting the operational rules for warps in Table 2, no token intervenes a div token and a sync token and the active mask of a sync token comprises both the currently active threads and the active mask of the previous div token. Similarly, each `while` statement is directly followed by a brk token and the currently active threads are contained in the brk token's active mask. By induction and observing that no warp rule alters previously pushed tokens, the active mask of a token τ_1 is a subset of the active mask of another token τ_2 lower on the stack, if neither token is of type div. Analogously, the warp's current active mask always is a subset of the active masks of all tokens on the stack; except for div tokens, which always disable all active threads. A full proof of this lemma can be found in the accompanying technical report [10].

Lemma 1. *Let $\varphi, \eta \rhd W, \Theta, \Delta, \mu$ be a reachable warp configuration.*

1. *If $W = \mathsf{div}_{\Theta'}\ W_0$, then $W_0 = S\ \mathsf{sync}_{\Theta''}\ W_1$ and $\Theta \cup \Theta' \subseteq \Theta''$.*
2. *If $W = \underline{\texttt{while}}\ (e)\ S\ W_0$, then $W_0 = \mathsf{brk}_{\Theta'}\ W_1$ and $\Theta \subseteq \Theta'$.*
3. *If $W = W_1\ t_{1,\Theta_1}\ W_2\ t_{2,\Theta_2}\ W_3$ and $t_1 \neq \mathsf{div} \neq t_2$, then $\Theta_1 \subseteq \Theta_2$.*
4. *If $W = W_1\ t_{\Theta'}\ W_2$, then $\Theta \cap \Theta' = \emptyset$ if $t = \mathsf{div}$ or $\Theta \subseteq \Theta'$ otherwise.*

4 Simulating SIMT Execution by Interleaved Multi-Threading

The formalization of the SIMT execution model in the preceding section allows us to formally establish its semantic validity by constructing a simulation relation between the warp semantics and a standard interleaved multi-thread semantics. The simulation shows that the SIMT execution model is *correct* in the sense that warps execute control flow instructions in a way that can be reproduced by a certain schedule of the interleaved thread semantics.

4.1 Interleaved Multi-Thread Semantics

The concepts of active masks, disable masks, and divergence do not apply to individual threads. However, threads still depend on an instruction stack that comprises statements and *contexts* c. Similar to the tokens in a warp's instruction stack, contexts denote the thread's continuations once a loop is exited or a `break` or `return` statement is executed. Thread instruction stacks are defined as follows, with contexts c being either brk or call:

$$\mathit{TStack} \ni T ::= \varepsilon \mid s\ T \mid c\ T\ .$$

A *thread configuration* $\varphi, \nu \rhd T, \mu$ consists of a function environment φ, a variable environment ν, a thread instruction stack T, and a memory μ. A *thread transition* $\varphi, \nu \rhd T, \mu \Rightarrow^{\mathrm{t}} T', \mu'$ transforms a thread configuration into another thread configuration.

The rules of our *thread operational semantics* $\Rightarrow^{\mathrm{t}}$ are given in Table 4, where we reuse our notational conventions for warps. It is a fairly standard small-step semantics (see e.g. [26]), so we only remark the following: Contexts that reach the top of the

Table 4. Operational semantics of a single thread

$$(\text{skip}^{\text{t}}) \quad \texttt{;} \Rightarrow^{\text{t}} \varepsilon$$

$$(\text{assign}^{\text{t}}_{\text{rd}}) \quad \nu \rhd x \texttt{ = } e\texttt{;}, \mu \Rightarrow^{\text{t}} x \texttt{ = } v\texttt{;}, \mu \quad \text{where } \mathcal{E}[\![e]\!]\,\nu\,\mu = v$$

$$(\text{assign}^{\text{t}}_{\text{wr}}) \quad \nu \rhd x \texttt{ = } v\texttt{;}, \mu \Rightarrow^{\text{t}} \varepsilon, \mu\{\nu(x) \mapsto v\}$$

$$(\text{if}^{\text{t}}_{\text{tt}}) \quad \nu \rhd \texttt{if (}e\texttt{) } S_1 \texttt{ else } S_2, \mu \Rightarrow^{\text{t}} S_1, \mu \quad \text{if } \mathcal{E}[\![e]\!]\,\nu\,\mu \neq 0$$

$$(\text{if}^{\text{t}}_{\text{ff}}) \quad \nu \rhd \texttt{if (}e\texttt{) } S_1 \texttt{ else } S_2, \mu \Rightarrow^{\text{t}} S_2, \mu \quad \text{if } \mathcal{E}[\![e]\!]\,\nu\,\mu = 0$$

$$(\text{while}^{\text{t}}_{\text{tt}}) \quad \nu \rhd \texttt{while (}e\texttt{) } S, \mu \Rightarrow^{\text{t}} S\ \underline{\texttt{while}}\ \texttt{(}e\texttt{) } S\ \mathsf{brk}, \mu \quad \text{if } \mathcal{E}[\![e]\!]\,\nu\,\mu \neq 0$$

$$(\text{while}^{\text{t}}_{\text{ff}}) \quad \nu \rhd \texttt{while (}e\texttt{) } S, \mu \Rightarrow^{\text{t}} \varepsilon, \mu \quad \text{if } \mathcal{E}[\![e]\!]\,\nu\,\mu = 0$$

$$(\underline{\text{while}}^{\text{t}}_{\text{tt}}) \quad \nu \rhd \underline{\texttt{while}}\ \texttt{(}e\texttt{) } S, \mu \Rightarrow^{\text{t}} S\ \underline{\texttt{while}}\ \texttt{(}e\texttt{) } S, \mu \quad \text{if } \mathcal{E}[\![e]\!]\,\nu\,\mu \neq 0$$

$$(\underline{\text{while}}^{\text{t}}_{\text{ff}}) \quad \nu \rhd \underline{\texttt{while}}\ \texttt{(}e\texttt{) } S, \mu \Rightarrow^{\text{t}} \varepsilon, \mu \quad \text{if } \mathcal{E}[\![e]\!]\,\nu\,\mu = 0$$

$$(\text{break}^{\text{t}}) \quad \texttt{break;}\ [S]\ \mathsf{brk} \Rightarrow^{\text{t}} \varepsilon$$

$$(\text{call}^{\text{t}}) \quad \varphi \rhd f\texttt{ ();} \Rightarrow^{\text{t}} \varphi(f)\ \mathsf{call}$$

$$(\text{return}^{\text{t}}) \quad \texttt{return;}\ T_{\neg\mathsf{call}}\ \mathsf{call} \Rightarrow^{\text{t}} \varepsilon$$

$$(\text{context}^{\text{t}}) \quad c \Rightarrow^{\text{t}} \varepsilon$$

instruction stack are simply discarded by the rule $(\text{context}^{\text{t}})$. When a thread encounters a `break` or `return` statement, it skips all statements up to the next brk or call context on the stack, respectively. $T_{\neg\mathsf{call}}$ denotes a possibly empty list of instructions that does not contain any call contexts.

Multiple threads interleave execution. An *interleaved thread configuration* $\varphi, \eta \rhd \varsigma, \mu$ uses a *thread stack function* $\varsigma : \mathit{Thread} \to \mathit{TStack}$ to determine the instruction stack of each thread participating in the execution. An *interleaved thread transition* $\varphi, \eta \rhd \varsigma, \mu \Rightarrow^{\text{i}} \varsigma', \mu'$ describes a step transforming an interleaved thread configuration into another interleaved thread configuration by selecting an arbitrary thread and executing a thread transition. With the help of our notational conventions for warps, the rule for our interleaved thread semantics $\Rightarrow^{\text{i}}$ is given as:

$$\frac{\varphi, \eta(\theta) \rhd \varsigma(\theta), \mu \Rightarrow^{\text{t}} T, \mu'}{\varphi, \eta \rhd \varsigma, \mu \Rightarrow^{\text{i}} \varsigma\{\theta \mapsto T\}, \mu'}\ .$$

The thread semantics handles assignments in two steps: The rule $(\text{assign}^{\text{t}}_{\text{rd}})$ evaluates the expression before the rule $(\text{assign}^{\text{t}}_{\text{wr}})$ performs the actual memory update; if there was only one rule for assignments, the interleaved thread semantics would be unable to simulate the SIMT execution model. For instance, consider a statement `x = x + 1` for some shared variable `x`. As a warp executes all of its n active threads in lockstep, the value of `x` is incremented by 1 in total. With only one assignment rule for threads, the interleaved thread semantics would always increment `x` by n. With the two rules for assignment, however, the interleaved semantics nondeterministically increments `x` by l with $1 \leq l \leq n$, depending on the order in which the threads apply the rules $(\text{assign}^{\text{t}}_{\text{rd}})$ and $(\text{assign}^{\text{t}}_{\text{wr}})$. The simulation of the SIMT behavior therefore applies the

rule $(\text{assign}^{\text{t}}_{\text{rd}})$ to all active threads before any of them applies the rule $(\text{assign}^{\text{t}}_{\text{wr}})$; in the example, this guarantees that x is incremented by 1.

4.2 Simulation Relation

Figure 2 shows the intended simulation relation between the warp semantics and the interleaved thread semantics: Any reachable warp transition shall be matched by a finite sequence of zero, one, or more interleaved thread transitions (written as $\Rightarrow^{\text{i},*}$). Based on the simulation of one single warp transition, we extend the simulation to sequences of reachable warp transitions, meaning that all warp executions are correct with regard to the interleaved thread semantics.

$$\begin{array}{ccc} \varphi, \eta \rhd \pi_\Theta(W, \Delta), \mu & = = = = \Rightarrow^{\text{i},*} & \pi_{\Theta'}(W', \Delta'), \mu' \\ \uparrow \gamma & & \gamma \uparrow \\ \varphi, \eta \rhd W, \Theta, \Delta, \mu & \Longrightarrow^{\text{w}} & W', \Theta', \Delta', \mu' \end{array}$$

Fig. 2. Simulation of a reachable warp transition by a sequence of interleaved thread transitions

The simulation depends on the mutually recursive *projection* functions $\pi_\theta, \pi_{\neg\theta} : \mathit{WStack} \times \mathit{DisableState} \to \mathit{TStack}$ defined in Table 5 for each $\theta \in \mathit{Thread}$ that transform a warp instruction stack into a thread instruction stack. π_θ projects active threads, whereas $\pi_{\neg\theta}$ is used for inactive ones. The former simply outputs all statements it encounters, replaces all brk and call tokens by brk and call contexts, removes all sync and div tokens as they have no meaning for an individual thread, and deactivates the thread whenever a div token is encountered by calling $\pi_{\neg\theta}$. In the definition of π_θ, the active masks of the tokens on the stack do not have to be considered because of Lem. 1(4). For an inactive thread, $\pi_{\neg\theta}$ skips all instructions until it encounters an *activation token*, i.e., a token that reactivates the thread. The function $\mathit{actToken}$ determines whether a given token is a thread's activation token:

$$\mathit{actToken}_\theta(t_\Theta, \delta) \leftrightarrow \theta \in \Theta \wedge \mathit{awaits}(\delta, t) \; .$$

The projection functions π_θ and $\pi_{\neg\theta}$ are combined into the curried thread stack function $\pi_\Theta : (\mathit{WStack} \times \mathit{DisableMask}) \to \mathit{Thread} \to \mathit{TStack}$ with

$$\pi_\Theta(W, \Delta)(\theta) = \begin{cases} \pi_\theta(W, \Delta(\theta)) & \text{if } \theta \in \Theta \\ \pi_{\neg\theta}(W, \Delta(\theta)) & \text{otherwise} \; , \end{cases}$$

parameterized by Θ, which is used by the conversion function γ of Fig. 2 to turn warp configurations into interleaved thread configurations.

The existence proof for the simulation relation relies on a series of lemmata that relate warp configurations to interleaved thread configurations. The full proofs of the lemmata can be found in the accompanying technical report [10].

The first lemma establishes a relationship between the functions $\mathit{actToken}$ and enable, ensuring that a warp correctly activates inactive threads: It only activates inactive threads when it reaches their activation token and conversely, inactive threads are reactivated once their activation token is encountered.

Table 5. Definitions of the projection functions

$$\pi_\theta(\varepsilon, \delta) = \varepsilon$$

$$\pi_\theta(s\ W, \delta) = s\ \pi_\theta(W, \delta) \qquad \pi_{\neg\theta}(\varepsilon, \delta) = \varepsilon$$

$$\pi_\theta(\mathsf{call}_\Theta\ W, \delta) = \mathsf{call}\ \pi_\theta(W, \delta) \qquad \pi_{\neg\theta}(s\ W, \delta) = \pi_{\neg\theta}(W, \delta)$$

$$\pi_\theta(\mathsf{brk}_\Theta\ W, \delta) = \mathsf{brk}\ \pi_\theta(W, \delta) \qquad \pi_{\neg\theta}(\tau\ W, \delta) = \begin{cases} \pi_\theta(W, 0) & \text{if } actToken_\theta(\tau, \delta) \\ \pi_{\neg\theta}(W, \delta) & \text{otherwise} \end{cases}$$

$$\pi_\theta(\mathsf{sync}_\Theta\ W, \delta) = \pi_\theta(W, \delta)$$

$$\pi_\theta(\mathsf{div}_\Theta\ W, \delta) = \pi_{\neg\theta}(W, \delta)$$

Lemma 2. *Let $\varphi, \eta \rhd W, \Theta, \Delta, \mu$ be a reachable warp configuration, $\theta \notin \Theta$, and τ a token in W. Then $actToken_\theta(\tau, \Delta(\theta))$ is true if and only if $enable(\Delta, \tau) = (\Theta', \Delta')$ with $\theta \in \Theta'$.*

Using Lem. 2, we can show that all inactive threads of a reachable warp configuration simulate an arbitrary operational warp transition by simply remaining idle. This is because the instruction stacks of inactive threads remain unchanged.

Lemma 3. *Let $\varphi, \eta \rhd W, \Theta, \Delta, \mu \Rightarrow^{\mathrm{w}} W', \Theta', \Delta', \mu'$ be a reachable warp transition and let $\theta \notin \Theta$. Then $\pi_\Theta(W, \Delta)(\theta) = \pi_{\Theta'}(W', \Delta')(\theta)$.*

For the active threads in a reachable warp configuration, we would also like to proceed by focusing on a single thread, showing that each single active thread can simulate a warp transition. Assignments, however, are a special case that requires all active threads to be considered, as the order of conflicting writes to the same memory address is undefined. Lemma 4 therefore covers the simulation of assignments separately where $\Rightarrow^{\mathrm{i},+}$ denotes a finite sequence of one or more interleaved thread transitions. Lemma 5 covers the remaining cases focusing solely on one active thread; $\Rightarrow^{\mathrm{t},=}$ denotes zero or one thread transitions.

Lemma 4. *Let $\varphi, \eta \rhd W, \Theta, \Delta, \mu \Rightarrow^{\mathrm{w}} W', \Theta', \Delta', \mu'$ be a reachable warp transition using the rule* $(\mathrm{assign}^{\mathrm{w}})$. *Then $\varphi, \eta \rhd \pi_\Theta(W, \Delta), \mu \Rightarrow^{\mathrm{i},+} \pi_{\Theta'}(W', \Delta'), \mu'$.*

Lemma 5. *Let $\varphi, \eta \rhd W, \Theta, \Delta, \mu \Rightarrow^{\mathrm{w}} W', \Theta', \Delta', \mu'$ be a reachable warp transition not using the rule* $(\mathrm{assign}^{\mathrm{w}})$ *and let $\theta \in \Theta$. Then $\varphi, \eta(\theta) \rhd \pi_\Theta(W, \Delta)(\theta), \mu \Rightarrow^{\mathrm{t},=} \pi_{\Theta'}(W', \Delta')(\theta), \mu'$.*

The combination of Lemmata 3, 4, and 5 proves that there always exists a sequence of interleaved thread transitions to simulate some arbitrary reachable warp transition. This result is summarized in the following proposition, completing the proof of the simulation relation shown in Fig. 2.

Proposition 1. *Let $\varphi, \eta \rhd W, \Theta, \Delta, \mu \Rightarrow^{\mathrm{w}} W', \Theta', \Delta', \mu'$ be a reachable warp transition. Then $\varphi, \eta \rhd \pi_\Theta(W, \Delta), \mu \Rightarrow^{\mathrm{i},*} \pi_{\Theta'}(W', \Delta'), \mu'$.*

The full simulation result follows from Prop. 1 by induction: All sequences of reachable warp transitions can be simulated by sequences of interleaved thread transitions.

Theorem 1 (Simulation of the SIMT Execution Model). *Let* $\varphi, \eta \rhd W, \Theta, \Delta, \mu \Rightarrow^{\mathrm{w},*} W', \Theta', \Delta', \mu'$ *be a sequence of reachable warp transitions. Then* $\varphi, \eta \rhd \pi_{\Theta}(W, \Delta), \mu \Rightarrow^{\mathrm{i},*} \pi_{\Theta'}(W', \Delta'), \mu'$.

From Thm. 1 it follows directly that all threads simulating the execution of a warp terminate once the warp terminates, that is, they have fully executed the program. Lemma 3 shows that the instruction stacks of inactive threads do not change, hence ensuring that inactive threads do not skip any instructions. Additionally, Lemmata 2 and 3 guarantee that inactive threads are not left behind if the warp pops their activation token off the stack. However, the SIMT execution model cannot ensure that all inactive threads will eventually be reactivated, even though the call token at the bottom of the stack is an activation token for all threads: In some cases, there are tokens on the instruction stack that are never reached again; the instruction stack is continuously modified without shrinking below a certain threshold. Theorem 1 holds even for non-terminating programs, hence the interleaved thread semantics is still able to simulate the warp execution. Obvious reasons for non-termination are bugs causing non-terminating loops or infinite recursion; there is, however, a more fundamental problem with the SIMT execution model: unfairness.

5 Unfairness of the SIMT Execution Model

Divergent threads within a warp must be scheduled and executed one after another. Today's GPUs use an *unfair* scheduling strategy in the sense that one of the divergent paths is fully executed before execution of the second path begins; if the first one does not terminate, the second one is not executed at all. For some programs, this unfair scheduling strategy makes it impossible for the warp to eventually terminate, even though in the interleaved semantics all weakly fair schedules terminate (where weak fairness means that no thread is left behind indefinitely).

The SIMT execution model is not part of CUDA's or OPENCL's specification [13, 24]; instead, it is considered an implementation detail that programmers can "essentially ignore" for "the purposes of correctness" according to NVIDIA [24, p. 62]. Our findings in the preceding section support this statement insofar as they formally show that warps execute control flow instructions as if each thread executed them individually in some schedule. The correctness of the SIMT execution model (in the sense of simulatability) is therefore unaffected by fairness considerations. On the other hand, unfairness potentially affects program termination and thus *program correctness*, which may be the reason for the qualifying "essentially" in NVIDIA's statement.

5.1 Programs Affected by the Unfairness Problem

We first illustrate the problem of unfairness with two example programs before discussing it more generally: Suppose that in Prog. 2, the variable `lock` is shared among all threads of the warp with an initial value of 0, whereas `tid` stores the id of each thread. Execution of the program terminates if the interleaved thread semantics chooses a fair schedule; namely, it eventually executes the thread with id 0, causing the condition of the loop to evaluate to false for all other threads, which then terminate. A warp, on the other hand, schedules the else-path before allowing thread 0 to set `lock` to 1.

The loop therefore never terminates, thereby preventing the program from terminating. If the hardware were to execute the if-path first or if the conditional statement were reversed, the program would successfully terminate.

Program 3 (with `lock` and `tid` as above) also does not terminate when executed by a warp, although the unfairness has a different cause in this case. As the warp first encounters the loop, the condition evaluates to true for all threads except for thread 0. As the warp chooses the immediate post-dominator of the loop as the reconvergence point, thread 0 is not allowed to continue execution. Instead, the warp continuously executes the remaining threads, which never leave the loop as thread 0 never increments the value of `lock`. Again, a fair interleaved schedule would eventually allow each thread to increment `lock`, resulting in a successful termination of the program.

Programs affected by the unfairness problem are uncommon in practice. Particularly, Prog. 2 uses shared variables without any means of synchronization in both paths of an `if` statement, which is generally considered bad programming practice. Even if the code was not affected by the unfairness problem, it exploits the implicit knowledge about the sequential execution of the paths and might therefore break on future hardware if this assumption is no longer valid. Busy-loops like the one in Prog. 3 are often used in an attempt to implement global synchronization mechanisms that all NVIDIA GPUs are currently lacking [24]. Global synchronization is in fact impossible to implement, though not because of unfairness issues: A compute kernel might be executed by more threads than the hardware is capable of allocating concurrently, hence threads at the synchronization point might be waiting for threads that do not even exist yet and cannot be allocated, resulting in a deadlock.

The following lemma provides a sufficient criterion for programs that are unaffected by the unfairness problem. It is based on a new kind of thread transition $\rightsquigarrow^{\mathrm{t}}_{\eta}$ defined as

$$\frac{\varphi, \eta(\theta) \rhd T, \mu \Rightarrow^{\mathrm{t}} T', \mu'}{\varphi, \eta(\theta) \rhd T, \mu \rightsquigarrow^{\mathrm{t}}_{\eta} T', \mu''} \text{ where } \forall a \in \mathit{Addr}\,.\, a \notin \mathit{saddr}(\eta) \rightarrow \mu''(a) = \mu'(a)$$

with $\mathit{saddr}(\eta)$ denoting the set of addresses shared among the threads of thread environment η. Such a thread transition makes arbitrary changes to the contents of all shared addresses. If all possible sequences of $\rightsquigarrow^{\mathrm{t}}_{\eta}$ transitions terminate for all threads, the warp execution is guaranteed to terminate as well. Assume for a contradiction that the warp execution does not terminate. Then by Thm. 1 there is an infinite sequence of $\Rightarrow^{\mathrm{t}}$ transitions with a corresponding infinite sequence of $\rightsquigarrow^{\mathrm{t}}_{\eta}$ for at least one thread.

Lemma 6. *Let $\varphi, \eta \rhd P$ $\mathsf{call}_{\Theta}, \Theta, \{\Theta \mapsto 0\}, \mu_0$ be an initial warp configuration. If there is no infinite sequence $\varphi, \eta(\theta) \rhd P$ $\mathsf{call}, \mu_0 \rightsquigarrow^{\mathrm{t}}_{\eta} T_1, \mu_1 \rightsquigarrow^{\mathrm{t}}_{\eta} T_2, \mu_2 \rightsquigarrow^{\mathrm{t}}_{\eta} \ldots$ for all $\theta \in \Theta$, then $\varphi, \eta \rhd P$ $\mathsf{call}_{\Theta}, \Theta, \{\Theta \mapsto 0\}, \mu_0 \Rightarrow^{\mathrm{w},*} \varepsilon, \Theta', \Delta', \mu'$.*

Lemma 6 is only a sufficient condition for warp termination, but not a necessary one as exemplified by Prog. 4. Assuming that the shared variable `next` is initialized to 0, `tid` stores each thread's id, and the warp size is 32, the warp execution terminates: The condition of the `if` statement eventually evaluates to true for all threads, so `next` is equal to 32 at some point and the loop terminates. By contrast, a sequence of $\rightsquigarrow^{\mathrm{t}}_{\eta}$ transitions that always resets `next` to 0 obviously never terminates.

```
if (tid == 0)
  lock = 1;
else
  while (lock != 1) {}
```

Program 2. Unfair scheduling of divergent branches

```
while (lock != tid) {}
// ...
++lock;
```

Program 3. Reconverging at the immediate post-dominator results in unfair schedules

```
while (next != 32) {
  if (tid == next)
    ++next;
}
```

Program 4. Lem. 6 is not a necessary condition for warp termination

In practice, however, infinite sequences of $\rightsquigarrow^{\mathrm{t}}_{\eta}$ transitions are rarely caused by shared variables: Most compute kernels do not use shared variables in a way that affects loops or recursion and graphics APIs have only recently introduced shared variables or atomic operations for some shader types [12, 22].

5.2 Unfairness of Alternative SIMT Execution Models

Several alternative implementations of the SIMT execution model have been proposed, be it for performance reasons or generality [4, 6, 7, 18]. A stack-less approach, for instance, replaces the warp's reconvergence stack by a set of program counters, one for each thread and updated appropriately, that the warp uses to handle reconvergence. We formalize this stack-less warp semantics $\rightrightarrows^{\mathrm{w}}$ as follows, where the abstract function $schedule : (Thread \to TStack) \to 2^{Thread}$ selects a set of threads with the same PC, that is, a set of threads for which $\forall \theta, \theta' \in schedule(\varsigma)\,.\,\varsigma(\theta) = \varsigma(\theta')$ holds:

$$\frac{(\varphi, \eta(\theta) \rhd \varsigma(\theta), \mu \Rightarrow^{\mathrm{t}} T'_\theta, \mu'_\theta)_{\theta \in \Theta}}{\varphi, \eta \rhd \varsigma, \mu \rightrightarrows^{\mathrm{w}} \varsigma\{(\theta \mapsto T'_\theta)_{\theta \in \Theta}\}, \mu\{(a \mapsto \mu'_\theta(a))_{\theta \in \Theta, a \in Addr}\}} \quad \text{where } schedule(\varsigma) = \Theta$$

Collange [4] suggests a similar stack-less approach. By contrast, however, our stack-less semantics $\rightrightarrows^{\mathrm{w}}$ does not consider (function call) stack pointers when selecting the next PC to execute, as that only affects performance but does not influence correctness or fairness. As reconvergence is based on equality of program counters, the fairness of $\rightrightarrows^{\mathrm{w}}$ and Collange's approach depends on the fairness of the choice function $schedule$. Particularly, Collange's lowest program counter scheduling policy makes the overall mechanism unfair.

Fung et al. propose a stack-less technique for more than one warp: *dynamic warp formation* [7]. The SIMT core dynamically regroups all of its threads with the same PC into one or more warps. Fairness depends on the warp scheduling policy; of the five suggested policies, only DTime selecting the oldest warp is generally fair. *Thread block compaction* [6] is another approach proposed by Fung et al. It reintroduces the reconvergence stack, albeit at the thread block level. The stack is used for block-wide synchronization at divergent branches and reconvergence points; divergent warps are regrouped into non-divergent ones, restoring the original warp groupings upon encountering the reconvergence point. Due to the synchronization, thread block compaction suffers from the unfairness problem.

Meng et al. [18] propose *dynamic warp subdivision* where warps are dynamically subdivided on branch (or memory) divergence. Each so-called *warp-split* is individually

scheduled, therefore execution of divergent paths is interleaved. Additionally, threads might reconverge at some point past the immediate post-dominator, reuniting the warp-splits. Their approach consequently has the potential of solving the unfairness problem; in practice, however, unfairness is still an issue as warp subdivision is only allowed on statically determined "appropriate" branches in order to avoid undesirable *over-subdivision*.

6 Conclusions and Future Work

The single instruction, multiple threads execution model used by today's GPUs groups threads into batches that execute a compute kernel in lockstep, requiring a special mechanism to efficiently and correctly handle divergent control flow. Our formalization of the SIMT execution model allows us to prove its correctness in the sense that each SIMT execution corresponds to a standard interleaved multi-thread execution for some schedule. SIMT execution potentially affects program termination, however, as divergent threads are scheduled in an unfair way. Some alternative implementations of the SIMT execution model also exhibit this unwanted behavior.

As more and more GPU-accelerated algorithms are used in safety- or security-critical applications such as medical imaging [8, 21], the importance of formally verified program correctness increases. In particular, GPUs are capable of accelerating model checking algorithms that in turn are used in formal analyses of various problems in a wide range of application domains [2, 3]. Our work establishes the semantic validity of the underlying SIMT execution model, contributing to the development of formal verification tools and mechanisms for GPU-based applications. We plan to use a theorem prover to verify correctness and other properties of GPU-based programs.

Several research papers propose changes to the SIMT execution model in order to improve efficiency and performance. While the main point of interest is performance for the time being, new mechanisms should also explore the possibilities of solving the unfairness problem to avoid unexpected non-termination, especially since the current trend is the unification of the CPU and GPU programming models: For example, the CUDA 4.1 compiler is based on the LLVM compiler infrastructure [15] with the intention of allowing CUDA programs to run on either the GPU or the CPU [25]. In order to make the verification of program correctness independent of the execution model, we plan to study stronger criteria for the preservation of termination and other liveness properties when the underlying hardware uses the SIMT execution model instead of a weakly fair multi-thread semantics. Furthermore, it might be worthwhile to check whether our findings can be generalized to the SIMD execution models found in some contemporary CPU architectures.

References

1. AMD. Evergreen Family Instruction Set Architecture, Reference Guide (2011)
2. Barnat, J., Brim, L., Ceska, M., Lamr, T.: CUDA Accelerated LTL Model Checking. In: Proc. 15th Int. Conf. Parallel and Distributed Systems (ICPADS 2009), pp. 34–41 (2009)

3. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In: Proc. 9th Int. Wsh. Parallel and Distributed Methods in Verification (PDMV 2010), pp. 17–19 (2010)
4. Collange, S.: Stack-less SIMT Reconvergence at Low Cost. Technical Report HAL-00622654, INRIA (2011)
5. Coon, B.W., Nickolls, J.R., Nyland, L., Mills, P.C., Lindholm, J.E.: Indirect Function Call Instructions in a Synchronous Parallel Thread Processor, United States Patent Application #2009/0240931 (2009)
6. Fung, W.W.L., Aamodt, T.M.: Thread Block Compaction for Efficient SIMT Control Flow. In: Proc. 17th IEEE Int. Symp. High Performance Computer Architecture (HPCA 2011), pp. 25–36 (2011)
7. Fung, W.W.L., Sham, I., Yuan, G., Aamodt, T.M.: Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In: Proc. 40th Ann. IEEE/ACM Int. Symp. Microarchitecture (MICRO 2007), pp. 407–420 (2007)
8. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel Computing Experiences with CUDA. IEEE Micro 28(4), 13–27 (2008)
9. Habermaier, A.: The Model of Computation of CUDA and its Formal Semantics. Technical Report 2011-14, University of Augsburg (2011)
10. Habermaier, A., Knapp, A.: On the Correctness of the SIMT Execution Model of GPUs. Technical Report 2012-1, University of Augsburg (2012)
11. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 5th edn. Elsevier Science & Technology (2011)
12. Khronos Group Inc. The OpenGL Shading Language 4.20, Revision 6 (2011)
13. Khronos OpenCL Working Group. The OpenCL Specification 1.2, Revision 15 (2011)
14. Levinthal, A., Porter, T.: Chap – A SIMD Graphics Processor. SIGGRAPH Comput. Graph. 18, 77–82 (1984)
15. The LLVM Compiler Infrastructure, `http://www.llvm.org/` (01/04/2012)
16. Mantor, M., Houston, M.: AMD Graphic Core Next: Low Power High Performance Graphics & Parallel Compute. Presentation at the AMD Fusion Developer Summit (2011)
17. Mark, W.: Future Graphics Architectures. ACM Queue 6, 54–64 (2008)
18. Meng, J., Tarjan, D., Skadron, K.: Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In: Proc. 37th Ann. Int. Symp. Computer Architecture (ISCA 2010), pp. 235–246 (2010)
19. Moy, S., Lindholm, J.E.: Method and System for Programmable Pipelined Graphics Processing with Branching Instructions, United States Patent #6,947,047 (2005)
20. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc. (1997)
21. Nickolls, J.R., Dally, W.: The GPU Computing Era. IEEE Micro 30(2), 56–69 (2010)
22. NVIDIA. DirectCompute Programming Guide 3.2 (2010)
23. NVIDIA. cuobjdump. CUDA Toolkit 4.1 (2011)
24. NVIDIA. NVIDIA CUDA C Programming Guide 4.1 (2011)
25. NVIDIA. NVIDIA Opens Up CUDA Platform by Releasing Compiler Source Code (2011), `http://tiny.cc/NvidiaLLVM` (01/04/2012)
26. Reynolds, J.C.: Theories of Programming Languages. Cambridge University Press (1998)