Modeling and Analyzing Evaluation Cost of CUDA Kernels

STEFAN K. MULLER, Illinois Institute of Technology, Chicago, USA
JAN HOFFMANN, Carnegie Mellon University, Pittsburgh, USA

Motivated by the increasing importance of general-purpose Graphic Processing Units (GPGPU) programming, exemplified by NVIDIA's CUDA framework, as well as the difficulty, especially for novice programmers, of reasoning about performance in GPGPU kernels, we introduce a novel quantitative program logic for CUDA kernels. The logic allows programmers to reason about both functional correctness and resource usage of CUDA kernels, paying particular attention to a set of common but CUDA-specific performance bottlenecks: warp divergences, uncoalesced memory accesses, and bank conflicts. The logic is proved sound with respect to a novel operational cost semantics for CUDA kernels. The semantics, logic, and soundness proofs are formalized in Coq. An inference algorithm based on LP solving automatically synthesizes symbolic resource bounds by generating derivations in the logic. This algorithm is the basis of RaCUDA, an end-to-end resource-analysis tool for kernels, which has been implemented using an existing resource-analysis tool for imperative programs. An experimental evaluation on a suite of benchmarks shows that the analysis is effective in aiding the detection of performance bugs in CUDA kernels.

CCS Concepts: • **Theory of computation** → Parallel computing models; **Program analysis**; • **Software and its engineering** → **Software performance;**

Additional Key Words and Phrases: Resource-aware type system, performance analysis, CUDA, thread-level parallelism, program logics

ACM Reference Format:
Stefan K. Muller and Jan Hoffmann. 2024. Modeling and Analyzing Evaluation Cost of CUDA Kernels. *ACM Trans. Parallel Comput.* 11, 1, Article 5 (March 2024), 53 pages. https://doi.org/10.1145/3639403

1 INTRODUCTION

Many of today's computational problems, such as training a neural network or processing images, are massively data-parallel: many steps in these algorithms involve applying similar arithmetic or logical transformations to a large, possibly multi-dimensional, vector of data. Such algorithms are naturally suitable for execution on **Graphics Processing Units (GPUs)**, which consist of thousands of processing units designed for vector operations. Because of this synergy, **general-purpose GPU (GPGPU)** programming has become increasingly mainstream. With the rise of GPGPU programming has come tools and languages designed to enable this form of programming.

This article is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-18-C-0092 and by the National Science Foundation under SaTC Award No. 1801369, CAREER Award No. 1845514, and SHF Awards No. 1812876 and No. 2007784. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.
Authors' addresses: S. K. Muller, Illinois Institute of Technology, Computer Science Department, 10 W 35th Street, Chicago, IL, 60616; e-mail: smuller2@iit.edu; J. Hoffmann, Carnegie Mellon University, Computer Science Department, 5000 Forbes Avenue, Pittsburgh, PA, 15232; e-mail: jhoffmann@cmu.edu.

ACM 2329-4949/2024/03-ART5
https://doi.org/10.1145/3639403

Possibly the best-known such tool is CUDA, a platform for enabling general-purpose programs to run on NVIDIA GPUs. Among other features, CUDA provides an extension to C that allows programmers to write specialized functions, called *kernels*, for execution on the GPU. The language for writing kernels, called CUDA C or just CUDA, is very similar to C, enabling easy adoption by developers.

Nevertheless, writing a kernel that executes efficiently on a GPU is not as simple as writing a C function: small changes to a kernel, which might be inconsequential for sequential CPU code, can have drastic impact on its performance. The CUDA C Programming Guide [NVIDIA Corporation 2019] lists three particularly pervasive performance bottlenecks to avoid: *divergent warps*, *uncoalesced memory accesses*, and *shared memory bank conflicts*. Divergent warps result from CUDA's execution model: a group of threads (often 32 threads, referred to as a *warp*) execute the same instruction on possibly different data. C functions, however, can perform arbitrary branching that can cause different threads of a warp to *diverge*, i.e., take different branches. CUDA is able to compile such code and execute it on a GPU, but at a fairly steep performance cost, as the two branches must be executed sequentially. Even if a conditional only has one branch, there is nontrivial overhead associated with divergence [Bialas and Strzelecki 2016]. The other two bottlenecks have to do with the CUDA memory model and will be discussed in detail in Section 2.

A number of static [Alur et al. 2017; Cogumbreiro et al. 2021; Li and Gopalakrishnan 2010; Li et al. 2012; Liew et al. 2022; Pereira et al. 2016; Singhania 2018; Zheng et al. 2011] and dynamic [Boyer et al. 2008; Eizenberg et al. 2017; Peng et al. 2018; Wu et al. 2019] tools, including several profiling tools distributed with CUDA, aim to help programmers identify performance bottlenecks such as the three mentioned above. However, all of these tools merely point out potential performance bugs, and occasionally estimate the frequency at which such a bug might occur. Such an analysis cannot guarantee the absence of bugs and gives only a partial picture of the performance impacts. For example, it is not sufficient to simply profile the number of diverging conditionals, because it can be an optimization to factor out equivalent code in the two branches of a diverging conditional, resulting in two diverging conditionals but less overall sequentialization [Han and Abdelrahman 2011]. In addition, there are strong reasons to prefer sound static analyses over dynamic analyses or profiling, particularly in applications (e.g., real-time machine learning systems such as those deployed in autonomous vehicles) where input instances that cause the system's performance to degrade outside expected bounds can be dangerous. Such instances are not merely hypothetical: recent work [Shumailov et al. 2020] has developed ways of crafting so-called *sponge examples* that can exploit hidden performance bugs in neural networks to heavily increase energy and time consumption. For such systems, a static worst-case bound on resource usage could ensure safety.

In this article, we present an **automated amortized resource analysis (AARA)** [Hoffmann et al. 2011; Hofmann and Jost 2003] that statically analyzes the resource usage of CUDA kernels and derives worst-case bounds that are polynomials in the integer inputs of a kernel. Our analysis is parametric over a *resource metric* that specifies the abstract cost of certain operations or events. In general, a resource metric assigns a non-negative rational number to an operation that is an upper-bound on the actual cost of that operation regardless of context. The assigned cost can depend on runtime parameters, which have to be approximated in a static resource analysis. For example, one metric we consider, "sectors," estimates the number of reads and writes of memory. In CUDA, fixed-size blocks of memory read and written by a single warp can be handled as one hardware operation, so the number of such operations depends on the footprint of memory locations accessed by a warp (which we estimate using a specialized abstract interpretation) and the amount of memory accessed in a single operation (which is a hardware-specific parameter).

The main challenge of reasoning statically about CUDA programs is that reasoning about the potential values of variables is central to most static analysis techniques; in CUDA, every program

variable has potentially thousands of copies, one for each thread. Reasoning about the contents of variables then requires (1) reasoning independently about each thread, which is difficult to scale, or (2) reasoning statically about which threads are active at each point in a program. Some existing program logics for CUDA (for example, that of Kojima and Igarashi [2017]) take the latter approach, but these are difficult to prove sound and not very amenable to automated inference. We take a different approach and develop a novel program logic for CUDA that is sound and designed with automated inference in mind. In addition, the logic is *quantitative*, allowing us to use it to reason about both functional and resource-usage properties of CUDA programs simultaneously.

We formalize our program logic in a core calculus miniCUDA that models a subset of CUDA sufficient to expose the three performance bottlenecks listed above. The calculus is equipped with a novel cost semantics that formalizes the execution cost of a kernel under a given resource metric. A soundness theorem then shows that bounds derived with the program logic are sound with respect to this cost semantics. All of these results are *formalized in the Coq Proof Assistant*, and the Coq proof scripts are publicly available [Muller and Hoffmann 2023].

To automate the reasoning in our program logic, we have implemented the resource analysis tool **RaCuda (Resource-aware CUDA)**. If provided with the C implementation of a CUDA kernel and a resource metric, then RaCuda automatically derives a symbolic upper bound on the execution cost of the kernel as specified by the metric. We have implemented RaCuda on top of Absynth [Carbonneaux et al. 2017; Ngo et al. 2018], a resource analysis tool for imperative programs.

Using the aforementioned metrics, we evaluated RaCuda for precision and performance on a number of CUDA kernels derived from various sources including prior work and sample code distributed with CUDA. The evaluation shows our tool to be useful in identifying the presence *and quantifying the impact* of performance bottlenecks on CUDA kernels. RaCuda also shows promise as a tool for novice and intermediate CUDA programmers to debug the performance of kernels.

The features of the analysis described so far are sufficient to analyze properties of a kernel such as numbers of divergent warps or memory accesses. Analyzing the execution time of a kernel requires more care, because execution time does not compose in a straightforward way: Threads are scheduled onto processors by a (deliberately) underspecified scheduling algorithm, which exploits *thread-level parallelism* of kernels to hide the latency of operations such as memory accesses. We do not aim to develop a precise analysis of execution time in this work (even for CPUs where such Worst-case Execution Time analyses are well-studied, this is a separate and rich area of research). However, we do take steps toward such an analysis by showing how our analysis can compute the *work* and *span* of kernels, two metrics derived from the literature on parallel scheduling theory [Blelloch and Greiner 1995, 1996; Brent 1974; Eager et al. 1989; Muller and Acar 2016] that can be used to abstractly approximate the running time of parallel algorithms.

We next validate the work-span model by way of a *bounded implementation* [Blelloch and Greiner 1995, 1996]. Our bounded implementation is a small-step operational semantics for mini-CUDA, which more closely models the parallel execution of threads on a GPU. We show that the number of execution steps of a program in this model is approximated by the bounds derived from the work-span cost semantics.

The contributions of this article include:

— A core calculus miniCUDA, for CUDA kernels, equipped with a type system and a proof of type safety.
— Two sets of operational cost semantics for miniCUDA, one for the *lock-step* evaluation of individual warps and one for the *parallel* evaluation of many warps. Both formalize the execution of kernels on a GPU under a given resource metric.
— A small-step operational semantics for the core calculus, which more closely models low-level details of GPU execution. This semantics serves as a *bounded implementation* for the

parallel cost semantics, as we show that the parallel cost semantics accurately models the lower-level small step execution.
— A novel Hoare-style logic for qualitative and quantitative properties of miniCUDA.
— A Coq formalization of the cost semantics and soundness proofs of the program logic.
— An analysis tool RaCuda that can parse kernels written in a sizable subset of CUDA C and analyze them with respect to resource metrics such as number of bank conflicts and number of divergent warps.
— An empirical evaluation of our analysis tools on a suite of CUDA kernels.

With respect to the conference version of this article [Muller and Hoffmann 2021], the primary new contributions are the full presentation of the type system and progress and preservation proofs for miniCUDA, and the bounded implementation. The latter is a key contribution as it validates the presented cost semantics with respect to a low-level model of GPU execution. Without this validation, the cost assigned to a program by the cost semantics is based on the authors' understanding of CUDA execution at a high level of abstraction, and was previously validated empirically but not formally. All of these results are formalized in the Coq proof assistant. In addition, we provide additional rules and proof details that were omitted from the conference version for space reasons.

The remainder of this article is organized as follows. We begin with an introduction to the features of CUDA that will be relevant to this article (Section 2). In Section 3, we introduce the miniCUDA calculus and the lock-step cost semantics. We use the latter to prove the soundness of the resource inference in Section 4. In Section 5, we present the parallel cost semantics, which models the work and span of executing a kernel in parallel on a GPU. We also show that it is approximated by the lock-step semantics and therefore by the resource analysis. We describe the bounded implementation that validates this model in Section 6. Next, we describe in more detail our implementation of the analysis (Section 7) and evaluate it (Section 8). We conclude with a discussion of related work (Section 10).

2 A BRIEF INTRODUCTION TO CUDA

In this section, we introduce some basic concepts of CUDA using a simple running example. We focus on the features of CUDA necessary to explain the performance bottlenecks targeted by our analysis. It should suffice to allow a reader unfamiliar with CUDA to follow the remainder of the article and is by no means intended as a thorough guide to CUDA.

Kernels and Threads. A kernel is invoked on the GPU by calling it much like a regular function with additional arguments specifying the number and layout of threads on which it should run. The number of threads running a kernel is often quite large and CUDA organizes them into a hierarchy. Threads are grouped into *blocks* and blocks form a *grid*.

Threads within a block and blocks within a grid may be organized in one, two, or three dimensions, which are specified when the kernel is invoked. A thread running CUDA code may access the x, y, and z coordinates of its thread index using the designated identifiers `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`. CUDA also defines the indentifiers `blockDim.(x|y|z)`, `blockIdx.(x|y|z)`, and `gridDim.(x|y|z)` for accessing the dimensions of a block, the index of the current thread's block, and the dimensions of the grid, respectively. Most of the examples in this article assume that blocks and the grid are one-dimensional (i.e., y and z dimensions are 1), unless otherwise specified.

SIMT Execution. GPUs are designed to execute the same arithmetic or logical instruction on many threads at once. This is referred to as **Single Instruction, Multiple Thread (SIMT)**

```
__global__ void addSub_k (int *A, int *B, int w, int h) { addS_k }
```

addS$_0$ ≡

```
for (int i = 0; i < w; i++) {
  int j = blockIdx.x * blockDim.x
          + threadIdx.x;
  if (j % 2 == 0) {
    B[j * w + i] += A[i];
  } else {
    B[j * w + i] -= A[i];
  }
}
```

addS$_1$ ≡

```
for (int i = 0; i < w; i++) {
  int j = blockIdx.x * blockDim.x
          + threadIdx.x;
  B[2 * j * w + i] += A[i];
  B[(2 * j + 1) * w + i] -= A[i];
}
}
```

addS$_2$ ≡

```
int i = blockIdx.x * blockDim.x
        + threadIdx.x;
for (int j = 0; j < h; j += 2) {
  B[j * w + i] += A[i];
  B[(j + 1) * w + i] -= A[i];
}
```

addS$_3$ ≡

```
__shared__ int As[blockDim.x];
int i = blockIdx.x * blockDim.x
        + threadIdx.x;
As[threadIdx.x] = A[i];
for (int j = 0; j < h; j += 2) {
  B[j * w + i] += As[i];
  B[(j + 1) * w + i] -= As[i];
}
```

Fig. 1. Four implementations addSub$_0$, . . .,addSub$_3$ of a CUDA kernel that alternately adds and subtracts from rows of a matrix.

execution. To reflect this, CUDA threads are organized into groups called *warps*.[1] The number of threads in a warp is defined by the identifier `warpSize` but is generally set to 32. All threads in a warp must execute the same instruction (although some threads may be inactive).

SIMT execution leads to a potential performance bottleneck in CUDA code. If a branching operation such as a conditional is executed and two threads within a warp take different execution paths, then the GPU must serialize the execution of that warp. It first deactivates the threads that took one execution path and executes the other, and then switches to executing the threads that took the second execution path. This is referred to as a *divergence* or *divergent warp* and can greatly reduce the parallelism of a CUDA kernel.

The functions addSub$_k$ in Figure 1 implement four versions of a kernel that adds the `w`-length array `A` pointwise to even rows of the `w` × `h` matrix `B` and subtracts it from odd rows. The annotation `__global__` is a CUDA extension indicating that addSub$_k$ is a kernel. To simplify some versions of the function, we assume that `h` is even. Otherwise, the code is similar to standard C code.

[1] *Warp* refers to the set of parallel threads stretched across a loom during weaving; according to the CUDA programming manual [NVIDIA Corporation 2019], "The term *warp* originates from weaving, the first parallel thread technology."

Consider first the function $\texttt{addSub}_0$ that is given by $\texttt{addS}_0$. The `for` loop iterates over the columns of the matrix, and each row is processed in parallel by separate threads. There is no need to iterate over the rows, because the main program instantiates the kernel for each row.

The implementation of $\texttt{addSub}_0$ contains a number of performance bugs. First, the conditional diverges at every iteration of the loop. The reason is that every warp contains thread identifiers (`threadIdx`) that result in both odd and even values for the variable `j`. A straightforward way to fix this bug is to remove the conditional, unroll the loop, and perform both the addition of an even row and the subtraction of an odd row in one loop iteration. The resulting code, shown in $\texttt{addSub}_1$, does not have more parallelism than the original—the addition and subtraction are still performed sequentially—but will perform better, because it greatly reduces the overhead of branching.

Memory Accesses. The next performance bottleneck we discuss relates to the way CUDA handles *global* memory accesses. CUDA warps can access up to 128 consecutive bytes of such memory at once. When threads in a warp access memory, such as the accesses to arrays `A` and `B` in the example, CUDA attempts to coalesce these accesses together into as few separate accesses as possible. If a warp accesses four consecutive 32-bit elements of an array, then the memory throughput of that instruction is four times higher than if it performs four non-consecutive reads.

The execution of the function $\texttt{addSub}_1$ is unable to coalesce accesses to `B`, because, assuming `w` is larger than 32 and the arrays are stored in row-major order, no two threads within a warp access memory within 128 bytes of each other. This is fixed by instead iterating over the rows of the matrix and handling the columns in parallel. This way, all of the memory accesses by a warp are consecutive (e.g., threads 0 through 31 might access `A[0]` through `A[31]` and `B[w]` through `B[w+63]`). The updated code is shown in the function body $\texttt{addSub}_2$.

Shared Memory. In all of the kernels discussed so far, the arrays `A` and `B` reside in *global* memory, which is stored on the GPU and visible to all threads. CUDA also provides a separate *shared* memory space, which is shared only by threads within a block. Shared memory has a lower latency than global memory, so we can, for example, use it to store the values of `A` rather than access global memory every time. In the function $\texttt{addsub}_3$, we declare a shared array `As` and copy values of `A` into `As` before their first use.

Some care must be taken to ensure that the code of $\texttt{addsub}_3$ is performant because of how shared memory is accessed. Shared memory consists of a number, generally 32, of separate banks. Separate banks may be accessed concurrently, but multiple concurrent accesses to separate addresses in the same bank are serialized. It is thus important to avoid "bank conflicts." Most GPUs ensure that 32 consecutive 32-bit memory reads will not result in any bank conflicts. However, if a block accesses a shared array at a stride other than 1, bank conflicts can accumulate.

3 MINICUDA CORE CALCULUS

In this section, we present a core calculus, called miniCUDA, that captures the features of CUDA that are of primary interest in this article: control flow (to allow for loops and to study the cost of divergent warps) and memory accesses. We will use this calculus to present the theory of our resource analysis for CUDA kernels.

In designing the miniCUDA calculus, we have made a number of simplifying assumptions that make the presentation cleaner and more straightforward. One notable simplification is that a miniCUDA program consists of a single kernel, without its function header. In addition, we collapse the structure of thread and block indices into a single thread ID, denoted tid. This causes no loss of generality as a three-dimensional thread index can be converted in a straightforward way to a one-dimensional thread ID, given the block dimension. The resource analysis will be parametric

Types	τ	::=	$\mathsf{int} \mid \mathsf{bool} \mid B \mid \mathsf{arr}(\tau)$
Operands	o	::=	$x \mid p \mid c \mid \mathsf{tid}$
Arrays	A	::=	$G \mid S$
Expressions	e	::=	$o \mid o \ \mathsf{op} \ o \mid A[o]$
Statements	s	::=	$\mathsf{skip} \mid s;\ s \mid x \leftarrow e \mid A[o] \leftarrow e \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ (e)\ s$

Fig. 2. Syntax of miniCUDA.

over the block index and other parameters, and will estimate the maximum resource usage of any warp in any block.

Syntax. The syntax of the miniCUDA calculus is presented in Figure 2. Two types of data are particularly important to the evaluation and cost analysis of the calculus: integers are used for both array indices (which determine the costs of memory accesses) and loop bounds (which are crucial for estimating the cost of loops), and Booleans are used in conditionals. All other base types (e.g., float, string) are represented by an abstract base type B. We also include arrays of any type.

The terms of the calculus are divided into statements, which may affect control flow or the state of memory, and expressions, which do not have effects. We further distinguish *operands*, which consist of thread-local variables x, parameters p to the kernel (these include arguments passed to the kernel function as well as CUDA parameters such as the block index and warp size, which are distinguished from variables, because we assume them to be read-only and potentially have different cost profiles for accesses), constants c (of types int, bool, and B), and a designated variable tid for accessing the thread ID. For simplicity, all variables are in scope throughout the kernel (but each thread maintains its own local copy of each variable). Additional expressions include $o_1 \ \mathsf{op} \ o_2$, which stands for an arbitrary binary operation, and array accesses $A[o]$. The metavariable A stands for a generic array. When relevant, we use metavariables that indicate whether the array is stored in (G)lobal or (S)hared memory. Note that subexpressions of expressions are limited to operands; more complex expressions must be broken down into binary ones by binding intermediate results to variables. In addition, results of array reads may not be used directly as indices into an array (for reading or, as we will see shortly, for writing). This restriction simplifies reasoning about expressions without limiting expressivity.

Statements include two types of assignment: assignment to a local variable and to an array element. Statements also include conditionals and while loops. The keyword skip represents the "empty" statement. Statements may be sequenced with semicolons, e.g., $s_1;\ s_2$.

We present a static semantics for miniCUDA to show the internal consistency of our definitions. In addition, type safety enforces some "sanity checks" that will be necessary for our later results, namely, that array indices are integers. The static semantics are defined over signatures Σ that record the types of local variables, parameters, operators, functions, and arrays. Formally, a signature is a mapping from the set $\mathit{Vars} \cup \mathit{Params} \cup \mathit{Operators} \cup \mathit{Functions} \cup \mathit{Arrays}$ to types. The typing judgment for expressions (and operands) is $\Sigma \vdash e : \tau$, which indicates that under signature Σ, expression e has type τ. In our calculus, statements do not have return values and so do not have types, but the judgment $\Sigma \vdash s$ is used to indicate that, under signature Σ, the statement s is well-formed in that all of its subexpressions have the expected type. The rules for these judgments are given in Figures 3 and 4. In Rule OS:Const, the function $\mathit{Type}(\cdot)$ gives the type of a constant c, which stands for a (floating-point, integer, string, etc.) literal. In general, the rules ensure that subexpressions are well-typed at appropriate types, e.g., array indices must have type int and conditional expressions must have type bool. Sub-statements of conditionals and loops must be well-formed.

$$\frac{\Sigma(x) = \tau}{\Sigma \vdash x : \tau}\ (\text{OS:Var}) \qquad \frac{Type(c) = \tau}{\Sigma \vdash c : \tau}\ (\text{OS:Const}) \qquad \frac{\Sigma(p) = \tau}{\Sigma \vdash p : \tau}\ (\text{OS:Param}) \qquad \frac{}{\Sigma \vdash \mathsf{tid} : \mathsf{int}}\ (\text{OS:Tid})$$

$$\frac{\Sigma(\mathsf{op}) = \tau_1 \times \tau_2 \to \tau_3 \qquad \Sigma \vdash o_1 : \tau_1 \qquad \Sigma \vdash o_2 : \tau_2}{\Sigma \vdash o_1\ \mathsf{op}\ o_2 : \tau_3}\ (\text{ES:Op}) \qquad \frac{\Sigma(A) : \mathsf{arr}(\tau) \qquad \Sigma \vdash o : \mathsf{int}}{\Sigma \vdash A[o] : \tau}\ (\text{ES:Array})$$

Fig. 3. Typing rules for operands and expressions.

$$\frac{}{\Sigma \vdash \mathsf{skip}}\ (\text{SS:Skip}) \qquad \frac{\Sigma, x : \tau \vdash e : \tau}{\Sigma, x : \tau \vdash x \leftarrow e}\ (\text{SS:VWrite}) \qquad \frac{\Sigma(A) = \mathsf{arr}(\tau) \qquad \Sigma \vdash o : \mathsf{int} \qquad \Sigma \vdash e : \tau}{\Sigma \vdash A[o] \leftarrow e}\ (\text{SS:ArrWrite})$$

$$\frac{\Sigma \vdash e : \mathsf{bool} \qquad \Sigma \vdash s_1 \qquad \Sigma \vdash s_2}{\Sigma \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2}\ (\text{SS:If}) \qquad \frac{\Sigma \vdash e : \mathsf{bool} \qquad \Sigma \vdash s}{\Sigma \vdash \mathsf{while}\ (e)\ s}\ (\text{SS:While})$$

$$\frac{\Sigma \vdash s_1 \qquad \Sigma \vdash s_2}{\Sigma \vdash s_1;\ s_2}\ (\text{SS:Seq})$$

Fig. 4. Typing rules for statements.

Table 1. Resource Constants and Sample Resource Metrics

Const.	Type	Add'l argument	sectors	conflicts	divwarps	steps
M^{var}	$\mathbb{Q}$		0	0	0	1
M^{const}	$\mathbb{Q}$		0	0	0	1
M^{param}	$\mathbb{Q}$		0	0	0	1
M^{op}	$\mathbb{Q}$		0	0	0	1
M^{gread}	$\mathbb{N} \to \mathbb{Q}$	# of seq. reads	$\lambda n.\ F(n)$	$\lambda\_.\ 0$	$\lambda\_.\ 0$	$\lambda n.\ F(n)$
M^{sread}	$\mathbb{N} \to \mathbb{Q}$	# of conflicts	$\lambda\_.\ 0$	$\lambda n.\ F(n) - 1$	$\lambda\_.\ 0$	$\lambda n.\ F(n) - 1$
M^{if}	$\mathbb{Q}$		0	0	0	1
M^{div}	$\mathbb{Q}$		0	0	1	1
M^{vwrite}	$\mathbb{Q}$		0	0	0	1
M^{gwrite}	$\mathbb{N} \to \mathbb{Q}$	# of seq. reads	$\lambda n.\ F(n)$	$\lambda\_.\ 0$	$\lambda\_.\ 0$	$\lambda n.\ F(n)$
M^{swrite}	$\mathbb{N} \to \mathbb{Q}$	# of conflicts	$\lambda\_.\ 0$	$\lambda n.\ F(n) - 1$	$\lambda\_.\ 0$	$\lambda n.\ F(n) - 1$

We use $F(n)$ to convert from integers to rationals.

Costs and Resource Metrics. In the following, we present an operational cost semantics and then a quantitative program logic for miniCUDA kernels. Both the operational semantics and the logic are parametric over a *resource metric*, which specifies the exact resource being considered. A resource metric M is a function whose domain is a set of resource *constants* that specify particular operations performed by a CUDA program. The resource metric maps these constants to rational numbers, possibly taking an additional argument depending on the constant supplied. A resource metric applied to a constant rc is written M^{rc}, and its application to an additional argument n, if required, is written $M^{\mathsf{rc}}(n)$. The only resource constant that does not correspond to a syntactic operation is M^{div}, which is the cost overhead of a divergent warp. The resource constants for miniCUDA, their types and the meanings of their additional argument (if any) are defined in Table 1.

The cost of accessing an array depends upon a parameter specifying the number of separate accesses required (for global memory) or the maximum number of threads attempting to access a single shared memory bank (for shared memory). These values are supplied by two additional parameters to the operational semantics and the resource analysis. Given a set of array indices R,

the function MemReads(R) returns the number of separate reads (or writes) required to access all of the indices, and the function Conflicts(R) returns the maximum number of indices that map to the same shared memory bank. These parameters are separated from the resource metric, because they do not depend on the resource, but on the details of the hardware (e.g., the size of reads and the number of shared memory banks). We discuss these functions more concretely in the next subsection. Resource metrics applied to the appropriate constants simply take the output of these functions and return the cost (in whatever resource) of performing that many memory accesses. We require only that this cost be monotonic, i.e., that if $i \leq j$, then $M^{\text{gread}}(i) \leq M^{\text{gread}}(j)$, and similarly for M^{sread}, M^{gwrite}, and M^{swrite}.

Table 1 also lists the concrete cost values for four resource metrics we consider in our evaluation:

— conflicts: Counts the cost of bank conflicts.
— sectors: Counts the total number of reads and writes to memory, including multiple requests needed to serve uncoalesced requests.[2]
— divwarps: Counts the number of times a warp diverges.
— steps: Counts the total number of evaluation steps.

Lock-step Operational Semantics. We now define an operational semantics for evaluating mini-CUDA kernels that also tracks the cost of evaluation given a resource metric. We use this semantic model as the basis for proving the soundness of our resource analysis in the next section. The operational semantics evaluates an expression or statement over an entire warp at a time to produce a result and the cost of evaluation. Because it only evaluates one warp and threads of a warp execute in lock-step, we refer to this as the "lock-step" semantics to differentiate it from the "parallel" semantics we introduce in Section 5. Recall, however, that warps can *diverge* at conditionals, resulting in only some subset of the threads of a warp being active in each branch. We therefore track the set $\mathcal{T}$ of currently active threads in the warp as a parameter to the judgments. Finally, the operational semantics also requires a store σ, representing the values stored in memory. Expressions (and operands) differ from statements in that expressions do not alter memory but simply evaluate to a value. Because every thread might compute on different values, the result is not a single value but rather a *family* of values, one per active thread, which we represent as a family R indexed by $\mathcal{T}$. We write the result computed by thread t as $R(t)$. Following standard conventions, we build such result families by writing $f_{t\in\mathcal{T}}$, where f is a mathematical expression (not necessarily mini-CUDA syntax) that may refer to t, assumed to be drawn from the domain $\mathcal{T}$. As an example, the result of evaluating the operand tid is written $R_{\text{tid}} = (\mathit{Tid}(t))_{t\in\mathcal{T}}$ indicating that for all $t \in \mathcal{T}$, we have $R_{\text{tid}}(t) = \mathit{Tid}(t)$. The result family may also not reference t, in which case the result is constant, e.g., evaluating a constant c evaluates to the result family $(c)_{t\in\mathcal{T}}$. In contrast, statements do not return values but simply change the state of memory; statement evaluation therefore simply produces a new store. These distinctions are evident in the two semantic judgments:

$$\sigma; e \downarrow_M^{\mathcal{T}} R; C$$

indicates that, under store σ, the expression e evaluates on threads $\mathcal{T}$ to R with cost C. Statements are evaluated with the judgment

$$\sigma; s \Downarrow_M^{\mathcal{T}} \sigma'; C.$$

Evaluation rules for both judgments are presented in Figure 5. We now discuss additional representation details and discuss some rules in more detail. As suggested above, the elements of $\mathcal{T}$ are abstract thread identifiers t. The function $\mathit{Tid}(t)$ converts such an identifier to an integer thread ID. The domain of a store σ is $(\mathit{Arrays} \times \mathbb{Z}) \cup (\mathit{LocalVars} \times \mathit{Threads})$. For an array A (regardless of

[2]The term "sectors" comes from the NVIDIA profiling tools' terminology for this metric.

$$\text{(OC:Var)}\quad \frac{}{\sigma; x \downarrow^{\mathcal{T}}_M (\sigma(x,t))_{t\in\mathcal{T}}; M^{\mathbf{var}}} \qquad \text{(OC:Const)}\quad \frac{}{\sigma; c \downarrow^{\mathcal{T}}_M (c)_{t\in\mathcal{T}}; M^{\mathbf{const}}} \qquad \text{(OC:Param)}\quad \frac{}{\sigma; p \downarrow^{\mathcal{T}}_M (p)_{t\in\mathcal{T}}; M^{\mathbf{param}}}$$

$$\text{(EC:Op)}\quad \frac{\sigma; o_1 \downarrow^{\mathcal{T}}_M R_1; C_1 \qquad \sigma; o_2 \downarrow^{\mathcal{T}}_M R_2; C_2}{\sigma; o_1 \mathbin{\mathsf{op}} o_2 \downarrow^{\mathcal{T}}_M (R_1(t) \mathbin{\mathsf{op}} R_2(t))_{t\in\mathcal{T}}; C_1 + C_2 + M^{\mathbf{op}}} \qquad \text{(EC:GArr)}\quad \frac{\sigma; o \downarrow^{\mathcal{T}}_M R; C}{\sigma; G[o] \downarrow^{\mathcal{T}}_M (\sigma(G, R(t)))_{t\in\mathcal{T}}; C + M^{\mathbf{gread}}(\mathrm{MemReads}(R))}$$

$$\text{(OC:Tid)}\quad \frac{}{\sigma; \mathsf{tid} \downarrow^{\mathcal{T}}_M (Tid(t))_{t\in\mathcal{T}}; M^{\mathbf{var}}} \qquad \text{(EC:SArr)}\quad \frac{\sigma; o \downarrow^{\mathcal{T}}_M R; C}{\sigma; S[o] \downarrow^{\mathcal{T}}_M (\sigma(S, R(t)))_{t\in\mathcal{T}}; C + M^{\mathbf{sread}}(\mathrm{Conflicts}(R))}$$

$$\text{(SC:GWrite)}\quad \frac{\sigma; o \downarrow^{\mathcal{T}}_M R_1; C_1 \qquad \sigma; e \downarrow^{\mathcal{T}}_M R_2; C_2}{\sigma; G[o] \leftarrow e \Downarrow^{\mathcal{T}}_M \sigma[(G, R_1(t)) \mapsto R_2(t) \mid t \in \mathcal{T}]; C_1 + C_2 + M^{\mathbf{gwrite}}(\mathrm{MemReads}(R_1))}$$

$$\text{(SC:SWrite)}\quad \frac{\sigma; o \downarrow^{\mathcal{T}}_M R_1; C_1 \qquad \sigma; e \downarrow^{\mathcal{T}}_M R_2; C_2}{\sigma; S[o] \leftarrow e \Downarrow^{\mathcal{T}}_M \sigma[(S, R_1(t)) \mapsto R_2(t) \mid t \in \mathcal{T}]; C_1 + C_2 + M^{\mathbf{swrite}}(\mathrm{Conflicts}(R_1))} \qquad \text{(SC:Skip)}\quad \frac{}{\sigma; \mathsf{skip} \Downarrow^{\mathcal{T}}_M \sigma; 0}$$

$$\text{(SC:Seq)}\quad \frac{\sigma; s_1 \Downarrow^{\mathcal{T}}_M \sigma_1; C_1 \qquad \sigma_1; s_2 \Downarrow^{\mathcal{T}}_M \sigma_2; C_2}{\sigma; s_1;\ s_2 \Downarrow^{\mathcal{T}}_M \sigma_2; C_1 + C_2} \qquad \text{(SC:VWrite)}\quad \frac{\sigma; e \downarrow^{\mathcal{T}}_M R; C}{\sigma; x \leftarrow e \Downarrow^{\mathcal{T}}_M \sigma[(x,t) \mapsto R(t) \mid t \in \mathcal{T}]; C + M^{\mathbf{vwrite}}}$$

$$\text{(SC:IfT)}\quad \frac{\sigma; e \downarrow^{\mathcal{T}}_M (True)_{t\in\mathcal{T}}; C_1 \qquad \sigma; s_1 \Downarrow^{\mathcal{T}}_M \sigma_1; C_2}{\sigma; \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \Downarrow^{\mathcal{T}}_M \sigma_1; C_1 + M^{\mathbf{if}} + C_2}$$

$$\text{(SC:IfD)}\quad \frac{\mathcal{T}_T = \{t \in \mathcal{T} \mid R(t)\} \neq \emptyset \qquad \mathcal{T}_F = \{t \in \mathcal{T} \mid \neg R(t)\} \neq \emptyset \qquad \sigma; e \downarrow^{\mathcal{T}}_M R; C_1 \qquad \sigma; s_1 \Downarrow^{\mathcal{T}_T}_M \sigma_1; C_2 \qquad \sigma_1; s_2 \Downarrow^{\mathcal{T}_F}_M \sigma_2; C_3}{\sigma; \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \Downarrow^{\mathcal{T}}_M \sigma_2; C_1 + M^{\mathbf{if}} + C_2 + C_3 + M^{\mathbf{div}}}$$

$$\text{(SC:IfF)}\quad \frac{\sigma; e \downarrow^{\mathcal{T}}_M (False)_{t\in\mathcal{T}}; C_1 \qquad \sigma; s_2 \Downarrow^{\mathcal{T}}_M \sigma_1; C_2}{\sigma; \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \Downarrow^{\mathcal{T}}_M \sigma_1; C_1 + M^{\mathbf{if}} + C_2}$$

$$\text{(SC:WhileAll)}\quad \frac{\sigma; e \downarrow^{\mathcal{T}}_M (True)_{t\in\mathcal{T}}; C_1 \qquad \sigma; s \Downarrow^{\mathcal{T}}_M \sigma_1; C_2 \qquad \sigma_1; \mathsf{while}\ (e)\ s \Downarrow^{\mathcal{T}}_M \sigma_2; C_3}{\sigma; \mathsf{while}\ (e)\ s \Downarrow^{\mathcal{T}}_M \sigma_2; C_1 + C_2 + M^{\mathbf{if}} + C_3}$$

$$\text{(SC:WhileSome)}\quad \frac{\sigma; e \downarrow^{\mathcal{T}}_M R; C_1 \qquad \emptyset \neq \mathcal{T}_T \neq \mathcal{T} \qquad \sigma; s \Downarrow^{\mathcal{T}_T}_M \sigma_1; C_2 \qquad \mathcal{T}_T = \{t \in \mathcal{T} \mid R(t)\} \qquad \sigma_1; \mathsf{while}\ (e)\ s \Downarrow^{\mathcal{T}_T}_M \sigma_2; C_3}{\sigma; \mathsf{while}\ (e)\ s \Downarrow^{\mathcal{T}}_M \sigma_2; C_1 + C_2 + M^{\mathbf{if}} + M^{\mathbf{div}} + C_3}$$

$$\text{(SC:WhileNone)}\quad \frac{\sigma; e \downarrow^{\mathcal{T}}_M (False)_{t\in\mathcal{T}}; C}{\sigma; \mathsf{while}\ (e)\ s \Downarrow^{\mathcal{T}}_M \sigma; C + M^{\mathbf{if}}}$$

Fig. 5. Evaluation rules.

whether it stored in global or shared memory), $\sigma(A, n)$ returns the nth element of A. Note that, for simplicity of presentation, we assume that out-of-bounds indices (including negative indices) map to some default value. For a local variable x, $\sigma(x, t)$ returns the value of x for thread t.

We write $\Sigma \vdash_{\mathcal{T}} \sigma$ to mean that a store is well-typed with respect to a signature and a set of threads, if:

(1) For all $A : \tau \in \Sigma$ and all $n \in \mathbb{N}$, we have $\Sigma \vdash \sigma(A, n) : \tau$,
(2) For all $x : \tau \in \Sigma$ and all $t \in \mathcal{T}$, we have $\Sigma \vdash \sigma(x, t) : \tau$.

We remark that store typing is preserved by taking subsets of the thread set: if $\Sigma \vdash_{\mathcal{T}} \sigma$ and $\mathcal{T}' \subset \mathcal{T}$, then $\Sigma \vdash_{\mathcal{T}'} \sigma$.

The evaluation of expressions is relatively straightforward: We simply evaluate subexpressions in parallel and combine appropriately, but with careful accounting of the costs. The cost of execution is generally obtained by summing the costs of evaluating subexpressions with the cost of the head operation given by the resource metric M. For example, Rule EC:Op evaluates the two

operands and combines the results using the concrete binary operation represented by op. The cost is the cost of the two subexpressions, C_1 and C_2, plus M^{op}. Array accesses evaluate the operand to a set of indices and read the value from memory at each index. The cost of these operations depends on MemReads(R) or Conflicts(R), where R is the set of indices. Recall that these functions give the number of global memory reads necessary to access the memory locations specified by R, and the number of bank conflicts resulting from simultaneously accessing the memory locations specified by R, respectively. As discussed before, we leave these functions as parameters, because their exact definitions can change across versions of CUDA and hardware implementations. As examples of these functions, we give definitions consistent with common specifications in modern CUDA implementations [NVIDIA Corporation 2019]:

$$\begin{aligned} \text{MemReads}(R) &\triangleq \left|\left\{\left\lceil \tfrac{i}{32} \right\rceil \mid (i)_t \in R\right\}\right|, \\ \text{Conflicts}(R) &\triangleq \max_{j \in [0,31]} \{R(t) \equiv j \bmod 32 \mid t \in \mathit{Dom}(R)\}. \end{aligned}$$

Above, we assume that global reads are 128 bytes in size and array elements are 4 bytes. In reality, and in our implementation, MemReads(R) depends on the type of the array.

Statement evaluation is slightly more complex, as statements can update the state of memory and also impact control flow: the former is represented by updating the store σ and the latter is represented by changing the thread set $\mathcal{T}$ when evaluating subexpressions and substatements. For assignment statements, the new state comes from updating the state with the new assignment. We write $\sigma[(x,t) \mapsto R(t) \mid t \in \mathcal{T}]$ to indicate the state σ updated so that for all $t \in \mathcal{T}$, the binding (x,t), which is the element of the domain of σ corresponding to the value of local variable x on thread t now maps to $R(t)$, that is, to the value of result family R corresponding to thread t. Array updates are written similarly. Note, however, that an array update $\sigma[(A,R(t)) \mapsto R_v(t) \mid t \in \mathcal{T}]$, where $R(t)$ is the array index being written to by thread t, is undefined if $R(t) = R(t')$ for threads $t \neq t'$, i.e., if there is a data race. In CUDA, a data race results in undefined behavior, so we do not explicitly model programs with data races. Indeed, in the small-step semantics presented in Section 6, a write-write data race is a "stuck state" that cannot make progress, so we explicitly quantify over programs with no such races in our results.

Conditionals and while loops each have three rules. If a conditional evaluates to True for all threads (SC:IfT), then we simply evaluate the "if" branch with the full set of threads $\mathcal{T}$, and similar if all threads evaluate to False. If, however, there are non-empty sets of threads where the conditional evaluates to True and False ($\mathcal{T}_T$ and $\mathcal{T}_F$, respectively), then we must evaluate both branches. We evaluate the "if" branch with $\mathcal{T}_T$ and the "else" branch with $\mathcal{T}_F$. Note that the resulting state of the "if" branch is passed to evaluation of the "else" branch; this corresponds to CUDA executing the two branches in sequence. This rule, SC:IfD, also adds the cost M^{div} of a divergent warp. The three rules for while loops similarly handle the cases in which all, some or none of the threads in $\mathcal{T}$ evaluate the condition to be True. The first two rules both evaluate the body under the set of threads for which the condition is true and then reevaluate the loop. Rule SC:WhileSome also indicates that we must pay M^{div}, because the warp diverges.

Example 1. As an example, Figure 6 gives the cost semantics derivation for the evaluation of the statement

$$\text{if tid} < 16 \text{ then } G[\text{tid} * 4] \leftarrow 0 \text{ else } G[\text{tid}] \leftarrow 0.$$

For clarity, the three subderivations of the root are split out into *DCond*, which computes the condition, *DThen*, which computes the then branch, and *DElse*, which computes the else branch. These three derivations are shown separately in the figure. We assume that the initial thread set $\mathcal{T}$ consists of a full warp of threads with indices 0–31. First, the condition tid $<$ 16 is evaluated. This derivation has EC:Op as its root and computes both tid and 16, both of which have trivial

$$\dfrac{\dfrac{}{\sigma; \mathsf{tid} \downarrow_M^{\mathcal{T}} [x \mid x \in [0,31]]; M^{\mathsf{var}}}\,(\textsc{Oc:Tid}) \qquad \dfrac{}{\sigma; 16 \downarrow_M^{\mathcal{T}} 16_{[0,31]}; M^{\mathsf{const}}}\,(\textsc{Oc:Const})}{\sigma; \mathsf{tid} < 16 \downarrow_M^{\mathcal{T}} [x < 16 \mid x \in [0,31]]; M^{\mathsf{var}} + M^{\mathsf{const}} + M^{\mathsf{op}}}\,(\textsc{Ec:Op}) \quad DCond$$

$$\dfrac{\dfrac{\dfrac{}{\sigma; \mathsf{tid} \downarrow_M^{\mathcal{T}_{<16}} [x \mid x \in [0,15]]; M^{\mathsf{var}}}\,(\textsc{Oc:Tid}) \quad \dfrac{}{\sigma; 4 \downarrow_M^{\mathcal{T}_{<16}} 4_{[0,15]}; M^{\mathsf{const}}}\,(\textsc{Oc:Const})}{\sigma; \mathsf{tid} * 4 \downarrow_M^{\mathcal{T}_{<16}} [4x \mid x \in [0,15]]; M^{\mathsf{var}} + M^{\mathsf{const}} + M^{\mathsf{op}}} \quad \dfrac{}{\sigma; 0 \downarrow_M^{\mathcal{T}_{<16}} 0_{[0,15]}; M^{\mathsf{const}}}\,(\textsc{Oc:Const})}{\sigma; G[\mathsf{tid} * 4] \leftarrow 0 \Downarrow_M^{\mathcal{T}_{<16}} \sigma'; 2M^{\mathsf{const}} + M^{\mathsf{var}} + M^{\mathsf{op}} + M^{\mathsf{gwrite}}(2)}\,(\textsc{Sc:GWrite}) \quad DThen$$

$$\dfrac{\dfrac{}{\sigma; \mathsf{tid} \downarrow_M^{\mathcal{T}_{\geq 16}} [x \mid x \in [16,31]]; M^{\mathsf{var}}}\,(\textsc{Oc:Tid}) \quad \dfrac{}{\sigma; 0 \downarrow_M^{\mathcal{T}_{\geq 16}} 0_{[0,15]}; M^{\mathsf{const}}}\,(\textsc{Oc:Const})}{\sigma'; G[\mathsf{tid}] \leftarrow 0 \Downarrow_M^{\mathcal{T}_{\geq 16}} \sigma''; M^{\mathsf{var}} + M^{\mathsf{const}} + M^{\mathsf{gwrite}}(1)}\,(\textsc{Sc:GWrite}) \quad DElse$$

$$\dfrac{DCond \qquad DThen \qquad DElse}{\sigma; \mathsf{if}\ \mathsf{tid} < 16\ \mathsf{then}\ G[\mathsf{tid} * 4] \leftarrow 0\ \mathsf{else}\ G[\mathsf{tid}] \leftarrow 0 \Downarrow_M^{\mathcal{T}} \sigma''; C}\,(\textsc{Sc:IfD})$$

Fig. 6. Example derivation in the cost semantics.

derivations. The total cost is $M^{\mathsf{var}} + M^{\mathsf{const}} + M^{\mathsf{op}}$. This yields the result $[x < 16 \mid x \in [0,31]]$, i.e., an indexed family consisting of True for threads 0–15 and False for threads 16–31. The thread set $\mathcal{T}$ is thus divided into threads 0–15, which we will call $\mathcal{T}_{<16}$, and threads 16–31, which we will call $\mathcal{T}_{\geq 16}$. Because both sets are non-empty, we must apply rule SC:IfD. The first set evaluates the then branch, computing the array indices tid $*$ 4 and the assigned expression 0 for each thread in the set. After performing the assignment, the store becomes

$$\sigma' = \sigma[(G, 4x) \mapsto 0 \mid x \in [0,15]],$$

that is, G is updated with multiple-of-four indices from 0 to 60 mapped to 0. The cost is $2M^{\mathsf{const}} + M^{\mathsf{var}} + M^{\mathsf{op}} + M^{\mathsf{gwrite}}(2)$, adding up the costs from computing the two expressions and the cost of two sector writes (two because indices 0–31 are one sector and indices 32–63 are another). Next, the remaining threads, $\mathcal{T}_{\geq 16}$, execute the else branch. Because this is performed sequentially after the then branch, the store starts off as σ', which we obtained previously, and becomes

$$\sigma' = \sigma'[(G, x) \mapsto 0 \mid x \in [16,31]].$$

This write only accesses one sector and performs no arithmetic operations, so the cost is $M^{\mathsf{var}} + M^{\mathsf{const}} + M^{\mathsf{gwrite}}(1)$. Finally, the costs of the condition, the two branches and $M^{\mathsf{if}} + M^{\mathsf{div}}$ are added together to get the full cost of the conditional statement:

$$C = 4M^{\mathsf{const}} + 3M^{\mathsf{var}} + 2M^{\mathsf{op}} + M^{\mathsf{gwrite}}(2) + M^{\mathsf{if}} + M^{\mathsf{gwrite}}(1) + M^{\mathsf{div}}.$$

Example 2. In Figure 7, we show the derivation for the evaluation of while (tid $\geq i*16$) $i \leftarrow i+1$ starting, as before, with the thread set $\mathcal{T}$ being a full warp and a store σ such that $\sigma(i,t) = 0$ for all $t \in \mathcal{T}$. As before, the derivation is split up for clarity. The root is derivation D, which shows the first iteration of the loop. The first subderivation checks the condition, which produces a result R_1 in which all threads map to True, and has cost C_e. Because all threads map to True, this step in the derivation uses rule SC:WhileAll. To focus on the important points of a derivation for a while loop, we omit the remainder of the subderivation and leave the result and cost abstract. Similarly,

$$D_1 \quad \dfrac{\dfrac{\cdots}{\sigma''; \mathsf{tid} \geq i * 16 \downarrow_M^{\mathcal{T}} R_3; C_e}}{\sigma''; \mathsf{while}\ (\mathsf{tid} \geq i * 16)\ i \leftarrow i + 1 \Downarrow_M^{\mathcal{T}_{\geq 16}} \sigma''; C_e + M^{\mathsf{if}}}\ \text{(SC:WhileNone)}$$

$$D_2 \quad \dfrac{\dfrac{\cdots}{\sigma'; \mathsf{tid} \geq i * 16 \downarrow_M^{\mathcal{T}} R_2; C_e} \qquad \dfrac{\cdots}{\sigma'; i \leftarrow i + 1 \Downarrow_M^{\mathcal{T}_{\geq 16}} \sigma''; C_s} \qquad D_1}{\sigma; \mathsf{while}\ (\mathsf{tid} \geq i * 16)\ i \leftarrow i + 1 \Downarrow_M^{\mathcal{T}} \sigma'; 2C_e + C_s + 2M^{\mathsf{if}} + M^{\mathsf{div}}}\ \text{(SC:WhileSome)}$$

$$D \quad \dfrac{\dfrac{\cdots}{\sigma; \mathsf{tid} \geq i * 16 \downarrow_M^{\mathcal{T}} R_1; C_e} \qquad \dfrac{\cdots}{\sigma; i \leftarrow i + 1 \Downarrow_M^{\mathcal{T}} \sigma'; C_s} \qquad D_2}{\sigma; \mathsf{while}\ (\mathsf{tid} \geq i * 16)\ i \leftarrow i + 1 \Downarrow_M^{\mathcal{T}} \sigma; 3C_e + 2C_s + 3M^{\mathsf{if}} + M^{\mathsf{div}}}\ \text{(SC:WhileAll)}$$

Fig. 7. Example derivation with While.

the next subderivation evaluates $i \leftarrow i + 1$, which updates the store to

$$\sigma' = \sigma[(i, t) \mapsto 1 \mid t \in \mathcal{T}]$$

and has cost C_s. Finally, D_2 continues with the next iteration of the loop.

The derivation D_2 evaluates the condition as before, which this time results in R_2, in which threads 0–15 map to False and threads 16–31 map to True. This leads to a divergence and requires us to use rule SC:WhileSome. As a result, the loop body is only evaluated on thread $\mathcal{T}_{\geq 16}$, which, as before, consists of threads 16–31. The result is σ'', which performs a similar update as above. Note, however, the update is only performed on threads in $\mathcal{T}_{\geq 16}$, so threads 0–15 will continue to see $i = 1$. In addition, this rule will add a cost of M^{div} at the end.

Finally, D_1 evaluates the condition (now on threads $\mathcal{T}_{\geq 16}$ and in store σ'', in which all active threads have $i = 2$), which results in R_2, which maps all active threads to False. This forces us to apply rule SC:WhileNone, which returns the state σ'' and the cost $C_e + M^{\text{if}}$, corresponding to the cost of evaluating the condition and branching. The final cost reflects three evaluations of the condition, two evaluations of the body, three branches and one divergent branch.

Coq Mechanization. The syntax of miniCUDA operands, expressions, and statements are mechanized in Coq as the `Inductives` `opd`, `exp`, and `stmt`, respectively, and the inductive definitions closely match those of Figure 2. There is also an inductive definition `type`, which contains the types `TNat`, `TBool`, and `TAbs` (representing the abstract base type B). We do not need an explicit array type in the mechanization, because array and scalar variables are treated separately; the type of an array variable is the type of its elements. Values are represented by a corresponding inductive `value` containing natural numbers, Booleans, and abstract values of other base types. The type alias `result := list (nat * value)` represents our result families R as an association list mapping thread IDs to values. Inductive definitions `typed_X` implement the typing judgments for `values`, `results`, `opds`, `exps`, and `stmts`.

We define resource metrics as a record type `resmetric`, which has a field for each of the resource constants in Table 1, with the associated types. We add axioms to formalize the monotonicity assumptions for the $M^{\text{gread}}(\cdot)$, $M^{\text{sread}}(\cdot)$, $M^{\text{gwrite}}(\cdot)$, and $M^{\text{swrite}}(\cdot)$ metrics, as well as the restriction that all costs given in a resource metric are nonnegative. This accounts for the majority of axioms/assumptions in the Coq formalization.

Finally, the inductive definitions `eval_X` implement the cost semantics of Figure 5 for `opds`, `exps`, and `stmts`.

Preservation. Lemma 1 is a preservation result for evaluation: if an operand (respectively, expression) is well-typed with a signature Σ and σ meets that signature for a set of threads $\mathcal{T}$, then evaluating the operand (respectively, expression) with that set of threads produces a result set in which each result has the type of the operand (respectively, expression). In addition, if a statement is well-formed with a signature Σ and σ meets that signature for a set of threads $\mathcal{T}$, then evaluating the statement produces a well-formed store.

Lemma 1.

(1) *If* $\Sigma \vdash o : \tau$ *and* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and* $\sigma; o \downarrow_M^{\mathcal{T}} R; C$, *then for all* $t \in \mathcal{T}$, *we have* $\Sigma \vdash R(t) : \tau$.
(2) *If* $\Sigma \vdash e : \tau$ *and* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and* $\sigma; e \downarrow_M^{\mathcal{T}} R; C$, *then for all* $t \in \mathcal{T}$, *we have* $\Sigma \vdash R(t) : \tau$.
(3) *If* $\Sigma \vdash s$ *and* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and* $\sigma; s \Downarrow_M^{\mathcal{T}} \sigma'; C$, *then* $\Sigma \vdash_{\mathcal{T}} \sigma'$.

Proof. The three parts of the lemma are formalized and proven in the Coq mechanization. The only cases that do not immediately follow from induction or previous lemmas are those involving array reads and assignments. We sketch below the proof ideas used in those cases.

(1) By induction on the derivation of $\sigma; o \downarrow_M^{\mathcal{T}} R; C$.
   — OC:Var. By inversion on OS:Var, $x : \tau \in \Sigma$. By inversion on $\Sigma \vdash_{\mathcal{T}} \sigma$, for all $t \in \mathcal{T}$, we have $\Sigma \vdash \sigma(x, t) : \tau$.
(2) By induction on the derivation of $\sigma; e \downarrow_M^{\mathcal{T}} R; C$.
   — EC:Op. Then $e = o_1$ op o_2. By inversion on ES:Op, we have $\Sigma(\oplus) = \tau_1 \times \tau_2 \to \tau_3$ and $\Sigma \vdash o_1 : \tau_1$ and $\Sigma \vdash o_2 : \tau_2$. Let $t \in \mathcal{T}$. By induction, $\Sigma \vdash R_1(t) : \tau_1$ and $\Sigma \vdash R_2(t) : \tau_2$. By definition, $R(t) = R_1(t)$ op $R_2(t)$ and $\Sigma \vdash R(t) : \tau_3$.
   — EC:GArr, EC:HArr, EC:SArr. Then $e = A[o]$. By inversion on ES:Array, we have $A : \tau \in \Sigma$ and $\Sigma \vdash o :$ int. Let $t \in \mathcal{T}$. By induction, $\Sigma \vdash R(t) :$ int. By inversion on $\Sigma \vdash_{\mathcal{T}} \sigma$, we have $\Sigma \vdash \sigma(A)(R(t)) : \tau$.
(3) By induction on the derivation of $\sigma; s \Downarrow_M^{\mathcal{T}} \sigma'; C$.
   — SC:VWrite. Then $\sigma' = \sigma[(x, t) \mapsto R(t) \mid t \in \mathcal{T}]$, where $\sigma; e \downarrow_M^{\mathcal{T}} R; C_e$. By inversion on SS:VWrite, we have $x : \tau \in \Sigma$ and $\Sigma \vdash e : \tau$. By part (1), we have $\Sigma \vdash R(t) : \tau$ for all $t \in \mathcal{T}$, so $\Sigma \vdash_{\mathcal{T}} \sigma'$. SC:GWrite and SC:SWrite are similar.
   — SC:IfF, SC:IfF, SC:IfD, SC:WhileAll, SC:WhileSome, SC:WhileNone, and SC:Seq follow directly from induction. □

4 QUANTITATIVE PROGRAM LOGIC

In this section, we present declarative rules for a Hoare-style logic that can be used to reason about the resource usage of a warp of a miniCUDA kernel. The analysis for resource usage is based on the ideas of AARA [Hoffmann et al. 2011; Hofmann and Jost 2003]. The key idea of this analysis is to assign a non-negative numerical potential to states of computation. This potential must be sufficient to cover the cost of the following step and the potential of the next state. For imperative programs, the potential is generally a function of the values of local variables. Rules of a *quantitative Hoare logic* specify how this *potential function* changes during the execution of a statement. A derivation in the quantitative Hoare logic then builds a set of constraints on potential functions at every program point. In an implementation (Section 7), these constraints are converted to a linear program and solved with an LP solver.

As an example, consider the statement `for (int i = N; i >=0; i--) { f(); }`, and suppose we wish to bound the number of calls to `f`. This corresponds to a resource metric in which the cost of a function call is 1 and all other operations are free. The potential function at each point should be a function of the value of `i`: it will turn out to be the case that the correct solution is to set the potential to `i+1` in the body of the loop before the call to `f` and to `i` after the call. This difference

"pays for" for the cost of 1 for the function call. It also sets up the proper potential for the next loop iteration: when `i` is decremented following the loop, the potential once again becomes `i+1`.

The particular challenge of designing such a logic for CUDA is that each thread in a warp has a distinct local state. To keep inference tractable and scalable, we wish to reason about only one copy of each variable, but must then be careful about what exactly is meant by any function of a state, and in particular the resource functions: such a function on the values of local variables is not well-defined for CUDA local variables, which have a value for each thread. To solve this problem, we make an observation about CUDA programs: There is often a separation between local program variables that carry *data* (e.g., are used to store data loaded from memory or intermediate results of computation) and those that carry *potential* (e.g., are used as indices in `for` loops). To develop a sound and useful quantitative logic, it suffices to track potential for the latter set of variables, which generally hold the same value across all active threads.

Pre- and Post-Conditions. Conditions of our logic have the form $\{P; Q; X\}$ and consist of the *logical condition* P and the *potential function* Q (both of which we describe below) as well as a set X of variables whose values are uniform across the warp and therefore can be used as potential-carrying variables as described above. We write $\sigma, \mathcal{T} \vdash X$ to mean that for all $x \in X$ and all $t_1, t_2 \in \mathcal{T}$, we have $\sigma(x, t_1) = \sigma(x, t_2)$. The *logical condition* P is a reasonably standard Hoare logic pre- or post-condition and contains logical propositions that express properties over the state of the store. An example of a logical condition might be $i = 2 \cdot \mathsf{tid}$ for some integer constant k, indicating that program variable i contains twice the value of the thread ID for each thread (if i is used as an array access, such a fact will allow our analysis to compute the number of sectors accessed and number of bank conflicts). We write $\sigma, \mathcal{T} \vDash P$ to indicate that the condition P is true under the store σ and values $t \in \mathcal{T}$ for the thread identifier. If either the store or the set of threads is not relevant in a particular context, then we may use the shorthand $\sigma \vDash P$ to mean that there exists some $\mathcal{T}$ such that $\sigma, \mathcal{T} \vDash P$ or the shorthand $\mathcal{T} \vDash P$ to mean that there exists some σ such that $\sigma, \mathcal{T} \vDash P$.[3] We write $P \Rightarrow P'$ to mean that P *implies* P': that is, for all $\sigma, \mathcal{T}$ such that $\sigma, \mathcal{T} \vDash P$, it is the case that $\sigma, \mathcal{T} \vDash P'$. Note that the store (which does not contain the distinguished variable tid) and the set of threads are independent, and so if $\mathcal{T} \vDash P$ and $\sigma \vDash P$, then $\sigma, \mathcal{T} \vDash P$. We will sometimes write $\sigma \restriction_{\mathcal{T}}, \mathcal{T} \vDash P$ to indicate that P holds on just the part of σ that refers to the local state of threads in $\mathcal{T}$. We again assume that P treats these components of memory independently, i.e., that if $\mathcal{T}_1 \uplus \mathcal{T}_2 = \mathcal{T}$ and $\sigma \restriction_{\mathcal{T}_1}, \mathcal{T}_1 \vDash P$ and $\sigma \restriction_{\mathcal{T}_2}, \mathcal{T}_2 \vDash P$ then $\sigma, \mathcal{T} \vDash P$.

The second component of the conditions is a *potential function* Q, a mapping from stores and sets of variables X as described above to non-negative rational potentials. We use the potential function to track potential through a kernel to analyze resource usage. If $\sigma, \mathcal{T} \vdash X$, then $Q_X(\sigma)$ refers to the potential of σ under function Q, taking into account only the variables in X. Formally, we require (as a property of potential functions Q) that if for all $x \in X$ and $t \in \mathcal{T}$, we have $\sigma_1(x, t) = \sigma_2(x, t)$, then $Q_X(\sigma_1) = Q_X(\sigma_2)$. That is, Q can only consider the variables in X.

Intuitively, the potential function assigns a number (the *potential*) to each state of the program, as is standard in the "potential" (or "physicist's") method of amortized analysis. This potential will be constructed to decrease over time; the decrease in potential will be used to "pay for" program operations, and so we can use the initial potential to bound the cost of operations that occur during an execution. As a simple example of what a potential function might look like in practice, consider the loop example from the beginning of this section:

```
for (int i = N; i >=0; i--) { f(); }
```

As stated above, the potential function at the beginning of the loop body would be $i + 1$ and at the end of the loop body would be i. The potential function at the beginning of the program, which

[3] To aid in reasoning, you can read the shorthands as "σ is *compatible* with P" and "$\mathcal{T}$ is *compatible* with P."

sets up the potential for the first iteration of the loop, would be $N + 1$. The relation among these various potential functions is restricted by the rules of the quantitative Hoare logic, which we will describe in the remainder of the section, followed by a more faithful and complex example in Figure 9.

For a nonnegative rational cost C, we use the shorthand $Q + C$ to denote a potential function Q' such that for all σ and X, we have ${Q'}_X(\sigma) = Q_X(\sigma) + C$. We write $Q \geq Q'$ to mean that for all $\sigma, \mathcal{T}, X$ such that $\sigma, \mathcal{T} \vDash P$, we have $Q_X(\sigma) \geq {Q'}_X(\sigma)$.

In this section, we leave the concrete representation of the logical condition and the potential function abstract. In Section 7, we describe our implementation, including the representation of these conditions. For now, we make the assumptions stated above, as well as that logical conditions obey the standard rules of Boolean logic. We also assume that logical conditions and potential functions are equipped with an "assignment" operation $P' \Leftarrow P[x \leftarrow e]$ (respectively, $Q' \Leftarrow Q[x \leftarrow e]$) such that if $\sigma, \mathcal{T} \vDash P'$ and $\sigma; e \downarrow_M^{\mathcal{T}} R; C$, then

- $\sigma[(x, t) \mapsto R(t) \mid t \in \mathcal{T}], \mathcal{T} \vDash P$
- If $x \in X$ and there exists v such that $R(t) = v$ for all $t \in \mathcal{T}$, then ${Q'}_X(\sigma) = Q_X(\sigma[(x, t) \mapsto R(t) \mid t \in \mathcal{T}])$

In logical conditions for Hoare logic, the standard practice is to implement this operation by simply substituting e for x in P to obtain P'.[4] Reasonable representations of the potential function will also support such a substitution operation (e.g., in the example above where we used $i + 1$ as a potential function, we could substitute $i - 1$ for i to obtain i). For simplicity, we also assume that the potential function depends only on the values of local variables in the store and not on the values of arrays. This is sufficient to handle the benchmarks we studied. We write $\sigma; \mathcal{T} \vDash \{P; Q; X\}$ to mean $\sigma, \mathcal{T} \vDash P$ and $\sigma, \mathcal{T} \vdash X$.

Cost of Expressions. Before presenting the Hoare-style logic for statements, we introduce a simpler judgment that we use for describing the resource usage of operands and expressions. The judgment is written $P \vdash_M e : C$ and indicates that, under condition P, the evaluation of e costs at most C. The rules for this judgment are presented at the top of Figure 8. These rules are similar to those of Figure 5, with the exception that we now do not know the exact store used to evaluate the expression and must conservatively estimate the cost of array access based on the possible set of stores. We write $P \Rightarrow \text{MemReads}(o) \leq n$ to mean that for all σ and all $\mathcal{T}$ such that $\sigma, \mathcal{T} \vDash P$, if $\sigma; o \downarrow_M^{\mathcal{T}} R; C$, then $\text{MemReads}(R) \leq n$. The meaning of $P \Rightarrow \text{Conflicts}(o) \leq n$ is similar.

Inference Rules. The remainder of Figure 8 presents the inference rules for the Hoare-style logic for resource usage of statements. The judgment for these rules is written

$$\{P; Q; X\}\ s\ \{P'; Q'; X'\},$$

which states that if (1) P holds, (2) we have Q resources, and (3) all variables in X are thread-invariant, then if s terminates, it ends in a state where (1) P' holds, (2) we have Q' resources left over and (3) all variables in X' are thread-invariant. The simplest cases (e.g., Q:Skip and Q:Seq) simply thread conditions through without altering them (note that Q:Seq feeds the post-conditions of s_1 into the pre-conditions of s_2). Most other rules require additional potential in the pre-condition (e.g., $Q + C$), which is then discarded, because it is used to pay for an operation. For example, if s_1 uses C_1 resources and s_2 uses C_2 resources, then we might start with $Q + C_1 + C_2$, have $Q + C_2$ left in the post-condition of s_1, and have Q left in the post-condition of s_2.

The most notable rules are for conditionals if e then s_1 else s_2, which take into account the possibility of a divergent warp. There are four cases. First (Q:If1), we can statically determine that

[4]The intuition is that any fact that we want to be true of x after assigning e to x must be true of e before this assignment.

$$\text{(OQ:Var)}\ \frac{}{P \vdash_M x : M^{\mathsf{var}}} \qquad \text{(OQ:Const)}\ \frac{}{P \vdash_M c : M^{\mathsf{const}}} \qquad \text{(OQ:Param)}\ \frac{}{P \vdash_M p : M^{\mathsf{const}}} \qquad \text{(OQ:Tid)}\ \frac{}{P \vdash_M \mathsf{tid} : M^{\mathsf{var}}}$$

$$\text{(EQ:Op)}\ \frac{P \vdash_M o_1 : C_1 \qquad P \vdash_M o_2 : C_2}{P \vdash_M o_1\ \mathsf{op}\ o_2 : C_1 + C_2 + M^{\mathsf{op}}}$$

$$\text{(EQ:GArray)}\ \frac{P \vdash_M o : C \qquad P \Rightarrow \mathrm{MemReads}(o) \le n}{P \vdash_M G[o] : C + M^{\mathsf{gread}}(n)} \qquad \text{(EQ:SArray)}\ \frac{P \vdash_M o : C \qquad P \Rightarrow \mathrm{Conflicts}(o) \le n}{P \vdash_M S[o] : C + M^{\mathsf{sread}}(n)}$$

$$\text{(Q:Skip)}\ \frac{}{\vdash_M \{P; Q; X\}\ \mathsf{skip}\ \{P; Q; X\}} \qquad \text{(Q:If1)}\ \frac{\begin{array}{l}P \vdash_M e : C \qquad \vdash_M \{P \wedge e; Q; X\}\ s_1\ \{P'; Q'; X'\} \\ P \Rightarrow e\ \mathsf{unif} \qquad \vdash_M \{P \wedge \neg e; Q; X\}\ s_2\ \{P'; Q'; X'\}\end{array}}{\vdash_M \{P; Q + M^{\mathsf{if}} + C; X\}\ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \{P'; Q'; X'\}}$$

$$\text{(Q:If2)}\ \frac{P \vdash_M e : C \qquad P \Rightarrow e \qquad \vdash_M \{P; Q; X\}\ s_1\ \{P'; Q'; X'\}}{\vdash_M \{P; Q + M^{\mathsf{if}} + C; X\}\ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \{P'; Q'; X'\}} \qquad \text{(Q:If3)}\ \frac{P \vdash_M e : C \qquad P \Rightarrow \neg e \qquad \vdash_M \{P; Q; X\}\ s_2\ \{P'; Q'; X'\}}{\vdash_M \{P; Q + M^{\mathsf{if}} + C; X\}\ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \{P'; Q'; X'\}}$$

$$\text{(Q:If4)}\ \frac{\begin{array}{l}P \vdash_M e : C \qquad \vdash_M \{P \wedge e; Q; X\}\ s_1\ \{P_1; Q_1; X_1\} \qquad P \wedge \neg e \Rightarrow P' \qquad P_1 \Rightarrow P' \\ \{P'; Q_1; X_1 \setminus W(s_1)\}\ s_2\ \{P_2; Q_2; X_2\} \qquad P_1 \Rightarrow P'' \qquad P_2 \Rightarrow P'' \qquad X' = X_2 \setminus W(s_2)\end{array}}{\vdash_M \{P; Q + M^{\mathsf{if}} + C + M^{\mathsf{div}}; X\}\ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \{P''; Q_2; X'\}}$$

$$\text{(Q:Seq)}\ \frac{\vdash_M \{P; Q; X\}\ s_1\ \{P_1; Q_1; X_1\} \qquad \vdash_M \{P_1; Q_1; X_1\}\ s_2\ \{P'; Q'; X'\}}{\vdash_M \{P; Q; X\}\ s_1;\ s_2\ \{P'; Q'; X'\}}$$

$$\text{(Q:While1)}\ \frac{\begin{array}{l}P \Rightarrow e\ \mathsf{unif} \qquad P \vdash_M e : C \\ \vdash_M \{P \wedge e; Q; X\}\ s\ \{P; Q + M^{\mathsf{if}} + C; X\}\end{array}}{\vdash_M \{P; Q + M^{\mathsf{if}} + C; X\}\ \mathsf{while}\ (e)\ s\ \{P \wedge \neg e; Q; X\}} \qquad \text{(Q:While2)}\ \frac{P \vdash_M e : C \qquad X' = X \setminus W(s) \qquad \vdash_M \{P \wedge e; Q; X\}\ s\ \{P; Q + M^{\mathsf{if}} + M^{\mathsf{div}} + C; X\}}{\vdash_M \{P; Q + M^{\mathsf{if}} + M^{\mathsf{div}} + C; X\}\ \mathsf{while}\ (e)\ s\ \{P \wedge \neg e; Q; X'\}}$$

$$\text{(Q:VWrite1)}\ \frac{\begin{array}{l}x \in X \qquad P \Rightarrow e\ \mathsf{unif} \qquad P \vdash_M e : C \\ P \Leftarrow P'[x \leftarrow e] \qquad Q \Leftarrow Q'[x \leftarrow e]\end{array}}{\vdash_M \{P; Q + M^{\mathsf{vwrite}} + C; X\}\ x \leftarrow e\ \{P'; Q'; X\}} \qquad \text{(Q:VWrite2)}\ \frac{P \vdash_M e : C \qquad P \Leftarrow P'[x \leftarrow e]}{\vdash_M \{P; Q + M^{\mathsf{vwrite}} + C; X\}\ x \leftarrow e\ \{P'; Q; X \setminus \{x\}\}}$$

$$\text{(Q:GWrite)}\ \frac{P \vdash_M o : C_1 \qquad P \vdash_M e : C_2 \qquad P \Rightarrow \mathrm{MemReads}(o) \le n}{\vdash_M \{P; Q + M^{\mathsf{gwrite}}(n) + C_1 + C_2; X\}\ G[o] \leftarrow e\ \{P; Q; X\}} \qquad \text{(Q:SWrite)}\ \frac{P \vdash_M o : C_1 \qquad P \vdash_M e : C_2 \qquad P \Rightarrow \mathrm{Conflicts}(o) \le n}{\vdash_M \{P; Q + M^{\mathsf{swrite}}(n) + C_1 + C_2; X\}\ S[o] \leftarrow e\ \{P; Q; X\}}$$

$$\text{(Q:Weak)}\ \frac{\vdash_M \{P_2; Q_2; X_2\}\ s\ \{P_2'; Q_2'; X_2'\} \qquad P_1 \Rightarrow P_2 \qquad Q_1 \ge Q_2 \qquad X_1 \supset X_2 \qquad P_2' \Rightarrow P_1' \qquad Q_2' \ge Q_1' \qquad X_2' \supset X_1'}{\vdash_M \{P_1; Q_1 + C; X_1\}\ s\ \{P_1'; Q_1' + C; X_1'\}}$$

Fig. 8. Hoare logic rules for resource analysis.

the conditional expression e does not vary across a warp: this is expressed with the premise $P \Rightarrow e\ \mathsf{unif}$, which is shorthand for

$$\forall \sigma, \mathcal{T}.\ \sigma, \mathcal{T} \vDash P \Rightarrow \exists c.\ \sigma; e \downarrow_M^{\mathcal{T}} (c)_{t \in \mathcal{T}}; C.$$

That is, for any compatible store, e evaluates to a constant result family. In this case, only one branch is taken by the warp and the cost of executing the conditional is the maximum cost of executing the two branches (plus the cost M^{if} of the conditional and the cost C of evaluating the expression, which are added to the precondition). This is expressed by using Q' as the potential function in the post-condition for both branches. If the two branches do not use equal potential, then the one that has more potential "left over" may use rule Q:Weak (discussed in more detail later) to discard its extra potential and use Q' as a post-condition. We thus conservatively approximate the potential left over after executing one branch. In the next two cases (Q:If2 and Q:If3), we are able to statically determine that the conditional expression is either true or false in any compatible store (i.e., either $P \Rightarrow e$ or $P \Rightarrow \neg e$), and we need only the "then" or "else" branch, so only the respective branch is considered in these rules.

In the final case (Q:If4), we consider the possibility that the warp may diverge. In addition to accounting for the case where we must execute s_1 followed by s_2 in sequence, this rule must also subsume the three previous cases, as it is possible that we were unable to determine statically that

the conditional would not diverge (i.e., we were unable to derive the preconditions of Q:If1) but the warp does not diverge at runtime. To handle both cases, we require that the precondition of s_2 is implied by:

— $P \wedge \neg e$, the precondition of the conditional together with the information that e is false, so that s_2 can execute by itself if the conditional does not diverge, as well as by
— P_1, the postcondition of s_1, so that s_2 can execute sequentially after s_1.

In a similar vein, we require that the postcondition of the whole conditional is implied by the individual postconditions of both branches. In addition, we remove from X_1 the set of variables possibly written to by s_1 (denoted $W(s_1)$, this can be determined syntactically), because if the warp diverged, variables written to by s_1 no longer have consistent values across the entire warp. We similarly remove $W(s_2)$ from X_2.

Note that it is always sound to use rule Q:If4 to check a conditional. However, using this rule in all cases would produce a conservative over-estimate of the cost by assuming a warp diverges even if it can be shown that it does not. Our inference algorithm will maximize precision by choosing the most precise rule that it is able to determine to be sound.

The rules Q:While1 and Q:While2 charge the initial evaluation of the conditional ($M^{\text{if}} + C$) to the precondition. For the body of the loop, as with other Hoare-style logics, we must derive a loop invariant: the condition P must hold at both the beginning and end of each iteration of the loop body (we additionally know that e holds at the beginning of the body). In addition, the potential after the loop body must be sufficient to "pay" $M^{\text{if}} + C$ for the next check of the conditional, and still have potential Q remaining to execute the next iteration if necessary. Recall that Q is a function of the store. So this premise requires that the value of a store element (e.g., a loop counter) change sufficiently so that the corresponding decrease in potential $Q_X(\sigma)$ is able to pay the appropriate cost. The difference between the two rules is that Q:While1 assumes the warp does not diverge, so we need not pay M^{div} and also need not remove variables assigned by the loop body from X.

The rules for local assignment are an extension of the standard rule for assignment in Hoare logic. If $x \in X$ and $P \Rightarrow e$ unif, then we add a symmetric premise for the potential function. Otherwise, we cannot use x as a potential-carrying variable and only update the logical condition. The rules for array assignments are similar to those for array accesses, but additionally include the cost of the assigned expression e.

Finally, as discussed above, Q:Weak allows us to strengthen the preconditions and weaken the postconditions of a derivation. If s can execute with precondition $\{P_2; Q_2; X\}$ and postcondition $\{P_2'; Q_2'; X\}$, then it can also execute with a precondition P_1 that implies P_2 and a potential function Q_1 that is always greater than Q_2. In addition, it can guarantee any postcondition implied by P_2' and any potential function Q_1' that is always less than Q_2'. We can also take subsets of X as necessary in derivations. The rule also allows us to add a constant potential to both the pre- and post-conditions.

Example 3. Figure 9 steps through a derivation for the `addSub3` kernel from Section 2, with the pre- and post-conditions interspersed in red. The code has been translated to the syntax of miniCUDA, except that we use nested expressions and use expressions as array indices. Both of these features are syntactic sugar that are easily expanded to proper miniCUDA. Their use does not impact the derivation and is done solely for readability. For illustrative purposes, we consider only the costs of array accesses (writes and reads) and assume all other costs are zero. The potential annotation consists of two parts: the *constant* potential and a component that is proportional to the value of $h - j$ (initially we write this as just h, because j is undefined). The initial constant potential is consumed by the write on line 1, which involves a global memory access with four separate

		$L \triangleq 2M^{\mathsf{sread}}(1) + 2M^{\mathsf{gwrite}}(5)$
		$\{\top; M^{\mathsf{swrite}}(1) + M^{\mathsf{gread}}(4) + \frac{h}{2}L; \{w,h,j\}\}$
1	$As[\mathsf{tid}\%32] \leftarrow A[\mathsf{tid}]$	$\{\top; \frac{h}{2}L; \{w,h,j\}\}$
2	$j \leftarrow 0$	$\{\top; \frac{h-j}{2}L; \{w,h,j\}\}$
3	while $(j < h)$	$\{j < h; \frac{h-j}{2}L; \{w,h,j\}\}$
4	$\quad B[j*w+\mathsf{tid}] \leftarrow B[j*w+\mathsf{tid}] + As[\mathsf{tid}]$	$\{j < h; M^{\mathsf{sread}}(1) + M^{\mathsf{gwrite}}(5) + \frac{h-j-2}{2}L; \{w,h,j\}\}$
5	$\quad B[(j+1)*w+\mathsf{tid}] \leftarrow B[j*w+\mathsf{tid}] - As[\mathsf{tid}]$	$\{j < h; \frac{h-j-2}{2}L; \{w,h,j\}\}$
6	$\quad j \leftarrow j+2$	$\{j < h; \frac{h-j}{2}L; \{w,h,j\}\}$
		$\{j \geq h; 0; \{w,h,j\}\}$

Fig. 9. Derivation using the program logic. We define L to be $2M^{\mathsf{sread}}(1) + 2M^{\mathsf{gwrite}}(5)$.

reads (128 consecutive bytes with 32-byte reads[5] and a shared write with no bank conflicts. The information needed to determine the number of global memory sectors read and the number of bank conflicts is encoded in the logical conditions, which we leave abstract for now; we will discuss in Section 7 how this information is encoded and used. On line 2, we set j to 0, and rewrite the potential using $h - j$ instead of j (because $j = 0$, this is not a change in potential). On line 3, we establish the invariant of the loop body. On line 4, we transfer L to the constant potential (this is accomplished by Rule Q:Weak). We then spend part of this on the assignment on line 4 and the rest on line 5. These require five global reads each, because we read 128 bytes of consecutive memory with 32-byte reads and the first index is not aligned to a 32-byte boundary. This establishes the correct potential for the next iteration of the loop on line 6 after j is incremented. After the loop, we conclude $j \geq h$ and have no remaining potential.

Coq Mechanization. Our Coq mechanization assumes types `Pctx` and `Qctx` as `Parameters` representing logical conditions P and potential functions Q, respectively. We also assume any necessary operations on these types, as well as assumptions (such as the existence of assignment functions), as parameters. Inductive definitions `copd`, `cexp`, and `cstmt` formalize the judgments for proving quantitative Hoare triples on the cost of operands, expressions, and statements, respectively, given in Figure 8. The inductive definitions closely follow the inference rules.

Soundness. We prove that if there is a derivation under the analysis showing that a program can execute with precondition $\{P; Q; X\}$, then for any store σ and any set of threads $\mathcal{T}$ such that $\sigma; \mathcal{T} \vDash \{P; Q; X\}$, the cost of executing the program under σ and threads $\mathcal{T}$ is at most $Q_X(\sigma)$. We first state the soundness result of the resource analysis for expressions.

Lemma 2. *If* $\Sigma \vdash e : \tau$ *and* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and* $P \vdash_M e : C$ *and* $\sigma, \mathcal{T} \vDash P$ *and* $\sigma; e \downarrow_M^{\mathcal{T}} R; C'$, *then* $C' \leq C$.

Proof. By induction on the derivation of $\sigma; e \downarrow_M^{\mathcal{T}} R; C'$. The proof is fully formalized in Coq as a lemma `cexp_sound` involving the propositions `eval_exp` and `cexp` (which uses a helper lemma about operands). In this article, we will give sketches of the approach used to solve non-trivial cases.

— EC:Op. Then $\sigma; o_1 \downarrow_M^{\mathcal{T}} R_1; C_1'$ and $\sigma; o_2 \downarrow_M^{\mathcal{T}} R_2; C_2'$ and $C' = C_1' + C_2' + M^{\mathrm{op}}$. By inversion on EQ:Op, we have $P \vdash_M o_1 : C_1$ and $P \vdash_M o_2 : C_2$ and $C = C_1 + C_2 + M^{\mathrm{op}}$. By induction, $C_1' \leq C_1$ and $C_2' \leq C_2$.

[5]The amount of memory accessed by a single read is hardware-dependent and complex; this is outside the scope of this article.

— EC:GArr. Then $\sigma; o \downarrow_M^{\mathcal{T}} R; C'_o$ and

$$C' = C'_o + M^{\text{gread}}(\text{MemReads}(R)).$$

By inversion on EQ:GArray, we have $P \vdash_M o : C_o$ and $C = C_o + M^{\text{gread}}(\text{MemReads}(n))$ and $\text{MemReads}(R) \leq n$. By monotonicity of M, this means $M^{\text{gread}}(\text{MemReads}(R)) \leq M^{\text{gread}}(n)$. By induction, $C'_o \leq C_o$.

— EC:SArr. Then $\sigma; o \downarrow_M^{\mathcal{T}} R; C'_o$ and $C' = C'_o + M^{\text{sread}}(\text{Conflicts}(R))$. By inversion on EQ:SArray, we have $P \vdash_M o : C_o$. and $C = C_o + M^{\text{sread}}(n)$ and $\text{Conflicts}(R) \leq n$. By monotonicity of M, we have $M^{\text{sread}}(\text{Conflicts}(R)) \leq M^{\text{sread}}(n)$ and by induction, $C'_o \leq C_o$, so $C' \leq C$. □

The following lemma is necessary for the soundness proof for statements.

Lemma 3. *If* $\{P; Q; X\}\ s\ \{P'; Q'; X'\}$, *then* $X' \subset X$ *and* $Q \geq Q'$.

Proof. The two conclusions are proven in Coq as the lemmas `cstmt_mono_X` and `cstmt_mono_Q`, respectively. Both proofs are straightforward inductions on the derivation of $\{P; Q; X\}\ s\ \{P'; Q'; X'\}$ and most obligations are automatically discharged by Coq. □

Theorem 1. *If* $\Sigma \vdash s$ *and* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and* $\{P; Q; X\}\ s\ \{P'; Q'; X'\}$ *and* $\sigma; \mathcal{T} \vDash \{P; Q; X'\}$ *and* $\sigma; s \Downarrow_M^{\mathcal{T}} \sigma'; C_s$, *then* $\sigma'; \mathcal{T} \vDash \{P'; Q'; X'\}$ *and* $Q_X(\sigma) - C_s \geq Q'_{X'}(\sigma') \geq 0$.

Proof. Proven in Coq as the theorem `cstmt_sound` by induction on the derivation of $\{P; Q; X\}\ s\ \{P'; Q'; X'\}$. Here, we sketch the arguments used to prove non-trivial cases.

- Q:If1. By inversion, $P \vdash_M e : C$ and $\{P \wedge e; Q; X\}\ s_1\ \{P'; Q'; X'\}$ and $\{P \wedge \neg e; Q; X\}\ s_2\ \{P'; Q'; X'\}$. We have $\sigma; e \downarrow_M^{\mathcal{T}} R; C_1$. By Lemma 2, we have $C_1 \leq C$. By inversion, either $R = (True)_{t \in \mathcal{T}}$ or $R = (False)_{t \in \mathcal{T}}$. We consider the first case. In this case, $\sigma, \mathcal{T} \vDash P \wedge e$, so $\sigma; \mathcal{T} \vDash \{P \wedge e; Q; X\}$. We also have $\sigma; s_1 \Downarrow_M^{\mathcal{T}} \sigma'; C_2$. By induction, $\sigma'; \mathcal{T} \vDash \{P'; Q'; X'\}$ and $Q_X(\sigma) - C_2 \geq Q'_{X'}(\sigma') \geq 0$, so $(Q + M^{\text{if}} + C)_X(\sigma) - C_1 - M^{\text{if}} - C_2 \geq Q_X(\sigma) - C_2 \geq Q'_{X'}(\sigma') \geq 0$. The other case is symmetric to the above.
- Q:If4. By inversion, $P \vdash_M e : C$ and $\{P \wedge e; Q; X\}\ s_1\ \{P_1; Q_1; X_1\}$ and $\{P'; Q_1; X_1 \setminus W(s_1)\}\ s_2\ \{P_2; Q_2; X_2\}$. We also have $\sigma; e \downarrow_M^{\mathcal{T}} R; C_1$. By Lemma 2, we have $C_1 \leq C$. Let $\mathcal{T}_T = \{t \in \mathcal{T} \mid R(t)\}$ and $\mathcal{T}_F = \{t \in \mathcal{T} \mid \neg R(t)\}$. Proceed in cases.
  - $\mathcal{T}_T = \mathcal{T}$. Then, by inversion on SC:IfT, $\sigma; s_1 \Downarrow_M^{\mathcal{T}} \sigma_1; C_2$. We have $\sigma, \mathcal{T} \vDash P \wedge e$, so $\sigma; \mathcal{T} \vDash \{P \wedge e; Q; X\}$. By induction, $\sigma_1; \mathcal{T} \vDash \{P_1; Q_1; X_1\}$ and $Q_X(\sigma) - C_2 \geq Q_{1X_1}(\sigma_1) \geq 0$. Because $P_1 \Rightarrow P''$, we have $\sigma_1, \mathcal{T} \vDash P''$. By Lemma 3, $X_2 \subset X_1$ so $\sigma_1; \mathcal{T} \vDash \{P''; Q''; X''\}$. We have $(Q + M^{\text{if}} + C + M^{\text{div}})_X(\sigma) - C_1 - M^{\text{if}} - C_2 \geq Q_X(\sigma) - C_2 \geq Q_{2X_2}(\sigma_1) \geq 0$.
  - $\mathcal{T}_F = \mathcal{T}$. Then, by inversion on SC:IfF, $\sigma; s_2 \Downarrow_M^{\mathcal{T}} \sigma_1; C_2$. Because $P \wedge \neg e \Rightarrow P'$, we have $P'(\sigma)$. By Lemma 3, $X_1 \subset X$ and $Q \geq Q_1$, so we have $\sigma; \mathcal{T} \vDash \{P'; Q_1; X \setminus W(s_1)\}$. By induction, $\sigma_1; \mathcal{T} \vDash \{P_2; Q_2; X_2\}$ and $Q_X(\sigma) - C_2 \geq Q_{2X_2}(\sigma_1) \geq 0$. Because $P_2 \Rightarrow P''$, we have $\sigma_1, \mathcal{T} \vDash P''$ and $\sigma_1; \mathcal{T} \vDash \{P''; Q_2; X_2 \setminus W(s_2)\}$. We have $(Q + M^{\text{if}} + C + M^{\text{div}})_X(\sigma) - C_1 - M^{\text{if}} - C_2 \geq Q_X(\sigma) - C_2 \geq Q_{2X_2}(\sigma_1) \geq 0$.
  - $\mathcal{T}_T, \mathcal{T}_F \neq \emptyset$. Then, by inversion on SC:IfD, $\sigma; s_1 \Downarrow_M^{\mathcal{T}_T} \sigma_1; C_2$ and $\sigma_1; s_2 \Downarrow_M^{\mathcal{T}_F} \sigma_2; C_3$. We have $\sigma, \mathcal{T}_T \vDash P \wedge e$, so $\sigma; \mathcal{T}_T \vDash \{P \wedge e; Q; X\}$. By induction, $\sigma_1; \mathcal{T}_T \vDash \{P_1; Q_1; X_1\}$ and $Q_X(\sigma) - C_2 \geq Q_{1X_1}(\sigma_1)$. Because $P \wedge \neg e \Rightarrow P'$, we have $\sigma_1 \restriction_{\mathcal{T}_F}, \mathcal{T}_F \vDash P'$. Thus, $\sigma_1; \mathcal{T}_F \vDash \{P'; Q_1; X_1 \setminus W(s_1)\}$. By induction, $\sigma_2; \mathcal{T}_F \vDash \{P_2; Q_2; X_2\}$ and $Q_{1X \setminus W(s_1)}(\sigma_1) - C_3 \geq Q_{2X_2}(\sigma_2) \geq 0$. Because $P_1 \Rightarrow P''$, we have $\sigma_2 \restriction_{\mathcal{T}_T}, \mathcal{T}_T \vDash P''$ and because $P_2 \Rightarrow P''$, we

have $\sigma_2 \restriction_{\mathcal{T}_F}, \mathcal{T}_F \vDash P''$, so $\sigma; \mathcal{T} \vDash \{P''; Q_2; X_2 \setminus W(s_2)\}$. We have $(Q + M^{\text{if}} + C + M^{\text{div}})_X(\sigma) - C_1 - M^{\text{if}} - C_2 - C_3 - M^{\text{div}} \geq Q_X(\sigma) - C_2 - C_3 \geq Q_{1X_1}(\sigma_1) - C_3 \geq Q_{2X_2}(\sigma_2) \geq Q_{2X_2 \setminus W(s_2)}(\geq)0$.

- Q:While2. By inversion, $\{P \wedge e; Q; X\}\, s\, \{P; Q + M^{\text{if}} + C; X\}$. Proceed by an inner induction on the derivation of $\sigma; \text{while } (e)\ s \Downarrow^{\mathcal{T}}_M \sigma'; C_s$.
  - SC:WhileAll. Then $\sigma; e \downarrow^{\mathcal{T}}_M (True)_{t \in \mathcal{T}}; C_1$ and $\sigma; s \Downarrow^{\mathcal{T}}_M \sigma_1; C_2$ and $\sigma; \text{while } (e)\ s \Downarrow^{\mathcal{T}}_M \sigma_2; C_3$. By Lemma 2, $C_1 \leq C$. We have $\sigma; \mathcal{T} \vDash \{P \wedge e; Q; X\}$. By the outer induction hypothesis, $\sigma_1; \mathcal{T} \vDash \{P; Q + M^{\text{if}} + C + M^{\text{div}}; X\}$ and $Q_X(\sigma) - C_2 \geq (Q + M^{\text{if}} + C + M^{\text{div}})_X(\sigma_1) \geq 0$. By the inner induction hypothesis, $\sigma_2; \mathcal{T} \vDash \{P \wedge \neg e; Q; X\}$ and $(Q + M^{\text{if}} + C + M^{\text{div}})_X(\sigma_1) - C_3 \geq Q_X(\sigma_2)$. We thus have $\sigma_2; \mathcal{T} \vDash \{P \wedge \neg e; Q; X \setminus W(s)\}$ and $(Q + M^{\text{if}} + C + M^{\text{div}})_X(\sigma) - C_1 - C_2 - M^{\text{if}} - C_3 \geq Q_X(\sigma) - C_2 - C_3 \geq (Q + M^{\text{if}} + M^{\text{if}} + M^{\text{div}})_X(\sigma_1) - C_3 \geq Q_X(\sigma_2) \geq Q_{X \setminus W(s)}(\sigma_2)$.
  - SC:WhileSome. Then $\sigma; e \downarrow^{\mathcal{T}}_M R; C_1$ and $\sigma; s \Downarrow^{\mathcal{T}_T}_M \sigma_1; C_2$ and $\sigma; \text{while } (e)\ s \Downarrow^{\mathcal{T}_T}_M \sigma_2; C_3$. By Lemma 2, $C_1 \leq C$. We have $\sigma; \mathcal{T}_T \vDash \{P \wedge e; Q; X\}$. By the outer induction hypothesis, $\sigma_1; \mathcal{T}_T \vDash \{P; Q + M^{\text{if}} + C + M^{\text{div}}; X\}$ and $Q_X(\sigma) - C_2 \geq (Q + M^{\text{if}} + C + M^{\text{div}})_X(\sigma_1) \geq 0$. By the inner induction hypothesis, $\sigma_2; \mathcal{T}_T \vDash \{P \wedge \neg e; Q; X\}$ and $(Q + M^{\text{if}} + C + M^{\text{div}})_X(\sigma_1) - C_3 \geq Q_X(\sigma_2)$. We have $\sigma_2 \restriction_{\mathcal{T} \setminus \mathcal{T}_T}, \mathcal{T} \setminus \mathcal{T}_T \vDash P \wedge \neg e$. We thus have $\sigma_2; \mathcal{T} \vDash \{P \wedge \neg e; Q; X \setminus W(s)\}$ and $(Q + M^{\text{if}} + C + M^{\text{div}})_X(\sigma) - C_1 - C_2 - M^{\text{if}} - C_3 \geq Q_X(\sigma) - C_2 - C_3 \geq (Q + M^{\text{if}} + M^{\text{if}} + M^{\text{div}})_X(\sigma_1) - C_3 \geq Q_X(\sigma_2) \geq Q_{X \setminus W(s)}(\sigma_2)$.
  - SC:WhileNone. Then $\sigma; e \downarrow^{\mathcal{T}}_M (False)_{t \in \mathcal{T}}; C_1$ and by Lemma 2, $C_1 \leq C$. We have $\sigma, \mathcal{T} \vDash P \wedge \neg e$, so $\sigma; \mathcal{T} \vDash \{P \wedge \neg e; Q; X \setminus W(s)\}$. We also have $(Q + M^{\text{if}} + M^{\text{div}} + C)_X(\sigma) - C_1 - M^{\text{if}} \geq Q_X(\sigma) \geq 0$.
- Q:Seq. By inversion, $\{P; Q; X\}\, s_1\, \{P_1; Q_1; X_1\}$ and $\{P_1; Q_1; X_1\}\, s_2\, \{P'; Q'; X'\}$. We have $\sigma; s_1 \Downarrow^{\mathcal{T}}_M \sigma_1; C_1$ and $\sigma_1; s_2 \Downarrow^{\mathcal{T}}_M \sigma_2; C_2$. By induction, $\sigma_1; \mathcal{T} \vDash \{P_1; Q_1; X_1\}$ and $Q_X(\sigma) - C_1 \geq Q_{1X_1}(\sigma_1) \geq 0$ and $\sigma_2; \mathcal{T} \vDash \{P'; Q'; X'\}$ and $Q_{1X_1}(\sigma_1) - C_2 \geq Q'_{X'}(\sigma_2) \geq 0$.
- Q:VWrite1. By inversion, $P \vdash_M e : C$ and $P \Leftarrow P'[x \leftarrow e]$ and $Q \Leftarrow Q'[x \leftarrow e]$ and $\sigma' = \sigma[(x,t) \mapsto R(t) \mid t \in \mathcal{T}]$. We also have $\sigma; e \downarrow^{\mathcal{T}}_M R; C_1$. By Lemma 2, we have $C_1 \leq C$. By assumption, $\sigma', \mathcal{T} \vDash P'$ and $Q + M^{\text{vwrite}} + C_X(\sigma) - M^{\text{vwrite}} - C_1 \geq Q_X(\sigma) = Q'_X(\sigma') \geq 0$.
- Q:GWrite. By inversion, $P \vdash_M o : C_1$ and $P \vdash_M e : C_2$ and $P \Rightarrow \text{MemReads}(o) \leq n$. We have $\sigma; e_1 \downarrow^{\mathcal{T}}_M R_1; C'_1$ and $\sigma; e_2 \downarrow^{\mathcal{T}}_M R_2; C'_2$ and $\sigma' = \sigma[(G, R_1(t)) \mapsto R_2(t) \mid t \in \mathcal{T}]$. By Lemma 2, $C'_1 \leq C_1$ and $C'_2 \leq C_2$. By assumption, $\text{MemReads}(R_1) \leq n$, so $M^{\text{gwrite}}(\text{MemReads}(R_1)) \leq M^{\text{gwrite}}(n)$. We have $\sigma'; \mathcal{T}' \vDash \{P; Q; X\}$ and $Q + M^{\text{gwrite}}(n) + C_1 + C_{2X}(\sigma) - C_1 - C_2 - M^{\text{gwrite}}(\text{MemReads}(R_1)) \geq Q_X(\sigma') \geq 0$.
- Q:SWrite. By inversion, $P \vdash_M o : C_1$ and $P \vdash_M e : C_2$ and $P \Rightarrow \text{Conflicts}(o) \leq n$. We have $\sigma; e_1 \downarrow^{\mathcal{T}}_M R_1; C'_1$ and $\sigma; e_2 \downarrow^{\mathcal{T}}_M R_2; C'_2$ and $\sigma' = \sigma[(S, R_1(t)) \mapsto R_2(t) \mid t \in \mathcal{T}]$. By Lemma 2, $C'_1 \leq C_1$ and $C'_2 \leq C_2$. By assumption, $\text{Conflicts}(R_1) \leq n$, so $M^{\text{swrite}}(\text{Conflicts}(R_1)) \leq M^{\text{swrite}}(n)$. We have $\sigma'; \mathcal{T}' \vDash \{P; Q; X\}$ and $Q + M^{\text{swrite}}(n) + C_1 + C_{2X}(\sigma) - C_1 - C_2 - M^{\text{swrite}}(\text{Conflicts}(R_1)) \geq Q_X(\sigma') \geq 0$.
- Q:Weak. By inversion, $\{P_2; Q_2; X_2\}\, s\, \{P'_2; Q'_2; X'_2\}$ and $P_1 \Rightarrow P_2$ and $Q_1 \geq Q_2$ and $X_1 \supset X_2$ and $P'_2 \Rightarrow P'_1$ and $Q'_2 \geq Q'_1$ and $X'_2 \supset X'_1$. We have $\sigma; \mathcal{T} \vDash \{P_2; Q_2; X_2\}$, so by induction, $\sigma'; \mathcal{T} \vDash \{P'_2; Q'_2; X'_2\}$ and $Q_{2X_2}(\sigma) - C_s \geq Q'_{2X'_2}(\sigma') \geq 0$. We then have $\sigma'; \mathcal{T} \vDash \{P'_1; Q'_1; X'_1\}$ and $Q'_{2X'_2}(\sigma') \geq Q'_{1X'_1}(\sigma')$. □

5 LATENCY AND THREAD-LEVEL PARALLELISM

The analysis developed in the previous section is useful for predicting cost metrics of CUDA kernels such as divergent warps, global memory accesses and bank conflicts, the three performance

bottlenecks that are the primary focus of this article: one can specify a resource metric that counts the appropriate operations, run the analysis to determine the maximum cost for a warp and multiply by the number of warps that will be spawned to execute the kernel (this is specified in the main program when the kernel is called). If we wish to work toward predicting actual execution time of a kernel, then the story becomes more complex; we begin to explore this question in this section. A first approach would be to determine, via profiling, the runtime cost of each operation and run the analysis with a cost metric that assigns appropriate costs. Such an approach might approximate the execution time of a single warp, but it is not immediately clear how to compose the results to account for multiple warps, unlike divergences or memory accesses, which we simply sum together.

Indeed, the question of how execution times of warps compose is a complex one because of the way in which GPUs schedule warps. Each ***Streaming Multiprocessor (SM)***, the computation units of the GPU, can execute instructions on several warps simultaneously, with the exact number dependent on the hardware. However, when a kernel is launched, CUDA assigns each SM a number of threads that is generally greater than the number it can simultaneously execute. It is profitable to do so, because many instructions incur some amount of latency after the instruction is executed. For example, if a warp executes a load from memory that takes 16 cycles, the SM can use those 16 cycles to execute instructions on other warps. At each cycle, the SM selects as many warps as possible that are ready to execute an instruction and issues instructions on them.

To predict the execution time of a kernel, we must therefore reason about both the number of instructions executed and their latency. In this section, we show how our existing analysis can be used to derive these quantities and, from them, approximate execution time bounds on a block of a CUDA kernel (we choose the block level for this analysis, because it is the granularity at which synchronization occurs and so composing execution times between blocks is more straightforward).

To derive such an execution time bound, we leverage a result from the field of parallel scheduling [Muller and Acar 2016], which predicts execution times of programs based on their *work*, the total computational cost (not including latency) of operations to be performed, and *span*, the time required to perform just the operations along the critical path (including latency). One can think of the work as the time required to execute a program running only one thread at a time, and the span as the time required to execute the program running all threads at the same time (assuming infinitely parallel hardware). Given these two quantities, it is possible to bound the execution time of a program assuming a *greedy* scheduler [Eager et al. 1989], that is, one that does not leave processors idle when there is work available. The bound is given by Theorem 1 of Muller and Acar [2016], stated below, which itself extends results of Eager et al. [1989] and Brent [1974].

Theorem 2. *Consider a parallel computation with work W and span S. Any greedy schedule of this computation for P workers has length at most $\frac{W}{P} + S$.*

The proof of this theorem, as well as the formal definitions of work, span, and schedules, are outside the scope of this article, but Section 10 gives pointers to the relevant related work.

At an abstract level, the theorem can apply directly to the CUDA setting. In our CUDA setting, the work is the total number of instructions executed by the kernel across all warps and the span is the maximum time required to execute any warp from start to finish. The assumption of a greedy schedule requires that an SM issues instructions on as many warps as possible each cycle. With these analogies, we can instantiate Theorem 2 to state the following bound:

Corollary 1. *Suppose a block of a kernel has work W and span S and that the block is scheduled on an SM that can issue P instructions at a time. Then, the time required by the SM to execute the block is at most $\frac{W}{P} + S$.*

Proof. This is a direct result of Theorem 1 of Reference [Muller and Acar 2016], which shows the same bound for a general computation of work W and span S under a greedy schedule on P processors. □

Note that the purpose of Corollary 1 is mainly to serve as a goalpost: it gives a bound that (at least in approximation) should be achievable with idealized assumptions about the behavior of GPU execution. Showing that this is achievable under a more precise model of GPU execution is the purpose of Section 6. The remainder of this section further develops the notions of work and span in the setting of CUDA.

We can, and will, use our analysis of the previous sections to independently calculate the work and span of a warp. Recall that the lockstep semantics of Section 4, and therefore the quantitative Hoare logic based on it, was designed to estimate the cost of a single warp. It turns out, however, that if we run the lockstep semantics on multiple warps (by including all of the desired threads in the set $\mathcal{T}$), the resulting cost is a slight overestimate of the actual work or span that we obtain from using a semantics that is designed for multiple warps—we show this fact later in this section by developing such a "parallel" semantics and formally comparing the two semantics. Because the lockstep semantics soundly overapproximates the cost given by the parallel semantics, the logic of Section 4, which soundly approximates the cost of the lockstep semantics, is sound with respect to the parallel semantics as well.

The goal of this section is to make the above intuition formal. First, however, we give an example that demonstrates why such a fact is neither obvious nor always guaranteed. The reason stems from the fact that warps of a block can synchronize with each other using the `__syncthreads()` built-in function,[6] which acts as a barrier forcing all warps to wait to proceed until all warps have reached the synchronization point. Consider the following code:

```
__global__ void tricky(int *A) {
  if (threadIdx.x < 32) {
    A[threadIdx.x] = 42;
  }
  __syncthreads();
  if (threadIdx.x >= 32) {
    A[threadIdx.x] = 42;
  }
}
```

Assume that the latency of the write to global memory dominates the span of the computation. Each warp performs only one write, and so analyzing the span of the program for one warp (either threads 0–31 or 32–63) would count the latency of only one write. However, because of the synchronization in the middle, the span of the block must actually account for the latency of two writes: threads 32 and up must wait for threads 0 through 31 to perform their writes before proceeding. There would be no way to know how to find the correct span for the block by composing the spans of the two warps, without more detailed knowledge of the code. Instead, we perform the analysis using the quantitative Hoare logic on threads 0–63 at once; this assumes that all 64 threads run in lockstep, which is not accurate. However, in this case (and, as we show, in all cases), this assumption results in a sound approximation: both conditionals are analyzed as potentially divergent conditionals, in which the inactive threads wait for the active threads. Note here that this convenience comes with some loss of precision: in the case of this code, the cost may (depending on the

[6]We have not mentioned `__syncthreads()` up to this point, because it was not particularly relevant for the warp-level analysis, but it is supported by our implementation of miniCUDA and used in many of the benchmarks.

chosen resource metric) include the cost of two divergent warps, when indeed neither conditional results in a divergent warp, because all of the threads *within one warp* take the same branch.

We now proceed to develop a *parallel* cost semantics, motivated above, that models entire CUDA blocks and tracks the cost in terms of work and span. The cost semantics tracks the work and span for each warp. At each synchronization, we take the maximum span over all warps to account for the fact that all warps in the block must wait for each other at that point. A cost is now a pair (c^w, c^s) of the work and span, respectively. We use the functions fst and snd to take the left and right components of a pair, respectively. A resource metric M maps resource constants to costs reflecting the number of instructions and latency required by the operation. We can take projections M_w and M_s of such a resource metric, which projects out the work and span components, respectively, of each cost. For the purposes of calculating the span, we assume that the span of an operation (the second component of the cost) reflects the time taken to process the instruction plus the latency (in other words, the latency is the span of the operation minus the work). We represent the cost of a block as a warp-indexed family of costs C. We use $\emptyset$ to denote the collection $((0,0)_{i \in Warps})$, where *Warps* is the set of all warps in the block. To add a cost (c^w, c^s) to a collection of costs for just the warps in the set W, we use the shorthand $C \oplus_W (c^w, c^s)$, defined as follows:

$$(C \oplus_W (c^w, c^s))_i \triangleq \begin{cases} (c_0^w + c^w, c_0^s + c^s) & C_i = (c_0^w, c_0^s) \wedge i \in W, \\ C_i & \text{otherwise.} \end{cases}$$

We will overload the above notation to add a cost onto a collection for a subset of threads:

$$C \oplus_{\mathcal{T}} C \triangleq C \oplus_{\{WarpOf(t) | t \in \mathcal{T}\}} C,$$

where $WarpOf(t)$ is the warp containing thread t. For the purposes of the present definition, warps may be thought of as abstract objects and so $WarpOf(t)$ is a mapping from threads to warps preserving the property that each warp contains exactly 32 threads. In our Coq formalization, warps are identified by integers and so we define $WarpOf(t) \triangleq Tid(t) \bmod 32$ where $Tid(t)$ is the integer thread identifier of t.

We denote the work of a collection by $W(C)$ and the span by $S(C)$. We can calculate the work and span of a block by summing and taking the maximum, respectively, over the warps:

$$\begin{aligned} W(C) &\triangleq \Sigma_{i \in Warps} \mathsf{fst}\ C_i, \\ S(C) &\triangleq \max_{i \in Warps} \mathsf{snd}\ C_i. \end{aligned}$$

Note that here it is safe to take the maximum span over the warps, because we have also done so at each synchronization point.

The cost semantics we describe is presented as a big-step operational semantics that evaluates a block of threads in parallel (as opposed to a single warp in lock-step, like the operational semantics of Section 3). However, unlike standard big-step semantics (including both the warp-level semantics of Section 3 and the parallel block-level semantics presented in the conference version of this article [Muller and Hoffmann 2021]), the semantics does not fully evaluate a program to completion. Instead, it evaluates only until the new statement sync (which is the miniCUDA representation of `__syncthreads`) is encountered, or until the program terminates if no sync is encountered. Because all warps synchronize at each sync and execute in parallel in between, we may divide the execution of a kernel on a block of threads into several segments delimited by calls to sync; all threads run each segment to completion before synchronizing, and then all threads move at once to the next segment. The cost semantics we present therefore models the execution of one of these segments. Stopping at segment boundaries rather than proceeding until the kernel returns makes it easier to prove the correspondence with the bounded implementation, and also makes it clearer when we compose the spans of each segment. The result of evaluation is the continuation

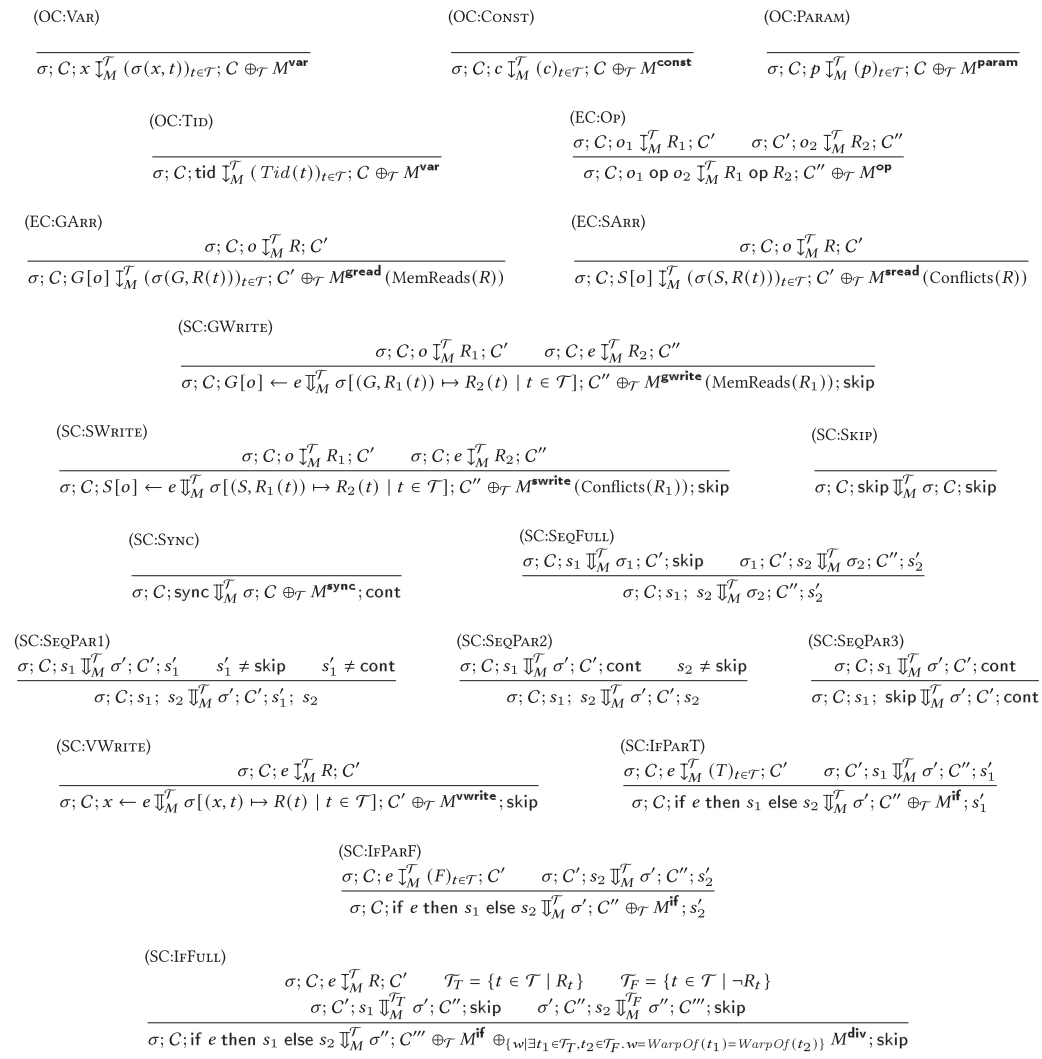

Fig. 10. Rules for thread-level parallelism between `syncthreads` (Part 1 of 2).

of the program that should run after the synchronization point, that is, the statement that should be executed to run the next segment.

Figures 10 and 11 give rules for the cost semantics. The rules for operands and expressions are similar to the lock-step semantics of Section 3. The judgment for statements is now

$$\sigma; C; s \Downarrow_M^{\mathcal{T}} \sigma'; C'; s',$$

meaning that, on a set of threads $\mathcal{T}$, with store σ, the statement s runs until completion or a synchronization point, resulting in final store σ' and continuation s'. If the cost collection is initially C, then the cost extended with the cost of executing s is C'. Rule SC:Sync immediately returns cont on evaluating a sync. The cont statement is a placeholder that will be propagated into outer statements to assist with building up the full continuation; the rules are constructed so that the final statement will not contain cont unless the entire statement being evaluated is sync (which we can

$$\text{(SC:WhileFull)}\quad \frac{\sigma; C; e \downarrow_M^{\mathcal{T}} R; C' \qquad \sigma; C'; s \Downarrow_M^{\mathcal{T}_T} \sigma'; C''; \mathsf{skip} \qquad \mathcal{T}_T = \{t \in \mathcal{T} \mid R_t\} \qquad \sigma'; C''; \mathsf{while}\ (e)\ s \Downarrow_M^{\mathcal{T}_T} \sigma''; C'''; \mathsf{skip}}{\sigma; C; \mathsf{while}\ (e)\ s \Downarrow_M^{\mathcal{T}} \sigma''; C''' \oplus_{\mathcal{T}} M^{\mathbf{if}} \oplus_{\{w \mid \exists t_1 \in \mathcal{T}_T, t_2 \in \mathcal{T} \setminus \mathcal{T}_T. w = WarpOf(t_1) = WarpOf(t_2)\}} M^{\mathbf{div}}; \mathsf{skip}}$$

$$\text{(SC:WhileNone)}\quad \frac{\sigma; C; e \downarrow_M^{\mathcal{T}} (F)_{t \in \mathcal{T}}; C'}{\sigma; C; \mathsf{while}\ (e)\ s \Downarrow_M^{\mathcal{T}} \sigma; C' \oplus_{\mathcal{T}} M^{\mathbf{if}}; \mathsf{skip}}$$

$$\text{(SC:WhilePar1)}\quad \frac{\sigma; C; e \downarrow_M^{\mathcal{T}} (T)_{t \in \mathcal{T}}; C' \qquad \sigma; C'; s \Downarrow_M^{\mathcal{T}} \sigma'; C''; s' \qquad s' \neq \mathsf{skip} \qquad s' \neq \mathsf{cont}}{\sigma; C; \mathsf{while}\ (e)\ s \Downarrow_M^{\mathcal{T}} \sigma'; C'' \oplus_{\mathcal{T}} M^{\mathbf{if}}; s';\ \mathsf{while}\ (e)\ s}$$

$$\text{(SC:WhilePar2)}\quad \frac{\sigma; C; e \downarrow_M^{\mathcal{T}} (T)_{t \in \mathcal{T}}; C' \qquad \sigma; C'; s \Downarrow_M^{\mathcal{T}} \sigma'; C''; \mathsf{skip} \qquad \sigma'; C''; \mathsf{while}\ (e)\ s \Downarrow_M^{\mathcal{T}} \sigma''; C'''; s'}{\sigma; C; \mathsf{while}\ (e)\ s \Downarrow_M^{\mathcal{T}} \sigma''; C''' \oplus_{\mathcal{T}} M^{\mathbf{if}}; s'}$$

$$\text{(SC:WhilePar3)}\quad \frac{\sigma; C; e \downarrow_M^{\mathcal{T}} (T)_{t \in \mathcal{T}}; C' \qquad \sigma; C'; s \Downarrow_M^{\mathcal{T}} \sigma'; C''; \mathsf{cont}}{\sigma; C; \mathsf{while}\ (e)\ s \Downarrow_M^{\mathcal{T}} \sigma'; C'' \oplus_{\mathcal{T}} M^{\mathbf{if}}; \mathsf{while}\ (e)\ s}$$

Fig. 11. Rules for thread-level parallelism between syncthreads (Part 2 of 2).

assume without loss of generality will not be the case, because a `__syncthreads` at the end of a kernel has little impact on performance). Conditionals, sequences, and loops have multiple rules depending on whether subexpressions evaluate fully or stop at a synchronization point. Rule SC:SeqFull handles the case where s_1 evaluates fully before reaching a sync; this is indicated by the fact that the continuation is skip. In this case, as in the lock-step rules, we proceed to evaluate s_2. If s_1 does not fully evaluate (i.e., its continuation is not skip), then we do not proceed to evaluate s_2 but return the continuation immediately. If the continuation of s_1 is exactly cont, then the continuation is simply s_2; we will start evaluating s_2 immediately upon resuming evaluation (this is handled by rule SC:SeqPar2). Otherwise, the cont may be nested inside other statements within the continuation s_1', which we should run upon resuming (rule SC:SeqPar1). In the special case where s_2 is skip, we ignore it and simply pass along the continuation.[7]

Rule SC:IfParT handles a conditional where e is true on all threads—it runs s_1 until it completes or reaches a sync. Rule SC:IfParF is similar for cases where the condition evaluates to false on all threads. Rule SC:IfFull handles cases where the condition is neither true for all threads or false for all threads—note that such a case does not necessarily indicate a divergent warp, as it is possible that all threads take the same branch *within each warp* (as in the `tricky` example considered previously). This rule requires both s_1 and s_2 to evaluate to skip, that is, neither may synchronize, as synchronizing within a branch of a conditional that is not taken by all threads within a block is an error in CUDA. Finally, the rule adds the cost M^{div} to only those warps for which the condition diverges, which we formalize as

$$\{w \mid \exists t_1 \in \mathcal{T}_T, t_2 \in \mathcal{T}_F. w = \mathit{WarpOf}(t_1) = \mathit{WarpOf}(t_2)\},$$

that is, warps containing both a thread t_1 where the condition is true and a thread t_2 where the condition is false.

Rule SC:WhileFull is similar to SC:IfFull, including the calculation of which warps to charge the cost of a divergence (the rule does not define $\mathcal{T}_F$, so the set of threads for which the condition is false is written $\mathcal{T} \setminus \mathcal{T}_T$). Synchronizing within a loop where not all threads are active

[7] In a big-step semantics, it would be harmless to return a skip that will be removed when computation resumes. However, our results of the next section require statements to be formed such that they do not begin with skip or cont unless the entire statement is just skip or cont. Maintaining this invariant in the cost semantics will become useful in proofs.

$$\text{(SF:Full)}\ \frac{\sigma; C; s \Downarrow_M^{\mathcal{T}} \sigma'; C'; \mathsf{skip}}{\sigma; C; s \Downarrow_M^{\mathcal{T}} \sigma'; C'} \qquad \text{(SF:Partial)}\ \frac{\sigma; (W,S); s \Downarrow_M^{\mathcal{T}} \sigma'; C'; s' \quad s' \neq \mathsf{skip} \quad \sigma'; (W, (\max S)_{Warps}); s' \Downarrow_M^{\mathcal{T}} \sigma''; C''}{\sigma; (W,S); s \Downarrow_M^{\mathcal{T}} \sigma''; C''}$$

Fig. 12. Rules for full evaluation with thread-level parallelism.

is also disallowed. Rule SC:WhileNone handles the case where the loop condition is false, and rules SC:WhilePar1, SC:WhilePar2 and SC:WhilePar3 handle the case where the condition is true for all threads. In SC:WhilePar1, the loop body synchronizes in the middle so the continuation is propagated. The continuation is s'; while (e) s indicating that the current loop body should finish and then the loop should continue. In SC:WhilePar2, one iteration of the loop body terminates successfully, so the loop is continued until it terminates or synchronizes. In SC:WhilePar3, the loop body synchronizes at the end; in this case, rather than set the continuation to be cont; while (e) s we (as in SC:SeqPar2) simply continue with the next statement, which is the next iteration of the while loop.

Figure 12 gives two rules that string these partial evaluations together into a full evaluation. Rule SF:Full applies if the statement evaluates fully without encountering a sync; the continuation is simply discarded. Rule SF:Partial runs the statement until the warps synchronize, and then takes the maximum span over the warps before proceeding recursively with evaluation on the continuation.

Example 4. As an example, Figure 13 gives a derivation using the thread-level semantics for evaluating the statement s_1; sync; s_2, where

$$\begin{aligned} s_1 &\triangleq \text{if tid} < 32 \text{ then } G[\text{tid}] \leftarrow 42 \text{ else skip}, \\ s_2 &\triangleq \text{if tid} \geq 32 \text{ then } G[\text{tid}] \leftarrow 42 \text{ else skip}. \end{aligned}$$

This statement translates the kernel `tricky` above into miniCUDA syntax. We will associate the sequencing operator to the left, so we first evaluate s_1; sync. The first derivation, D_1, evaluates s_1 to update the store to σ' (the exact definition of σ' and the subderivation are similar to the example in Figure 6 and are omitted). The derivation uses the rule SC:IfFull, because the condition tid < 32 evaluates to True in some threads and False in others. However, there are no warps that actually diverge and so the resulting cost graph is $C \oplus_{\mathcal{T}} M^{\mathsf{if}} \oplus_{\emptyset} M^{\mathsf{div}} = C \oplus_{\mathcal{T}} M^{\mathsf{if}}$. This statement evaluates fully to skip, so the sequence s_1; sync can evaluate using rule SC:SeqFull, which handles the case where s_1 evaluates fully and we simply evaluate the remainder of the sequence, which in this case is sync. This evaluates to cont. The outer sequence expression s_1; sync; s_2 can therefore only evaluate using SC:SeqPar2, which "catches" the cont returned by SC:SeqFull and returns the continuation of the sequence, which will be executed after the synchronization, which is s_2.

The next derivation, D_2, picks up the execution of the program (with the new store and cost graph) after the synchronization and executes s_2 using SC:IfFull, which proceeds as above. Because this evaluation ends in skip, we may lift this derivation to a full evaluation derivation (the judgment of Figure 12) using Rule SF:Full. Finally, we can compose the two derivations to get a full evaluation of the program starting at the beginning. Rule SF:Partial composes these derivations: D_1 computes up to the first sync and D_2 finishes the execution. If there were multiple instances of sync in the program, then D_2 would in turn have nested uses of SF:Partial with subderivations that execute the remaining components of the program.

Coq Mechanization. We represent a warp as a natural number and a family of costs C as a map from warp numbers to pairs `Cost * Cost` representing the work and span of that warp. We also define a new record type `wsmetric` for the paired resource metrics used in this section, which

$$D_1 \quad \dfrac{\dfrac{\dfrac{\dfrac{\dots}{\sigma; C; s_1 \Downarrow^{\mathcal{T}}_M \sigma'; C \oplus_{\mathcal{T}} M^{\mathbf{if}}; \mathsf{skip}} \quad \dfrac{}{\sigma; C \oplus_{\mathcal{T}} M^{\mathbf{if}}; \mathsf{sync} \Downarrow^{\mathcal{T}}_M \sigma'; C \oplus_{\mathcal{T}} M^{\mathbf{if}} \oplus_{\mathcal{T}} M^{\mathbf{sync}}; \mathsf{cont}}\,(\text{SC:Sync})}{\sigma; C; s_1; \mathsf{sync} \Downarrow^{\mathcal{T}}_M \sigma'; C \oplus_{\mathcal{T}} M^{\mathbf{if}} \oplus_{\mathcal{T}} M^{\mathbf{sync}}; \mathsf{cont}}\,(\text{SC:SeqFull})}{\sigma; C; s_1; \mathsf{sync}; s_2 \Downarrow^{\mathcal{T}}_M \sigma'; C \oplus_{\mathcal{T}} M^{\mathbf{if}} \oplus_{\mathcal{T}} M^{\mathbf{sync}}; s_2}\,(\text{SC:SeqPar2})$$

$$D_2 \quad \dfrac{\dfrac{\dfrac{\dots}{\sigma'; C \oplus_{\mathcal{T}} M^{\mathbf{if}} \oplus_{\mathcal{T}} M^{\mathbf{sync}}; s_2 \Downarrow^{\mathcal{T}}_M \sigma''; C \oplus_{\mathcal{T}} M^{\mathbf{if}} \oplus_{\mathcal{T}} M^{\mathbf{sync}} \oplus_{\mathcal{T}} M^{\mathbf{if}}; \mathsf{skip}}\,(\text{SC:IfFull})}{\sigma'; C \oplus_{\mathcal{T}} M^{\mathbf{if}} \oplus_{\mathcal{T}} M^{\mathbf{sync}}; s_2 \Downarrow^{\mathcal{T}}_M \sigma''; C \oplus_{\mathcal{T}} M^{\mathbf{if}} \oplus_{\mathcal{T}} M^{\mathbf{sync}} \oplus_{\mathcal{T}} M^{\mathbf{if}}}\,(\text{SF:Full})$$

$$D \quad \dfrac{D_1 \qquad D_2}{\sigma; C; s_1; \mathsf{sync}; s_2 \Downarrow^{\mathcal{T}}_M \sigma''; C \oplus_{\mathcal{T}} M^{\mathbf{if}} \oplus_{\mathcal{T}} M^{\mathbf{sync}} \oplus_{\mathcal{T}} M^{\mathbf{if}}}\,(\text{SF:Partial})$$

Fig. 13. Example derivation using the thread-level parallel cost semantics.

gives both the work and span. The definition is similar to the definition of `resmetric` except with pairs of costs.

We encode the thread-level cost semantics of Figures 10 and 11 in a similar fashion to the lockstep cost semantics, as three inductive definitions `par_eval_opd`, `par_eval_exp`, and `partial_eval_stmt`. The last of these is "partial," because it evaluates only up to sync. The definition `par_eval_stmt` encodes the "full evaluation" rules of Figure 12.

Soundness. We now show that the analysis of Section 4, when applied on work and span independently, soundly approximates the parallel cost semantics of Figures 10 and 11. We do this in two stages: first, we show that the costs derived by the parallel semantics are overapproximated by the costs derived by the lock-step cost semantics of Section 3 (extended with a rule treating `__syncthreads()` as a no-op). Second, we apply the soundness of those cost semantics. The lock-step cost semantics were designed to model only single-warp execution, and so it may seem odd to model an entire block using them. However, doing so results in a sound overapproximation: for example, in the kernel shown above, the lock-step cost semantics ignores the synchronization but assumes that the two branches must be executed in sequence anyway, because not all threads take the same branch. As we now prove, these two features of the warp-level cost semantics cancel out. Lemma 4 states that if evaluating an expression e with initial cost collection C results in a cost collection C', then e can evaluate using M_w and M_s, under the lock-step semantics, with costs C_w and C_s, respectively. The difference in span between C and C' is overapproximated by C_s and the difference in work is overapproximated by C_w times the number of warps. Lemma 5 is the equivalent result for statements. Both proofs are straightforward inductions and are formalized in Coq.

Lemma 4. *If* $\Sigma \vdash e : \tau$ *and* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and* $\sigma; C; e \downarrow^{\mathcal{T}}_M R; C'$, *then there exist* C_w *and* C_s *such that*

(1) $\sigma; e \downarrow^{\mathcal{T}}_{M_w} R; C_w$
(2) $\sigma; e \downarrow^{\mathcal{T}}_{M_s} R; C_s$
(3) $W(C') - W(C) \le C_w \,|\{\mathit{WarpOf}(t) \mid t \in \mathcal{T}\}|$
(4) $S(C') - S(C) \le C_s$

Lemma 5. *If* $\Sigma \vdash s$ *and* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and* $\sigma; C; s \Downarrow^{\mathcal{T}}_M \sigma'; C'$, *then there exist* C_w *and* C_s *such that*

(1) $\sigma; s \Downarrow^{\mathcal{T}}_{M_w} \sigma'; C_w$
(2) $\sigma; s \Downarrow^{\mathcal{T}}_{M_s} \sigma'; C_s$

(3) $W(C') - W(C) \leq C_w \,|\{\mathit{WarpOf}(t) \mid t \in \mathcal{T}\}|$
(4) $S(C') - S(C) \leq C_s$

We now apply Theorem 1 to show that the analysis of Section 4, when run independently using the projections of the resource metric M, soundly approximates the work and span of a block.

Theorem 3. *If* $\Sigma \vdash s$ *and* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and*

$$\vdash_M \{P; Q_w; X_w\}\, s\, \{P'; Q'_w; X'_w\} \quad \text{and} \quad \vdash_M \{P; Q_s; X_s\}\, s\, \{P'; Q'_s; X'_s\}$$

and $\sigma; \mathcal{T} \vDash \{P; Q_w; X_w\}$ *and* $\sigma; \mathcal{T} \vDash \{P; Q_s; X_s\}$ *and* $\sigma; \emptyset; s \Downarrow^{\mathcal{T}}_M \sigma'; C$, *then*

$$W(C) \leq (Q_{w\,X_w}(\sigma) - Q'_{w\,X'_w}(\sigma'))\,|\{\mathit{WarpOf}(t) \mid t \in \mathcal{T}\}|$$

and

$$S(C) \leq Q_{s\,X_s}(\sigma) - Q'_{s\,X'_s}(\sigma').$$

Proof. The proof is formalized in Coq, but the reasoning is fairly simple so we outline it here as well. Note that $W(\emptyset) = S(\emptyset) = 0$, so by Lemma 5, we have C_w and C_s such that

(1) $\sigma; s \Downarrow^{\mathcal{T}}_{M_w} \sigma'; C_w$
(2) $\sigma; s \Downarrow^{\mathcal{T}}_{M_s} \sigma'; C_s$
(3) $W(C) \leq C_w \,|\{\mathit{WarpOf}(t) \mid t \in \mathcal{T}\}|$
(4) $S(C) \leq C_s$

and by Theorem 1, we have $Q_{w\,X_w}(\sigma) - C_w \geq Q'_{w\,X'_w}(\sigma') \geq 0$ and $Q_{s\,X_s}(\sigma) - C_s \geq Q'_{s\,X'_s}(\sigma') \geq 0$. The result follows. □

Summary. We now briefly summarize at a high level the contributions of Sections 3–5. Section 3 formally defined the miniCUDA calculus, which we use for our theoretical results. In particular, this section defined the "lock-step cost semantics," which gives rules for evaluating a miniCUDA kernel while tracking the cost of the execution. We refer to this as the lock-step cost semantics, because it assumes a set of threads $\mathcal{T}$ all running in lock-step, that is, one warp of a GPU.

Section 4 defined a *quantitative program logic* for bounding the execution cost of a miniCUDA kernel *without* running it or simulating its execution (as is done by the cost semantics). The key result of this section is the soundness of the quantitative program logic, that is, that the cost predicted by the logic for a given kernel is at least the cost of actually executing the kernel, as tracked by the cost semantics.

Bounding the execution cost of one warp, as is done in Sections 3 and 4, does not, however, give any indication of how to *compose* these costs when more than one warp is active. Suppose a computation is spread across 10 warps and each takes 1ms to run. It should certainly not be the case that the entire computation takes 10 ms to run. Nor, if the GPU can execute 5 warps simultaneously, is it likely to be the case that the computation takes exactly 2 ms to run—the GPU will most likely interleave the computation of more than 5 warps to hide the latency of operations such as memory accesses. Prior results in parallel scheduling theory bound the execution time in such cases by the number of processors (in this case, the number of warps that can be executed simultaneously) and two quantitative properties of the program known as the *work* and *span*. Section 5 shows how the quantitative program logic of Section 4 can be used to bound the work and span of a *block* of threads. The section shows that this is sound even though the logic was designed for only one warp by defining a "parallel" (as opposed to lock-step) cost semantics that more faithfully models the parallel execution of a block, and showing that it can be simulated by the lock-step cost semantics.

6 BOUNDED IMPLEMENTATION

Section 5 presents a technique for conservatively approximating the work and span of a CUDA kernel, and validates it against our cost semantics, now extended to track work and span. The goal of this endeavor was to place a bound on execution time using Theorem 2. However, it remains to be shown that the work W and span S assigned to a program by our high-level, big-step cost semantics are accurate and, in particular, that a program can actually be executed in time $\frac{W}{P} + S$ on P threads of a real GPU. In this section, we provide evidence of this assertion in the form of a *bounded implementation* [Blelloch and Greiner 1995, 1996]. Broadly speaking, we use the term bounded implementation to mean a faithful, low-level model of the execution of a parallel program on a parallel machine that tracks the number of operations (or "clock cycles") of the machine. It is used to validate a higher-level cost semantics like those of Sections 3 and 5 in that we show that the cost given by the high-level model approximates the cost given by the bounded implementation (which is constructed to be more accurate operationally, but is often more unwieldy to work with formally).

Our bounded implementation is a lower-level, small-step model of GPU execution where we measure time by the number of steps in an evaluation. In addition to serving to validate the parallel cost semantics, this small-step operational semantics for miniCUDA allows us to prove progress and preservation for the type system of Section 3. We then establish the desired correspondence between the small-step semantics and the cost semantics of Section 5.

The small-step semantics we present steps a program until it reaches a synchronization point sync, similar to the partial evaluation cost semantics of Section 5 (in fact, we prove a correspondence between these two judgments). However, we only use this judgment to model the lock-step execution of one warp and we define another judgment to model the execution of many warps in parallel. As before, our small-step semantics is parameterized by a resource metric M. The resource metric now provides the latency of each operation; because we are counting work in steps of the semantics, the work of every operation is 1.[8] The unit of latency is now assumed to be execution steps; that is, an operation with latency 0 takes one step to execute and the next instruction can execute immediately. An operation with latency 1 takes one step to execute but execution must then pause for one step before continuing, and so on. The small-step rules are given in Figures 14 and 15 and are grouped into rules for operands, expressions, and statements. The judgment for operands and expressions is $e \underset{M;\sigma;\mathcal{T}}{\longmapsto^{C}} R$. As with the cost semantics, M is the resource metric, σ is the store, $\mathcal{T}$ is the set of threads, and R is a thread-indexed family of result values. The cost C is the latency of the operation, which will correspond to how many steps execution must pause before executing the next statement. Most rules step an operand or expression to a result in one step. Exceptions are ES:Op1, which evaluates just the first operand (the second will be evaluated on the next step) and ES:GArr1 and ES:SArr1, which evaluate the array index to a result.

Before introducing the small-step rules for statements, we add two additional syntactic forms that are not part of the user-level syntax. The statements $\mathsf{switch}(\mathcal{T}', s)$ and $\mathsf{switch}_\diamond(\mathcal{T}', s)$ execute s at $\mathcal{T}'$ (which is assumed to be a subset of the current set of threads). These statements are used to implement the thread-masking behavior of conditionals when warps diverge. For example, when a condition is true on threads $\mathcal{T}_T$ and false on $\mathcal{T}_F$, a conditional will step to

$$\mathsf{switch}(\mathcal{T}_T, s_1);\ \mathsf{switch}_\diamond(\mathcal{T}_F, s_2).$$

The difference between $\mathsf{switch}(\mathcal{T}', s)$ and $\mathsf{switch}_\diamond(\mathcal{T}', s)$ is that the latter consumes an additional step of evaluation to model the cost of masking, whereas the former changes the set of active threads without any additional evaluation steps. Thus, the statement above consumes

[8] We could model some operations taking more than one cycle of processing time by splitting the operation across multiple steps of the small-step semantics.

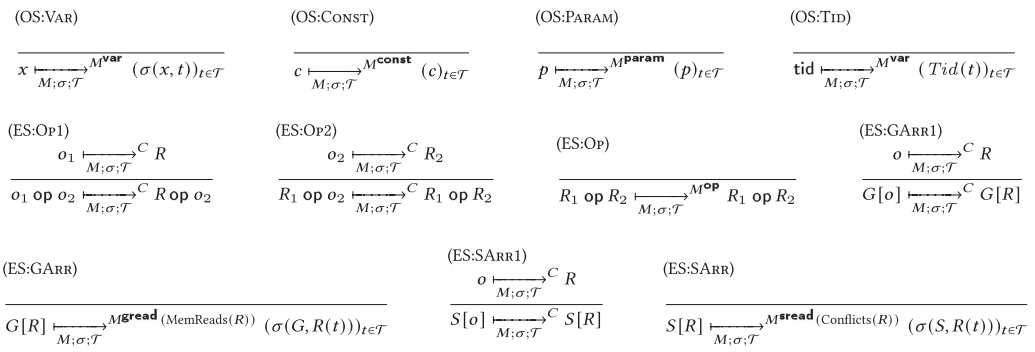

Fig. 14. Small-step rules for a single warp (Part 1 of 2).

one additional evaluation step to model the overhead of changing the thread mask—if we had only $\mathsf{switch}_\diamond(\mathcal{T}', s)$, this overhead would consume two steps. This allows us to model the work of a divergent warp as being the same as the work of every other operation—we could of course choose to model divergent warps as taking a different amount of time by adjusting the use of the two switch statements.

The small-step rules for statements, in Figure 15, operate on *configurations* $\langle \sigma, s \rangle$ consisting of a store and a statement. The judgment $\langle \sigma, s \rangle \underset{M;\mathcal{T}}{\longmapsto}^{C} \langle \sigma', s' \rangle$ states that under store σ and threads $\mathcal{T}$, the statement s steps to s' and the store becomes σ'. The latency of the step is C, which, as before, is the number of steps the execution will have to pause after this step. There are 3 rules for conditionals if e then s_1 else s_2: first, SS:If1 steps e until it is a result. If e is true on all threads, then the conditional steps to s_1 (rule SS:IfT), and similar if it is false on all threads (rule SS:IfF). If the warp diverges, then the set of threads is split and the conditional steps to two switch statements, which will execute in sequence as described above. Sequences $s_1; s_2$ step s_1 until it becomes a trivial statement, at which point rule SS:SeqSkip immediately steps the sequence to s_2. The auxiliary judgment $\mathsf{skip}(s)$ (read "s is trivial") is true if s is skip, is a sequence of trivial statements or is $\mathsf{switch}(\mathcal{T}', s')$ where s' is trivial (this last case, together with SS:SeqSkip, allows $\mathsf{switch}(\mathcal{T}', s')$ to immediately continue on to the next statement when s' is trivial without costing another step). Because this judgment is syntactic and easily decidable, we also use it in the negated form $\neg\mathsf{skip}(s)$. Note that there are no rules for sync or cont. When evaluation reaches a sync, this warp will stop executing. We use another judgment, defined later, to step past the synchronization point and resume execution.

Coq mechanization of small-step semantics. The small-step operational semantics is encoded as three inductive definitions `step_opd`, `step_exp`, and `step_stmt`. We also formalize the judgment $\mathsf{skip}(s)$ as `is_skip`. To formalize the preservation and progress results (described below) we also define a series of judgments (on operands, expressions, and statements) referred to as "valid," which are similar to the "typed" judgments but allow for well-typed result families to be included as operands and expressions. Two additional inductive definitions, `validwithswitch` and `validwithcont` allow statements to include switch and cont, which arise as intermediate statements during evaluation and so need to be considered for the proofs of type safety but are not included in the standard typing judgments.

We require an additional property of statements that is guaranteed by the small-step semantics but has not been formalized up to now, which is essentially that skip and cont do not appear at the beginning of statements (because they would have been removed by rules such as SS:SeqSkip). We formalize this as the inductive definition `stmt_wf` and prove that it is preserved by the semantics.

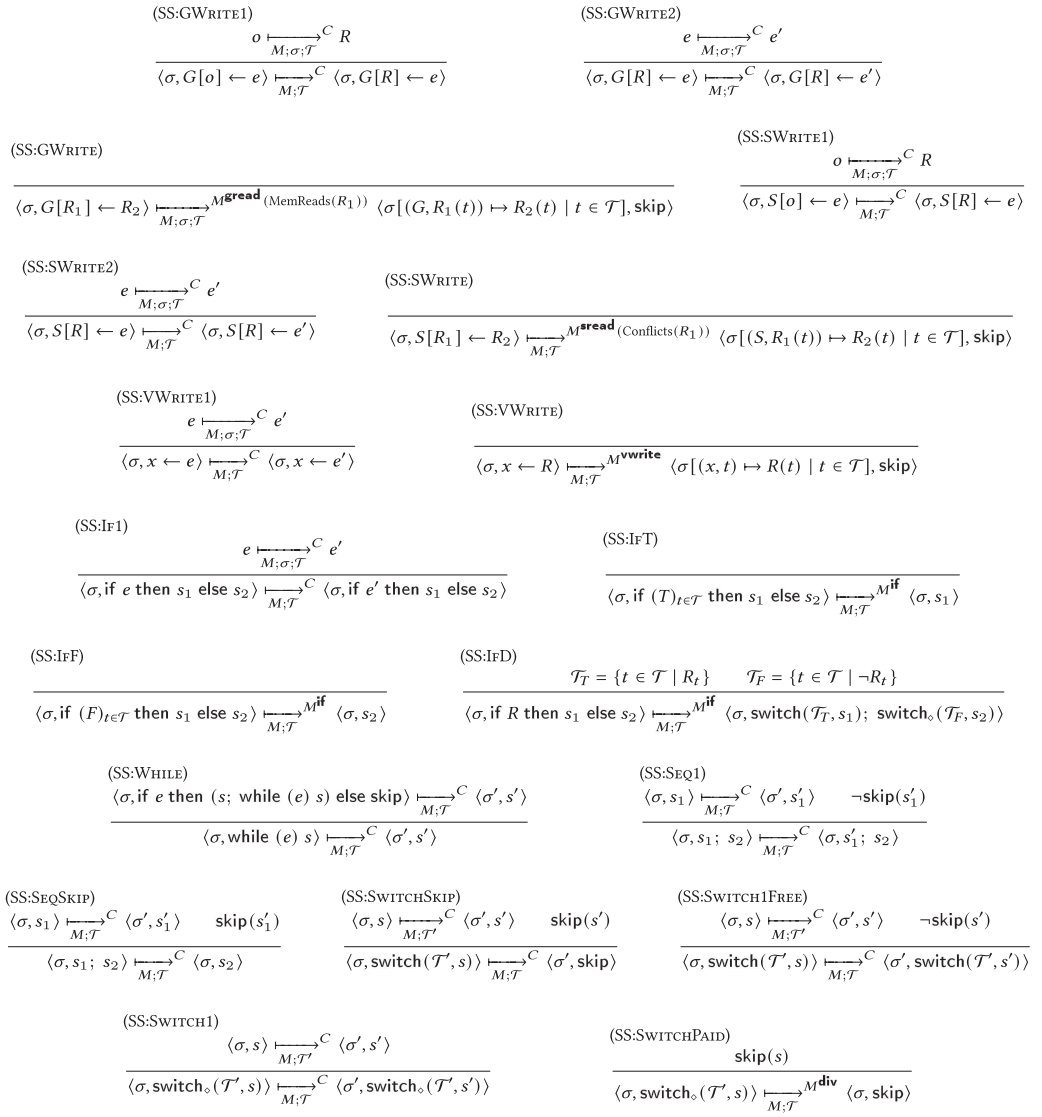

Fig. 15. Small-step rules for a single warp (Part 2 of 2).

We define a new record `natmetric` for resource metrics suitable for the small-step semantics (i.e., those that return a nonnegative integer latency for each resource constnat). The function `metric_of_nat` converts this to a standard resource metric and the function `ws` pairs two resource metrics (one that gives the work, for which we use the constant resource metric that returns 1 for each operation, and one for the latency) to return a `resmetric`.

Type Safety. We are now able to state and prove a standard preservation result stating that steps in the semantics preserve typing (recall that statements do not have types, but a well-formed statement steps to a well-formed statement). A result family is considered to be an expression with type τ, that is, $\Sigma \vdash R : \tau$, if for all $t \in \mathcal{T}$, we have $\Sigma \vdash R(t) : \tau$.

Theorem 4 (Preservation). (1) *If* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and* $\Sigma \vdash e : \tau$ *and* $e \xrightarrow[M;\sigma;\mathcal{T}]{}^{C} e'$, *then* $\Sigma \vdash e' : \tau$.

(2) *If* $\Sigma \vdash_{\mathcal{T}} \sigma$ *and* $\Sigma \vdash s$ *and* $\langle \sigma, s\rangle \xrightarrow[M;\mathcal{T}]{}^{C} \langle \sigma', s\rangle$, *then* $\Sigma \vdash_{\mathcal{T}} \sigma'$ *and* $\Sigma \vdash s'$.

Proof. By induction on the respective step judgments. The proof is formalized in Coq. □

Recall that store updates $\sigma[(A, R(t)) \mapsto R'(t) \mid t \in \mathcal{T}]$ are undefined in the case where $R(t) = R(t')$ for two distinct threads, which is a particular instance of a write-write data race (other types of data races are possible but are out of the scope of this work). Therefore, such a data race is a "stuck state": there is no well-defined state for such an assignment to step to. Because this violates progress, data races must be formalized and accounted for in the progress result. We say that the statement $A[R_1] \leftarrow R_2$ has a data race if $R_1(t) = R_1(t')$ where $t, t' \in \mathcal{T}$ and $t \neq t'$. We also say that $s; s_2$ and $\text{switch}_{\diamond}(\mathcal{T}', s)$ and $\text{switch}(\mathcal{T}', s)$ have data races if s has a data race.

Progress can also fail (at least with the current rules) when a statement reaches a sync. We say a statement s is blocked on a sync if $s = \text{sync}$ or s is $s_1; s_2$ or $\text{switch}_{\diamond}(\mathcal{T}', s_1)$ or $\text{switch}(\mathcal{T}', s_1)$ and s_1 is blocked on a sync. This is formalized in Coq as `is_sync`.

The progress theorem then states that a well-typed expression or operand (that is not already a result) can take a step, and that a well-typed statement is trivial (skip(s)), has a data race, is blocked on a sync, or can take a step.

Theorem 5 (Progress). *If* $\Sigma \vdash_{\mathcal{T}} \sigma$, *then:*

(1) *If* $\Sigma \vdash e : \tau$, *then* e *is a result or there exists* e' *such that* $e \xrightarrow[M;\sigma;\mathcal{T}]{}^{C} e'$

(2) *If* $\Sigma \vdash s$ *and* s *is not* cont, *then*
   (a) skip(s) *or*
   (b) s *has a data race or*
   (c) s *is blocked on a* sync *or*
   (d) *there exist* σ' *and* s' *such that* $\langle \sigma, s\rangle \xrightarrow[M;\mathcal{T}]{}^{C} \langle \sigma', s'\rangle$.

Proof. By induction on the respective typing judgments. The proof is formalized in Coq. □

Soundness with respect to partial evaluation. Lemma 6 establishes the connection between the small-step evaluation judgment and the partial evaluation cost semantics of Section 5. In particular, we show that if an expression (respectively, configuration) takes a step with latency C and the resulting expression (respectively, configuration) has costs C', then the original expression (respectively, configuration) has one additional unit of work and C additional units of span. In other words, taking a step decreases the work and span by the expected amounts. Because the work and span cannot drop below zero, this is a key property in bounding the number of steps that can be taken by a small-step evaluation, which is our goal in this section.

Recall that the small-step semantics uses a resource metric that returns only a latency (the work of each operation is assumed to be 1), while the cost semantics uses a resource metric that returns both a work and a latency. We therefore introduce the operation $WS(-)$ on resource metrics, where for any resource constant rc,

$$(WS(M))^{\text{rc}} = (1, M^{\text{rc}}).$$

Lemma 6. *Let* $\mathcal{T}$ *be a set of threads such that for all* $t, t' \in \mathcal{T}$, $WarpOf(t) = WarpOf(t')$. *Then*

— *If* $e \xrightarrow[M;\sigma;\mathcal{T}]{}^{C} e'$ *and* $\sigma; \emptyset; e' \downarrow^{\mathcal{T}}_{WS(M)} R; C'$, *then* $\sigma; \emptyset; e \downarrow^{\mathcal{T}}_{WS(M)} R; C' \oplus_{\mathcal{T}} (1, C)$.

— *If* $\langle \sigma, s\rangle \xrightarrow[M;\mathcal{T}]{}^{C} \langle \sigma', s'\rangle$ *and* $\sigma'; \emptyset; s' \Downarrow^{\mathcal{T}}_{WS(M)} \sigma''; s''; C'$, *then* $\sigma; \emptyset; s \Downarrow^{\mathcal{T}}_{WS(M)} \sigma''; s''; C' \oplus_{\mathcal{T}} (1, C)$.

Proof. By induction on the derivation of the respective step judgments. We prove two representative cases for statements. The full proof is formalized in Coq.

$$\frac{s_1 \to s_1' \qquad s_1' \neq \mathsf{cont}}{s_1;\ s_2 \to s_1';\ s_2} \qquad \frac{s_1 \to \mathsf{cont} \qquad s_2 \neq \mathsf{skip}}{s_1;\ s_2 \to s_2} \qquad \frac{s_1 \to \mathsf{cont}}{s_1;\ \mathsf{skip} \to \mathsf{cont}} \qquad \frac{}{\mathsf{sync} \to \mathsf{cont}}$$

Fig. 16. Small-step rules for syncthreads.

— SS:VWrite1. Then $s = x \leftarrow e$ and $s' = x \leftarrow e'$ and $e \xrightarrow[M;\sigma;\mathcal{T}]{}^{C} e'$ and $\sigma'; \emptyset; s' \Downarrow^{\mathcal{T}}_{WS(M)} \sigma''; s''; C'$. By inversion on SC:VWrite, $\sigma; \emptyset; e' \downarrow^{\mathcal{T}}_{WS(M)} R; C''$ and $C' = C'' \oplus_{\mathcal{T}} (1, M^{\mathsf{vwrite}})$. By induction, $\sigma; \emptyset; e \downarrow^{\mathcal{T}}_{WS(M)} R; C'' \oplus_{\mathcal{T}} (1, C)$. By SC:VWrite, $\sigma; \emptyset; s \Downarrow^{\mathcal{T}}_{WS(M)} \sigma''; s''; C'' \oplus_{\mathcal{T}} (1, C) \oplus_{\mathcal{T}} (1, M^{\mathsf{vwrite}})$.
— SS:VWrite. Then $s = x \leftarrow R$ and $s' = \mathsf{skip}$ and $\sigma' = \sigma[(x,t) \mapsto R(t) \mid t \in \mathcal{T}]$ and $C = M^{\mathsf{vwrite}}$ and $\sigma'; \emptyset; s' \Downarrow^{\mathcal{T}}_{WS(M)} \sigma''; s''; C'$. By inversion on SC:Skip, we have $C' = \emptyset$ and $\sigma'' = \sigma'$. By SC:VWrite, $\sigma; \emptyset; s \Downarrow^{\mathcal{T}}_{WS(M)} \sigma''; \emptyset \oplus_{\mathcal{T}} (1, M^{\mathsf{vwrite}})$;. □

Figure 16 defines the judgment $s \to s'$, which takes a statement that is blocked on a sync and steps it to its continuation so that s' is ready to execute after the synchronization is completed. As in the cost semantics, the sync is converted to cont, which is propagated through sequences. Note again that s' does not contain cont unless $s = \mathsf{sync}$. Lemma 7 shows that a step with this judgment results in the same final statement as evaluation with the cost semantics and that the latency in the cost semantics is 1 (just the work of the synchronization). This judgment is formalized in Coq as the inductive definition sync_step.

Lemma 7. *If $s \to s'$, then $\sigma; \emptyset; s \Downarrow^{\mathcal{T}}_{WS(M)} \sigma; s'; C$, where $S(C) = 1$.*

Proof. By induction on the derivation of $s \to s'$. All cases are straightforward and are formalized in Coq. □

We extend the progress and preservation results to include this new judgment. Combined with the existing progress and preservation lemmas, this means that a well-typed program can step to completion, alternating between normal steps and synchronizations. The preservation result says that, unless the entire statement is sync (and therefore steps to cont), the resulting statement is well-formed and therefore progress applies to s'.

Lemma 8 (Preservation with Synchronizations). *If $\Sigma \vdash s$ and $s \to s'$, then $\Sigma \vdash s'$ or $s' = \mathsf{cont}$.*

Proof. By induction on the derivation of $s \to s'$. All cases are straightforward and are formalized in Coq. □

The progress result allows for a stuck state in which a statement is blocked on a sync inside a switch (which indicates an invalid synchronization when some threads are inactive and is formalized in Coq as bad_sync), but shows that there are no other stuck states.

Lemma 9 (Progress with Synchronizations). *If s is blocked on a* sync*, then either the* sync *is nested inside a* switch *or there exists s' such that $s \to s'$.*

Proof. The proof is formalized in Coq and considers the few cases in which s can be blocked on a sync, which we sketch below.

— $s = \mathsf{sync}$. Then $s \to \mathsf{cont}$.
— $s = s_1; s_2$ and s_1 is blocked. By induction, either the sync is nested inside a switch or $s_1 \to s_1'$. In the latter case, $s \to s'$ by one of the first three rules of Figure 16.
— $s = \mathsf{switch}_\diamond(\mathcal{T}', s_0)$ or $s = \mathsf{switch}(\mathcal{T}', s_0)$ and s_0 is blocked on a sync. In this case, there is a sync when some threads are inactive. □

Block-level execution. Thus far, we have focused on modeling the single-step execution of threads in one warp. We now turn our attention to modeling the execution of an entire block. We represent one warp within a block as a triple $(\delta, s, \mathcal{T})$, where s is the statement being executed by the warp and $\mathcal{T}$ is the complete set of threads in the warp (not just the currently active threads; this masking will be accomplished by switch statements within s). The first component, δ, is a *delay* that tracks the number of steps of latency that must pass before this warp is ready to execute again. A delay of 0 indicates that the warp is ready to execute. If it then executes an operation with a latency of C, then its delay becomes C. We represent a block B as a set of warps; although the thread identifiers provide a clear ordering on warps, it will be convenient for us to treat this set as unordered.

We use the judgment $\langle B, \sigma\rangle \Rrightarrow_{M;P} \langle B', \sigma'\rangle$ to indicate that B steps in parallel on P processors to B', changing the store from σ to σ'. There are two rules for this judgment, shown in Figure 17. Rule BL:Step steps as many warps as possible, utilizing thread-level parallelism. The auxiliary judgment $B \xrightarrow{P-Greedy} (E, U, D)$ splits B into a set E of Executed warps, a set U of Unexecuted warps, and a set D of Delayed warps, where:

— For all $(\delta, s, \mathcal{T}) \in E \cup U$, we have $\delta = 0$.
— For all $(\delta, s, \mathcal{T}) \in D$, we have $\delta > 0$.
— $|E| \leq P$ and if $|E| < P$, then all threads in U are blocked on a sync.

This models a *greedy scheduler* [Eager et al. 1989], which chooses as many ready threads as possible for execution (where a thread is not ready if it is delayed or blocked on a synchronization). The threads in E are stepped, and the delays of the threads in D are decreased by one. The rule threads the store through the steps of all of the threads in E, as if the steps are executed sequentially, but one can also think of this as executing all of the steps in parallel and, in the case of data races, prioritizing writes by warps "earlier" in the selected (arbitrary) ordering. This rule can apply until all threads are blocked on a synchronization, at which point we wish to resume all threads. This is accomplished by Rule BL:Sync, which steps all warps using the $s \to s'$ judgment.

For example, consider the block

$$B \triangleq \{(0, x \leftarrow 0; \mathsf{sync}, \mathcal{T}_1), (0, \mathsf{sync}, \mathcal{T}_2), (1, \mathsf{sync}, \mathcal{T}_3)\},$$

where $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ are the sets of threads in three respective warps. Then, we have

$$B \xrightarrow{P-Greedy} (\{(0, x \leftarrow 0; \mathsf{sync}, \mathcal{T}_1)\}, \{(0, \mathsf{sync}, \mathcal{T}_2)\}, \{(1, \mathsf{sync}, \mathcal{T}_3)\}).$$

Here, the first warp may execute, the second warp is blocked on a sync, and the third warp is delayed, so this is the only possible greedy split of the block.

By rule BL:Step, $B \Rrightarrow_{M;P} B'$, where

$$B' = \{(M^{\mathsf{vwrite}}, \mathsf{sync}, \mathcal{T}_1), (0, \mathsf{sync}, \mathcal{T}_2), (0, \mathsf{sync}, \mathcal{T}_3)\}.$$

The executed warp has stepped, leaving it delayed by the latency of a variable write, and the previously delayed warp has had its delay decreased. On the next step, the second and third warps will be in U, because they are blocked and not delayed; the first warp is delayed (assuming $M^{\mathsf{vwrite}} > 0$). Rule BL:Step will decrement the delay until it reaches 0 at which point rule BL:Sync will apply and step the entire block to

$$\{(0, \mathsf{cont}, \mathcal{T}_1), (0, \mathsf{cont}, \mathcal{T}_1), (0, \mathsf{cont}, \mathcal{T}_1)\}.$$

To prove a correspondence between the bounded implementation and the cost semantics, we also need to define a version of the cost semantics over blocks. In the previous section, we used the cost semantics to reason about an entire block, but this assumes that all warps are executing the same statement at once. In the context of a big-step semantics, this assumption is not particularly troublesome, as these judgments evaluate the entire block of code between synchronizations at

$$\text{(BL:Step)}\quad \frac{B = \{(\delta_1, s_1, \mathcal{T}_1), \ldots, (\delta_n, s_n, \mathcal{T}_n)\} \qquad B \xrightarrow{P\text{–}Greedy} (\{e_1, \ldots, e_i\}, \{u_1, \ldots, u_j\}, \{d_1, \ldots, d_k\}) \qquad \langle s_{e_1}, \sigma\rangle \xrightarrow[M;\mathcal{T}_{e_1}]{}^{C_{e_1}} \langle s'_{e_1}, \sigma'_1\rangle \ldots \langle s_{e_i}, \sigma'_{e_{i-1}}\rangle \xrightarrow[M;\mathcal{T}_{e_i}]{}^{C_{e_i}} \langle s'_{e_i}, \sigma'\rangle}{\langle B, \sigma\rangle \Rrightarrow_{M;P} \langle \{(C_{e_1}, s'_{e_1}, \mathcal{T}_{e_1}), \ldots, (C_{e_i}, s'_{e_i}, \mathcal{T}_{e_i}), (0, s_{u_1}, \mathcal{T}_{u_1}), \ldots, (0, s_{u_j}, \mathcal{T}_{u_j}), (\delta_{d_1} - 1, s_{d_1}, \mathcal{T}_{d_1}), \ldots, (\delta_{d_k} - 1, s_{d_k}, \mathcal{T}_{d_k})\}, \sigma'\rangle}$$

$$\text{(BL:Sync)}\quad \frac{\forall i, 1 \le i \le n. s_i \to s'_i}{\langle \{(0, s_1, \mathcal{T}_1), \ldots, (0, s_n, \mathcal{T}_n)\}, \sigma\rangle \Rrightarrow_{M;P} \langle \{(0, s'_1, \mathcal{T}_1), \ldots, (0, s'_n, \mathcal{T}_n)\}, \sigma\rangle}$$

Fig. 17. Rules for stepping a block.

$$\text{(BE:Full)}\quad \frac{\sigma; C; s_1 \Downarrow_M^{\mathcal{T}_1} \sigma'_1; C'_1; \mathsf{skip} \ldots \sigma'_{n-1}; C'_{n-1}; s_n \Downarrow_M^{\mathcal{T}_n} \sigma'; C'; \mathsf{skip}}{\sigma; C; \{(\delta_1, s_1, \mathcal{T}_1), \ldots, (\delta_n, s_n, \mathcal{T}_n)\} \Downarrow_M \sigma'; C'}$$

$$\text{(BE:Partial)}\quad \frac{\sigma; C; s_1 \Downarrow_M^{\mathcal{T}_1} \sigma'_1; C'_1; s'_1 \ldots \sigma'_{n-1}; C'_{n-1}; s_n \Downarrow_M^{\mathcal{T}_n} \sigma'; (W, S); s'_n \qquad \sigma'; (W, (\max S)_{Warps}); \{(\delta_1, s'_1, \mathcal{T}_1), \ldots, (\delta_n, s'_n, \mathcal{T}_n)\} \Downarrow_M \sigma''; C'' \qquad \forall i \in [1, n]. s'_i \neq \mathsf{skip}}{\sigma; C; \{(\delta_1, s_1, \mathcal{T}_1), \ldots, (\delta_n, s_n, \mathcal{T}_n)\} \Downarrow_M \sigma''; C''}$$

Fig. 18. Multi-warp cost semantics rules.

once, which will indeed be done by all warps within a block; differences in exact timing need not concern us in a big-step semantics. However, in the bounded implementation, which is a more precise model of GPU execution, warps within a block execute independently and may get out of sync, and we still need to be able to apply the cost semantics to an intermediate state of evaluation.

Figure 18 defines the cost semantics rules for executing a block to completion. Rule BE:Full is simply the block-level version of Rule SF:Full of Figure 12; it takes a block, all of whose warps are on the last component of execution, and evaluates all of them to skip using the partial evaluation judgments of Figures 10 and 11. Note that either all or none of the warps in a block should evaluate to skip as it is an error for only some threads in a block to synchronize. Rule BE:Partial is the block-level version of Rule SF:Partial of Figure 12. It evaluates each warp using the partial evaluation judgments of Figures 10 and 11; this rule applies when all of the warps evaluate only partially, returning a continuation. The rule then recursively evaluates the continuations (which will in turn use either BE:Full or another instance of BE:Partial) to continue evaluating all the warps in the block.

Coq mechanization of block-level evaluation. We represent a warp configuration triple $(\delta, s, \mathcal{T})$ as a record `cfg` consisting of a `stmt`, a `list thread`, and a `nat` representing the delay. The type `block` is defined as `list cfg`. The rules BL:Step and BL:Sync are encoded as an inductive definition `par_step_or_sync` with two constructors corresponding to the two two step rules. The constructor for BL:Step itself is broken into a separate definition `par_step` with two constructors that account for two slightly different situations, one in which all warps step and one in which only a subset do. The premises for these constructors involve definitions `step_mult` and `sync_step_mult`, which generalize the `step_stmt` and `sync_step` judgments, respectively, to multiple warps. They also use the judgment `disjoint_dom_blocks : block -> block -> Prop`, which states that two blocks consist of disjoint sets of threads. This judgment allows us to, for example, divide B into disjoint sets of warps E, U, and D as required by BL:Step (in the Coq

formalization, the requirements of the $B \xrightarrow{P-Greedy} (E, U, D)$ judgment are directly encoded as judgements over the three sub-blocks in the premises of the appropriate constructor for `par_step`).

The rules of Figure 18 are encoded as inductive definitions `partial_eval_block` and `par_eval_block`: the former applies `partial_eval_stmt` (from Section 5) to each warp in a block, and the latter runs `partial_eval_block` on each component of an execution until the program completes (similar to how `par_eval_stmt` generalizes `partial_eval_stmt`).

Finally, the inductive definition `run_par` runs `par_step_or_sync` for a given number of steps, and is used to state the final theorem.

Soundness of bounded implementation. Lemma 10 extends Lemmas 6 and 7 to show that if a block takes a (greedy) parallel step, this either decreases the work by P or decreases the span by S. The intuition is that there are two cases, depending on how many warps are ready to execute. If at least P warps are ready, then a greedy selection will execute P of them, decreasing the remaining work by P. If fewer than P warps are ready, then a greedy selection will execute all ready warps. All warps, whether they are executed, delayed, or waiting to synchronize with others, thus become one step closer to becoming ready and so their span decreases by one.

Lemma 10. *If* $\langle B, \sigma\rangle \Rrightarrow_{M;P} \langle B', \sigma'\rangle$ *and* $\sigma'; \emptyset; B' \Downarrow_M \sigma''; C''$, *then* $\sigma; \emptyset; B \Downarrow_M \sigma''; C$ *and either:*
(1) $W(C) - W(C'') \geq P$ *and* $S(C'') \leq S(C)$ *or*
(2) $S(C'') < S(C)$ *and* $W(C'') \leq W(C)$.

Proof. By inductive application of Lemmas 6 and 7. Proven in Coq. □

This lemma is the key result needed to bound the number of steps in a small-step execution by the aspirational bound given in Theorem 2.

Theorem 6. *If* $\langle B, \sigma\rangle \Rrightarrow^n_{M;P} \langle B', \sigma'\rangle$ *and for all* $(\delta, s, \mathcal{T}) \in B'$, *we have* $s = \mathsf{skip}$, *then* $\sigma; \emptyset; B \Downarrow_M \sigma'; C$ *and* $n \leq \frac{W(C)}{P} + S(C)$.

Proof. The proof is fully formalized in Coq. However, because of the importance of this theorem, we give a high-level version of the proof here.

By Rule SC:Skip and the rules of Figure 18, we have $\sigma'; \emptyset; B' \Downarrow_M \sigma'; \emptyset$. Proceed by induction on the derivation of $\langle B, \sigma\rangle \Rrightarrow^n_{M;P} \langle B', \sigma'\rangle$. If $B = B'$ and $n = 0$, then the result is trivial. Suppose $\langle B, \sigma\rangle \Rrightarrow_{M;P} \langle B'', \sigma''\rangle \Rrightarrow^{n-1}_{M;P} \langle B', \sigma'\rangle$. Then, by induction, $\sigma''; \emptyset; B'' \Downarrow_M \sigma'; C'$ and $n - 1 \leq \frac{W(C')}{P} + S(C')$. By Lemma 10, we have $\sigma; \emptyset; B \Downarrow_M \sigma''; C'$ and $W(C) - W(C') \geq P$ and $S(C') \leq S(C)$ or $S(C') < S(C)$ and $W(C') \leq W(C)$. In the first case, we have

$$n \leq \frac{W(C') + P}{P} + S(C') \leq \frac{W(C)}{P} + S(C).$$

In the second case, since we have constrained latencies to be integers,

$$n \leq \frac{W(C')}{P} + S(C') + 1 \leq \frac{W(C)}{P} + S(C).$$ □

7 INFERENCE AND IMPLEMENTATION

In this section, we discuss the implementation of the logical conditions and potential functions of Section 4 and the techniques used to automate the reasoning in our tool RaCuda. The design of the Boolean conditions and potential functions are similar to existing work [Carbonneaux et al. 2017, 2015]. We have implemented the inference algorithms as an extension to the Absynth tool [Carbonneaux et al. 2017; Ngo et al. 2018]. We begin by outlining our implementation, and then detail the instantiations of the potential annotations and logical conditions.

Implementation Overview. Absynth is an implementation of AARA for imperative programs. The core analysis is performed on a **control-flow-graph (CFG)** intermediate representation. Absynth first applies standard abstract interpretation to gather information about the usage and contents of program variables. It then generates templates for the potential annotations for each node in the graph and uses syntax-directed rules similar to those of the quantitative Hoare logic to collect linear constraints on coefficients in the potential templates throughout the CFG. These constraints are then solved by an LP solver.

RaCuda uses a modified version of Front-C[9] to parse CUDA. A series of transformations lowers the code into a representation similar to miniCUDA, and then to IMP, a resource-annotated imperative calculus that serves as a front-end to Absynth. Part of this process is adding annotations that express the cost of each operation in the given resource metric.

We have extended the abstract interpretation pass to gather CUDA-specific information that allows us to bound the values of the functions MemReads(·) and Conflicts(·) (recall that these functions were used to select which rules to apply in the program logic of Figure 8, but their definitions were left abstract). We describe the extended abstraction domain in the next subsection. For the most part, we did not modify the representation of potential functions, but we briefly discuss this representation at the end of this section.

Logical Conditions. The logical conditions of the declarative rules of Section 4 correspond to the abstraction domain we use in the abstract interpretation. The abstract interpretation is designed to gather information that will be used to select the most precise rules (based on memory accesses, bank conflicts and divergent warps) when applying the program logic. The exact implementation of the abstraction domain and analysis is therefore orthogonal to the implementation of the program logic, and is somewhat more standard (see Section 10 for comparisons to related work). Still, for completeness, we briefly describe our approach here.

The abstraction domain is a pair $(\mathcal{P}, \mathcal{V})$. The first component is a set of constraints of the form $\sum_{x \in Var} k_x x + k \leq 0$, where $k_x, k \in \mathbb{N}$. These form a constraint system on the runtime values of variables, which we can decide using Presburger arithmetic. The second component is a mapping that stores, for each program variable x, whether x may currently be used as a potential-carrying variable (see the discussion in Section 4). It also stores two projections of x's abstract value, one notated $\mathcal{V}_{\text{tid}}(x)$ that tracks its dependence on tid (in practice, this consists of three components tracking the projections of x's dependence on the x, y and z components of `threadIdx`, which is three-dimensional as described in Section 2) and one notated $\mathcal{V}_{\text{const}}(x)$ that tracks its constant component. Both projections are stored as polynomial functions of other variables, or $\top$, indicating no information about that component. These projections provide useful information for the CUDA-specific analysis. For example, if $\mathcal{V}_{\text{tid}}(x) = (0, 0, 0)$, then the value of x is guaranteed to be constant across threads. As another example, if $\mathcal{V}_{\text{tid}}(x) = (1, 1, 1)$, then $x = \text{tid} + c$, where c does not depend on the thread, and so the array access $A[x]$ has a stride of 1.

The extension of the analysis to expressions, and its use in updating information at assignments and determining whether conditional expressions might diverge, is straightforward. The use of this information to predict uncoalesced memory accesses and bank conflicts is more interesting. We assume the following definitions of MemReads(·) and Conflicts(·), now generalized to use m as the number of array elements accessed by a global read and B as the number of shared memory banks.

$$\begin{aligned} \text{MemReads}(R) &\triangleq \left|\left\{\left\lceil \tfrac{i}{m} \right\rceil \mid (i)_t \in R\right\}\right|, \\ \text{Conflicts}(R) &\triangleq \max_{j \in [0, B-1]} \{a \equiv j \mod B \mid a \in \mathit{Cod}(R)\}. \end{aligned}$$

[9] https://github.com/BinaryAnalysisPlatform/FrontC

Theorem 7 formalizes and proves the soundness of a bound on MemReads(x) given abstract information about x.

Theorem 7. *If* $\mathcal{V}_{\text{tid}}(x) = k$ *and* $\mathcal{P} \Rightarrow \text{tid} \in [t, t']$ *and* $\sigma, \mathcal{T} \vDash (\mathcal{P}, \mathcal{V})$ *and* $\sigma; x \downarrow_M^{\mathcal{T}} R; C$, *then* $\mathit{MemReads}(R) \leq \lceil \frac{k(t'-t)}{m} \rceil + 1$.

Proof. By the definition of $\sigma, \mathcal{T} \vDash (\mathcal{P}, \mathcal{V})$, we have $\mathcal{T} \subset [t, t']$. Let $a = \min_{t \in \mathcal{T}} R(t)$ and $b = \max_{t \in \mathcal{T}} R(t)$. We have

$$\text{MemReads}(R) \leq \left\lfloor \frac{b}{m} \right\rfloor - \left\lfloor \frac{a}{m} \right\rfloor + 1 \leq \frac{b-a}{m} + 2 \leq \left\lceil \frac{b-a}{m} \right\rceil + 1 \leq \left\lceil \frac{k(t'-t)}{m} \right\rceil + 1. \qquad \square$$

Theorem 8 proves a bound on Conflicts(x) given abstract information about x. This bound assumes that x is divergent; for non-divergent operands o, by assumption, we have Conflicts(o) = 1. The proof relies on Lemma 11, a stand-alone result about modular arithmetic.

Theorem 8. *If* $\mathcal{V}_{\text{tid}}(x) = k > 0$ *and* $\mathcal{P} \Rightarrow \text{tid} \in [t, t']$ *and* $\sigma, \mathcal{T} \vDash (\mathcal{P}, \mathcal{V})$ *and* $\sigma; x \downarrow_M^{\mathcal{T}} R; C$, *then*

$$\mathit{Conflicts}(R) \leq \left\lceil \frac{t'-t}{\min(t'-t, \frac{B}{\gcd(k,B)})} \right\rceil.$$

Proof. Let $t_0 \in \mathcal{T}$. By the definition of $\sigma, \mathcal{T} \vDash (\mathcal{P}, \mathcal{V})$, we have $\mathit{Tid}(t_0) \in [t, t']$. We have $R(t_0) = kt_0 + c$. Let $R' = (kt \mod B)_{t \in \mathcal{T}}$. The accesses in R access banks from R' at uniform stride, and so the maximum number of times any such bank is accessed in R is $\lceil \frac{t'-t}{|\mathit{Dom}(R')|} \rceil$. The result follows from Lemma 11. $\square$

Lemma 11. *Let* $k, m, n, a \in \mathbb{N}$ *and* $m \leq n$. *Then* $|\{i \cdot a \mod n \mid i \in \{k, \ldots, k+m-1\}\}| = \min(m, \frac{n}{\gcd(a,n)})$.

Proof. Let $c = \frac{\text{lcm}(a,n)}{a} = \frac{n}{\gcd(a,n)}$. Then $A = \{ka, 2ka, \ldots, (k+c-1)a\}$ is a residue system modulo n (that is, no two elements of the set are congruent modulo n), because if $ik \cdot a \equiv jk \cdot a \mod n$ for $j - i \leq c$, then $ak(j-i)$ is a multiple of a and n smaller than ca, which is a contradiction. This means that if $m \leq c$, then $|\{i \cdot a \mod n \mid i \in \{k, \ldots, k+m-1\}\}| = m$. Now consider the case where $m > c$ and let $c + k < i < m + k$. Let $b = (i-k) \mod c$. We then have $(i-k)a \equiv ba \mod n$, and so ia is already included in A. Thus, $\{i \cdot a \mod n \mid i \in \{k, \ldots, k+m-1\}\} = A$. $\square$

As an example of how the abstraction information is tracked and used in the resource analysis, we return to the code example in Figure 9. Figure 19 steps through the abstract interpretation of the same code. For the purposes of this example, we have declared two variables `temp0` and `temp1` to hold intermediate computations. This reflects more closely the intermediate representation on which the abstract interpretation is done. We establish $\mathcal{P}$ from parameters provided to the analysis that specify that `blockDim.x` is 32, which also bounds `threadIdx.x`. The assignment to `i` on line 3 then establishes that `i` is a multiple of `threadIdx.x` and has a constant component of 32`blockIdx.x`. This information is then used on line 4 to bound MemReads(`i`) and Conflicts(`threadIdx.x`). By Theorem 7, we can bound MemReads(`i`) by $\lceil \frac{32}{m} \rceil + 1$. Note that both t and t' in the statement of Theorem 7 are multiples of 32 (and, in practice, m will divide 32), so we can improve the bound to $\lceil \frac{32}{m} \rceil$. By Theorem 8, we can bound Conflicts(`threadIdx.x`) by $\lceil \frac{32}{\min(32, \frac{32}{\gcd(1,32)})} \rceil = 1$.

When `j` is declared, it is established to have no thread-dependence. Its constant component is initially zero, but the loop invariant sets $\mathcal{V}_{\text{const}}(\texttt{j}) = \top$. The assignments on lines 6 and 8 propagate the information that `i` depends on `threadIdx.x` as well as some additional information about the

```
1  __global__ void addSub3 (int *A, int *B, int w, int h) {
2    __shared__ int As[blockDim.x];
3    int i = blockIdx.x * blockDim.x + threadIdx.x;
4    As[threadIdx.x] = A[i];
5    for (int j = 0; j < h; j += 2) {
6      int temp0 = j * w + i;
7      B[temp0] += As[i];
8      int temp1 = (j + 1) * w + i;
9      B[temp1] -= As[i];
10   }
11 }
```

$\texttt{blockDim.x} = 32, \texttt{threadIdx.x} < 32$

$\mathcal{V}_{\textbf{tid}}(\texttt{i}) = (1,0,0), \mathcal{V}_{\textbf{const}}(\texttt{i}) = 32\texttt{blockIdx.x}$

$\mathcal{V}_{\textbf{tid}}(\texttt{j}) = (0,0,0), \mathcal{V}_{\textbf{const}}(\texttt{j}) = \top$

$\mathcal{V}_{\textbf{tid}}(\texttt{temp0}) = (1,0,0), \mathcal{V}_{\textbf{const}}(\texttt{temp0}) = \texttt{j} * \texttt{w}$

$\mathcal{V}_{\textbf{tid}}(\texttt{temp1}) = (1,0,0), \mathcal{V}_{\textbf{const}}(\texttt{temp1}) = (\texttt{j}+1) * \texttt{w}$

Fig. 19. Sample abstract interpretation.

constant components. This information is used in the two global loads to bound MemReads(`temp0`) and MemReads(`temp1`) by $\lceil \frac{32}{m} \rceil + 1$. In this case, without further information about the value of `w`, we are unable to make any assumptions about alignment and cannot derive a more precise bound. As above, we can determine Conflicts(`i`) ≤ 1 for the loads on both lines.

Potential functions. Our implementation of potential functions is taken largely from prior work on AARA for imperative programs [Carbonneaux et al. 2017; Ngo et al. 2018]. We instantiate a potential function Q as a linear combination of a fixed set I of *base functions* from stores to rational costs, each depending on a portion of the state. A designated base function b_0 is the constant function and tracks constant potential. For each program, we select a set of N base functions, plus the constant function, notated $b_0, b_1, \ldots, b_N$, that capture the portions of the state relevant to calculating potential. A potential function Q is then a linear combination of the selected base functions:

$$Q(\sigma) = q_0 + \sum_{i=1}^{N} q_i b_i(\sigma).$$

In the analysis we use, base functions are generated by the following grammar:

$$\begin{array}{lll} M & ::= & 1 \mid x \mid M \cdot M \mid |[P, P]|, \\ P & ::= & k \cdot M \mid P + P. \end{array}$$

In the above, x stands for a program variable and $k \in \mathbb{Q}$ and $|[x, y]| = \max(0, y - x)$. The latter function is useful for tracking the potential of a loop counter based on its distance from the loop bound (as we did in Figure 9). These base functions allow the computation of intricate polynomial resource bounds; transferring potential between them is accomplished through the use of *rewrite functions*, described in more detail in prior work [Carbonneaux et al. 2017].

8 EVALUATION

We evaluated the range and and precision of RaCuda's analysis on a set of benchmarks drawn from various sources. In addition, we evaluated how well the cost model we developed in Section 3 approximates the actual cost of executing kernels on a GPU—this is important, because our analysis (and its soundness proofs) target our cost model, so the accuracy of our cost model is as important a factor in the overall performance of the analysis as is the precision of the analysis itself. Table 2 lists the benchmarks we used for our experiments. For each benchmark, the table lists the source (benchmarks were either from sample kernels distributed with the CUDA Toolkit, modified from such kernels by us, or written entirely by us). For benchmarks taken from the CUDA Toolkit, we used the samples distributed with v10.2 of the Toolkit, and we list in footnotes the path within the samples directory of each kernel. The table also shows the number of lines of code in each kernel, and the arguments to the kernel whose values appear as parameters in the

Table 2. Benchmark Suite Used for Our Experiments

Benchmark	Source	LoC	Params.	Block
matMul	SDK[10]	26	N	32×32
matMulBad	SDK*	27	N	32×32
matMulTrans	SDK*	26	N	32×32
mandelbrot	SDK[11]	78	N	32×1
vectorAdd	SDK[12]	5	N	256×1
reduceN	SDK[13]	14–18	N	256×1
histogram256	SDK[14]	19	N	64×1
SYN-BRDIS	[Han and Abdelrahman 2011]*	31	M, N	32×1
SYN-BRDIS-OPT	[Han and Abdelrahman 2011]*	29	M, N	32×1
addSub0	Us	9	h, w	$h \times 1$
addSub1	Us	7	h, w	$\frac{h}{2} \times 1$
addSub2	Us	7	h, w	$w \times 1$
addSub3	Us	8	h, w	$w \times 1$

SDK = benchmarks distributed with CUDA Toolkit—the path is given in a footnote.
SDK* = benchmarks derived from SDK by authors.
[Han and Abdelrahman 2011]* = benchmarks for warp divergence by Han and Abdelrahman [2011] and modified by authors.
Us = benchmarks written by authors.

cost results. The kernels used may appear small, but they are representative of CUDA kernels used by many real applications; recall that a CUDA kernel corresponds essentially to a single C function and that an application will likely combine many kernels used for different purposes. We also give the x and y components of the block size we used as a parameter to the analysis for each benchmark (a z component of 1 was always used). Some of the benchmarks merit additional discussion. The matrix multiplication (matMul) benchmark came from the CUDA SDK; we also include two of our own modifications to it: one that deliberately introduces a number of performance bugs (matMulBad), and one (matMulTrans) that transposes one of the input matrices in an (as it happens, misguided) attempt to improve shared memory performance. For all matrix multiplication kernels, we use square matrices of dimension N. The CUDA SDK includes several versions of the "reduce" kernel (collectively reduceN), in which they iteratively improve performance between versions. We include the first four in our benchmark suite; later iterations use *cooperative groups*, an advanced feature of CUDA that we do not currently support (see Section 9 for a discussion of this and other limitations). Kernels reduce2 and reduce3 use complex loop indices that confuse our inference algorithm for some benchmarks, so we performed slight manual refactoring on these examples (kernels reduce2a and reduce3a) so our algorithm can derive bounds for them—we include both the original and modified versions. We also include the examples from Section 2 (addSubN). Finally, for evaluation of warp divergences, we include a synthetic kernel SYN-BRDIS developed by Han and Abdelrahman [2011] as an example of an optimization to reduce the impact of warp divergences, as well as SYN-BRDIS-OPT, which is the result of applying their optimization to SYN-BRDIS. We analyzed each benchmark under the four resource metrics defined in Table 1.

[10] `v10.2/0_Simple/matrixMul`.
[11] `v10.2/2_Graphics/Mandelbrot`.
[12] `v10.2/0_Simple/vectorAdd`.
[13] `v10.2/6_Advanced/reduction`N.
[14] `v10.2/3_Imaging/histogram`.

8.1 Evaluation of the Cost Model

Two of the resource metrics above, "conflicts" and "sectors," correspond directly to metrics collected by NVIDIA's NSight Compute profiling tool for CUDA. This allows us to compare the upper bounds predicted by RaCuda with actual results from CUDA executions (which we do in the next subsection) as well as to evaluate how closely the cost semantics we presented in Section 3 tracks with the real values of the corresponding metrics, which we now discuss.

To perform this comparison, we equipped RaCuda with an "evaluation mode" that simulates execution of the input kernel using rules similar to the cost semantics of Figure 5 under a given execution metric. The evaluation mode, like the analysis mode, parses the kernel and lowers it into the miniCUDA-like representation. The kernel code under this representation is then interpreted by the simulator. In addition, the evaluation mode takes an input file specifying various parameters such as the block and grid size, as well as the arguments passed to the kernel, including arrays and data structures stored in memory.

We ran each kernel on a range of input sizes. For kernels whose performance depends on the contents of the input, we used worst-case inputs. For the histogram benchmark, whose worst-case "conflicts" value depends heavily on the input (we will discuss this effect in more detail below), limitations of the OCaml implementation prevent us from simulating the worst-case input. We therefore leave this benchmark out of the results in this subsection.

Because of edge effects from an unfilled last warp, the precision of our analysis often depends on N mod 32 where N is the number of threads used. To quantify this effect, where possible, we tested inputs that were $32N$ for some N, as well as inputs that were $32N+1$ for some N (which will generally be the best and worst case for precision) as well as random input sizes drawn uniformly from an appropriate range. The matrix multiplication, reduce and histogram benchmarks require (at least) that the input size is a multiple of 32, so we are not able to report on non-multiple-of-32 input sizes for these benchmarks. We report the average error for each class of input sizes separately, as well as in aggregate. Average error between the simulated value (derived from our tool simulating execution under the cost semantics) and the profiled value (taken from a GPU execution profiled with NSight Compute) is calculated as

$$\frac{1}{|Inputs|} \sum_{i \in Inputs,\, Profiled(i) \neq 0} \frac{|Simulated(i) - Profiled(i)|}{Profiled(i)},$$

neglecting inputs that would cause a division by zero. In almost all cases in which the cost semantics is not exactly precise, the cost semantics overestimates the actual cost of GPU execution (for some inputs, the cost semantics provides a slight underestimate but these differences are small enough that they do not appear in the average results below). Because the soundness result (Theorem 1) applies to the soundness of the analysis with respect to the cost semantics, this gives some assurance that the analysis is also a sound upper bound with respect to actual execution values.

Table 3 reports the average relative error of the cost model with respect to profiled values. In many cases, the cost model is extremely precise (exact or within 5%). Larger differences in a small number of cells could be due to a number of factors: CUDA's memory model is quite complex, and we model only part of the complexity. In addition, our simulator does not track all values and parameters, sometimes relying on default values or symbolic execution. Finally, our simulator is essentially an interpreter running the CUDA source code, while the GPU executes code that has been compiled to a special-purpose assembly language. We do not attempt to model performance differences that may be introduced in this compilation process.

Table 3. Error of the Cost Semantics with Respect to Profiled Values for the "sectors" and "conflicts" Metrics

	Error on metric "sectors"				Error on metric "conflicts"			
Benchmark	$32N$	$32N+1$	Rand.	Avg.	$32N$	$32N+1$	Rand.	Avg.
matMul	5.5×10^{-5}			5.5×10^{-5}	0			0
matMulBad	0.01			0.01	0			0
matMulTrans	2.6×10^{-4}			2.6×10^{-4}	0			0
mandelbrot	0	0	0	0	0	0	0	0
vectorAdd	0	0.06	0.38	0.15	0	0	0	0
reduce0	0.21			0.21	0			0
reduce1	0.21			0.21	5.93			5.93
reduce2	0.21			0.21	0			0
reduce3	1.22			1.22	0			0
addSub0	5.1×10^{-3}	4.7×10^{-3}	0.01	5.1×10^{-3}	0	0	0	0
addSub1	4.5×10^{-3}	0.01	0.01	0.01	0	0	0	0
addSub2	0	0.02	0.03	0.01	0	0	0	0
addSub3	0	0.03	0.04	0.02	0	0	0	0

8.2 Evaluation of the Analysis

In this subsection, we compare the execution costs predicted by RaCuda with the actual execution costs obtained by profiling (for the "conflicts" and "sectors" metrics) or the simulated costs from our cost semantics (for the "divwarps" and "steps" metrics). We discuss the "conflicts" and "sectors" metrics first. Tables 4 and 5 contain the results for these two metrics. For each benchmark, we present the total time taken, and the cost bound inferred, by RaCuda's analysis. The timing results show that RaCuda is quite efficient, with analysis times usually under 1 second; analysis times on the order of minutes are seen for exceptionally complex kernels. Recall that RaCuda produces bounds for a single warp. To obtain bounds for the kernel, we multiplied by the number of warps required to process the entire input (often $\lceil \frac{N}{32} \rceil$ if the input size is N). Several of the kernels perform internal loops with a stride of 32. Precise bounds of such loops would, like the number of warps, be of the form $\lceil \frac{N}{32} \rceil$. However, Absynth can only produce polynomial bounds and so must approximate this bound by $\frac{N+31}{32}$, which is the tightest possible polynomial bound.

We also ran versions of each kernel on a GPU using NSight Compute. Similar to the above error calculation for evaluating the cost model, average error is calculated as

$$\frac{1}{|Inputs|} \sum_{i \in Inputs, Actual(i) \neq 0} \frac{Predicted(i) - Actual(i)}{Actual(i)},$$

neglecting inputs that would cause a division by zero.

The analysis for global memory sectors is fairly precise. Note also that, for the reduce kernels, the error is the same as the error of the cost model in Table 3; this indicates that the analysis precisely predicts the modeled cost and the imprecision is in the cost model, rather than the analysis. Most other imprecisions are because our abstraction domain is insufficiently complex to show, e.g., that memory accesses are properly aligned. We note, however, that more efficient versions of the same kernel (e.g., the successive versions of the reduce and addSub kernels) generally appear more efficient under our algorithm, and also that our analysis is most precise for better-engineered kernels that follow well-accepted design patterns (e.g., matMul, reduce3a, addSub3). These results indicate that our analysis can be a useful tool for improving CUDA kernels, because it can give useful feedback on whether modifications to a kernel have indeed improved its performance.

Table 4. Analysis Time, Inferred Bound, and Average Error for the "conflicts" Metric

			Error			
Benchmark	Time (s)	Predicted Bound	$32N$	$32N+1$	Random	Average
matMul	0.11	$31(31+N)\left\lceil\frac{N}{32}\right\rceil$	—			—
matMulBad	8.13	0	0			0
matMulTrans	0.12	$1023\frac{31+N}{32}\left\lceil\frac{N}{32}\right\rceil$	33.4			33.4
mandelbrot	346.88	0	0	0	0	0
vectorAdd	0.00	0	0	0	0	0
reduce0	0.03	0	0			0
reduce1	0.02	$23715\left\lceil\frac{N}{32}\right\rceil$	1806			1806
reduce2	1.07	n/b	n/b			n/b
reduce2a	0.03	0	0			0
reduce3	1.54	n/b	n/b			n/b
reduce3a	0.03	0	0			0
histogram	1.19	$56\frac{63+N}{64}$	0			0
addSub0	0.18	0	0	0	0	0
addSub1	0.01	0	0	0	0	0
addSub2	0.01	0	0	0	0	0
addSub3	0.01	0	0	0	0	0

For matMul, the correct value is 0 for all inputs. An entry of "n/b" indicates that our analysis was unable to determine a bound.

Table 5. Analysis Time, Inferred Bound, and Average Error for the "sectors" Metric

			Error			
Benchmark	Time (s)	Predicted Bound	$32N$	$32N+1$	Random	Average
matMul	1.05	$(4+10\frac{31+N}{32})\left\lceil\frac{N}{32}\right\rceil$	0.30			0.30
matMulBad	8.31	$15(31+N)\left\lceil\frac{N}{32}\right\rceil$	0.20			0.20
matMulTrans	1.05	$(4+10\frac{31+N}{32})\left\lceil\frac{N}{32}\right\rceil$	0.30			0.30
mandelbrot	350.31	36N	0.13	0.13	0.13	0.13
vectorAdd	0.00	$12\left\lceil\frac{N}{32}\right\rceil$	0.00	3.4×10^{-4}	7.2×10^{-5}	1.4×10^{-4}
reduce0	0.03	$5\left\lceil\frac{N}{32}\right\rceil$	0.21			0.21
reduce1	0.02	$5\left\lceil\frac{N}{32}\right\rceil$	0.21			0.21
reduce2	0.09	$5\left\lceil\frac{N}{32}\right\rceil$	0.21			0.21
reduce3	0.10	$9\left\lceil\frac{N}{32}\right\rceil$	1.22			1.22
histogram	1.14	$\frac{1}{64}(5740+10(63+N))$	0.25			0.25
addSub0	0.18	$132w\left\lceil\frac{h}{32}\right\rceil$	1.01	1.15	1.06	1.06
addSub1	0.01	$132w\left\lceil\frac{h}{64}\right\rceil$	0.02	0.17	0.06	0.07
addSub2	0.01	$14(h+1)\left\lceil\frac{w}{32}\right\rceil$	0.17	0.13	0.19	0.16
addSub3	0.01	$(4+10(h+1))\left\lceil\frac{w}{32}\right\rceil$	0.25	0.16	0.27	0.23

RaCuda is generally able to correctly infer that a kernel has no bank conflicts, but often overestimates the number of bank conflicts when some are present. Again, this means that RaCuda can be used to determine whether improvements need to be made to a kernel. We believe the bank conflicts analysis can be made more precise with a more complex abstraction domain.

Figure 20 plots our predicted cost versus the actual worst-case for two representative benchmarks. In the right figure, we plot executions of random inputs generated at runtime in addition

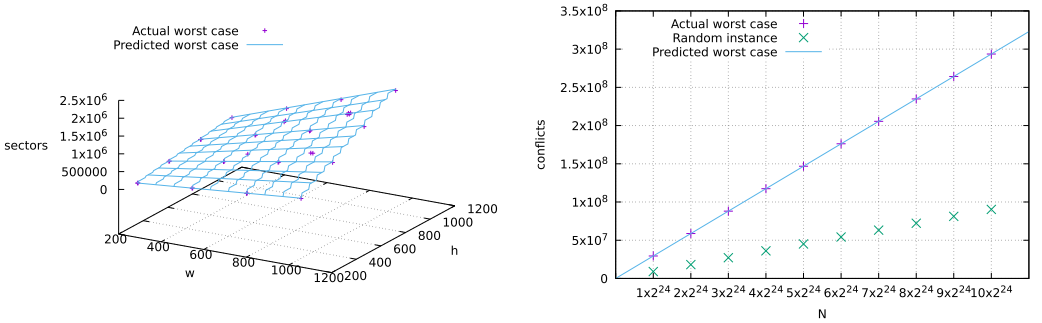

Fig. 20. Our inferred cost estimates (blue line) vs. actual worst-case costs (purple crosses) for various inputs. Left: addSub1, sectors; right: histogram, conflicts (also includes a random input, green crosses).

to the worst-case input. The benchmark used for this figure is histogram, whose shared memory performance displays interesting behavior depending on the input. The benchmark computes a 256-bin histogram of the number of occurrences of each byte (0x0–0xff) in the input array. The bins are stored in shared memory, and so occurrences of bytes that map to the same shared memory bank (e.g., 0x00 and 0x20) in the same warp might result in bank conflicts. In the worst case, all bytes handled by a warp map to the same memory bank, resulting in an eight-way conflict at each access (32 bins are accessed, but as there are only 256 bins, only $256/32 = 8$ map to each memory bank). However, in a random input, bytes are likely to be much more evenly distributed. This figure shows the benefit of static analysis over random testing in safety-critical applications where soundness is important: at least in this benchmark, random testing is highly unlikely to uncover, or even approach, the true worst case.

We present the results for the "divwarps" and "steps" metrics in Tables 6 and 7. For the purposes of these tables, the bounds are shown per warp rather than for the entire kernel (composing the "steps" metric across multiple warps is not so straightforward, as we discuss in Section 5). Again, the analysis is fairly efficient, though in this table we see that the performance of RaCuda is harmed most by nesting of divergent conditionals and loops, as in the benchmarks SYN-BRDIS and SYN-BRDIS-OPT. Still, analysis times remain at most on the order of minutes (and are still under 1 second for most benchmarks). For the "steps" and "divwarps" metrics, we do not compare to a profiled GPU execution, because NVIDIA's profiling tools do not collect metrics directly related to these. Instead, we compare to RaCuda's "evaluation mode".

These experiments show the utility of RaCuda over tools that merely identify one type of performance bug. Often, there is a tradeoff between two performance bottlenecks. For example, reduce3 has worse global memory performance than reduce2, but performs the first level of the reduction immediately from global memory, reducing shared memory accesses. By combining these into a metric (e.g., "steps") that takes account the relative cost of each operation, we can explore the tradeoff: we see that in terms of "steps," reduce2 is more efficient than reduce3, but the situation will likely be reversed depending on the actual runtime costs of each operation, which we do not attempt to profile for the purposes of this evaluation. As another example, the SYN-BRDIS-OPT kernel was designed to reduce the impact of divergent warps over SYN-BRDIS using a transformation called *branch distribution*. Branch distribution factors out code common to two branches of a divergent conditional (for example, `if e then (A; B; C) else (A'; B; C')` would become `(if e then A else A'); B; (if e then C else C'))`. In this code example (and in the benchmarks), the transformation actually *increases* the *number* of divergences: we can see this in the "divwarps" metric for the two benchmarks. However, the total amount of code that must execute sequentially is decreased, as evidenced by the "steps" metric.

Table 6. Analysis Time, Inferred Bound, and Average Error for the "divwarps" Metric

			Error			
Benchmark	Time (s)	Predicted Bound	$32N$	$32N+1$	Random	Average
matMul	0.12	0	0	0	0	0
matMulBad	7.98	$31+N$	0.04	0	0.01	0.02
matMulTrans	0.12	0	0	0	0	0
mandelbrot	471.76	n/b	n/b	n/b	n/b	
vectorAdd	0.01	1	—	—	—	—
reduce0	0.03	257	27.56			27.56
reduce1	0.02	257	41.83			41.83
reduce2	1.06	n/b	n/b			n/b
reduce2a	0.03	129.5	24.90			24.90
reduce3	1.52	n/b	n/b			n/b
reduce3a	0.04	130.5	25.10			25.10
histogram	1.14	0	0			0
addSub0	0.22	w	0	0	0	0
addSub1	0.01	0	0	0	0	0
addSub2	0.01	0	0	0	0	0
addSub3	0.01	0	0	0	0	0
SYN-BRDIS	9.35	N	0	0	0	0
SYN-BRDIS-OPT	18.67	$2N$	0	0	0	0

An entry of "n/b" indicates that our analysis was unable to determine a bound.

Table 7. Analysis Time, Inferred Bound, and Average Error for the "steps" Metric

			Error			
Benchmark	Time (s)	Predicted Bound	$32N$	$32N+1$	Random	Average
matMul	0.12	$75+1621\frac{31+N}{32}$	1.69	1.58	1.61	1.63
matMulBad	9.11	$51+2678\frac{31+N}{32}$	0.07	0.01	0.03	0.04
matMulTrans	0.12	$75+1652\frac{31+N}{32}$	1.61	1.50	1.53	1.56
mandelbrot	557.40	n/b	n/b	n/b	n/b	
vectorAdd	0.01	27	0	0	0	0
reduce0	0.04	6412	26.06			26.05
reduce1	0.03	30892	87.77			87.77
reduce2	1.22	n/b	n/b			n/b
reduce2a	0.04	3352	14.31			14.31
reduce3	1.76	n/b	n/b			n/b
reduce3a	0.04	4139	13.63			13.63
histogram	1.38	$383.75+128\frac{63+N}{64}$	0.51			0.51
addSub0	0.22	$7+177w$	0.59	0.59	0.59	0.59
addSub1	0.02	$7+183w$	0.01	0.01	0.01	0.01
addSub2	0.01	$14+32(h+1)$	0.07	0.02	0.05	0.05
addSub3	0.01	$22+29(h+1)$	0.08	0.02	0.06	0.06
SYN-BRDIS	331.43	$9+85MN+64N$	0	0	0	0
SYN-BRDIS-OPT	234.97	$9+69MN+62N$	0	0	0	0

An entry of "n/b" indicates that our analysis was unable to determine a bound.

Table 8. Analysis Times and Results for Each Function of the GPU-MMS Kernel

Function	Lines of code	Analysis time (s)	Bound
`xorMergeNodes`	35	—	—
`fillEmptyLeaf`	6	0.075	0
`heapifyEmptyNode`	8	0.100	0
`buildHeap`	18	5.413	0
`multimerge`	21	30.664	0
`multimergeLevel`	25	0.702	0

Functions for which no bound was found within 10 min are marked with —.

8.3 Case Study: Conflict-free Merge Sort

In this subsection, we focus on one metric ("shared") but on a more extensive kernel than was studied in the previous subsections. The benchmark we consider is the GPU-MMS algorithm [Karsin et al. 2018], a GPU sorting algorithm based on mergesort. Specifically, we analyze an open-source CUDA implementation of GPU-MMS by the designers of the algorithm.[15] The design of GPU-MMS is motivated by a number of important performance considerations, one of which is bank conflicts. In particular, the authors claim that GPU-MMS is free of bank conflicts.

We made a number of small syntactic modifications to the code to fit within the current restrictions of the RaCuda tool (see Section 9 for more detail on these restrictions and their impact). Two of the biggest changes were specializing templated functions to a particular data type (we chose `int`), and converting bitwise operations (which are not supported by the IMP intermediate representation) to arithmetic operations and conditionals. In addition, the kernel consists of many functions (inlined at compile time). RaCuda can currently only analyze one function at a time (this is not a fundamental limitation, see Section 9). This in itself did not cause problems, except that arguments to functions are assumed to be passed in global memory, whereas in the GPU-MMS kernels, many of the arguments to these inlined functions have already been copied to shared memory. To analyze these kernels properly and not miss potential bank conflicts, we added declarations of these arguments as arrays in shared memory to the top of each function. For RaCuda's purposes, these declarations essentially act as annotations that a variable is in shared memory and do not otherwise affect the analysis. The modifications took under an hour by one programmer, and required no understanding of the algorithm.

The results are shown in Table 8, broken down for each function in the kernel. The number of lines of code is also given in the table, as well as the time taken by RaCuda and the bound calculated for the "shared" metric, that is, the number of bank conflicts in the function. For all but one function, RaCuda was able to, within 30 s, correctly confirm that no bank conflicts are present. One function hit our timeout of 10 min.

9 LIMITATIONS AND FUTURE WORK

The contributions of this work are primarily foundational: to our knowledge, this is the first article presenting this general a quantitative program logic for GPU kernels (see Section 10 for a thorough discussion of related work), and the focus was on developing the underpinnings of the logic rather than supporting a large subset of CUDA. In particular, the implementation of Section 7 supports essentially a subset of CUDA that is syntactic sugar for miniCUDA. For example, `for` and `do-while` loops are easily desugared into `while` loops and are supported. We support all base types present

[15] https://github.com/algoparc/GPU-MMS

in C and most builtin operators (with the primary exception of some bitwise operations, which are not easily converted to IMP's arithmetic operations).

In this section, we list features that are not currently supported but that could be the subject of future work.[16] Some of these features would require additional foundational research, while the challenges for others are mostly softare engineering. We can divide these features into two categories. The first category is extending the set of CUDA features supported by our model and RaCuda, which would allow us to analyze more kernels, though not necessarily at higher precision. The second (not entirely orthogonal) category, is lowering the level of abstraction of our analysis to more precisely model performance characteristics that are currently outside the granularity of our analysis. Examples of the first category of future work include:

— *Atomic Functions.* In some sense, adding support for atomic functions (e.g., `__atomicAdd()`) would be one of the more engineering-focused extensions. We could extend resource metrics with resource constants for each atomic function (or groups of similar functions). It would be possible to extend the front-end to recognize these functions, tag the costs appropriately, and convert them to IMP code that implements the desired semantics.

— *C++ Features.* Because the front-end of our implementation uses a modifed C parser, we support very few C++ extensions. Some of these do not pose conceptual challenges. As an example, the instantiation of a template has little impact on performance (other than that a type parameter could be instantiated with types of different sizes, impacting the calculation of MemReads$(\cdot)$), and so we could handle this by copying templated functions, with one copy for each combination of type sizes with which it is instantiated. Handling objects would be more difficult, as objects themselves could contain potential in an interesting way (e.g., in the size of an array object). Tracking potential in this way has been investigated in the context of object-oriented languages by Rodriguez [2010] and for pointer-based data structures by Atkey [2010].

— *Functions.* We currently analyze only one function at a time. We do not believe there to be any fundamental challenges in supporting kernels spread across multiple functions, like the GPU-MMS kernel of Section 8.3, as the underlying theory and Absynth implementation support modular analysis of functions and function calls. As described in Section 8.3, we would need a mechanism for tracking across function calls whether certain arguments are stored in shared or global memory.

— *Cooperative groups.* Cooperative groups allow programmer-specified groups of threads to synchronize (in the same spirit as `__syncthreads()` but not necessarily at the granularity of entire blocks). If the set of threads in a cooperative group is known at compile time and remains fixed, then our results can apply to a cooperative group instead of a block, as the results of Section 5 are not tied to any particular notion of a block. However, allowing the set of threads that synchronize to be changed at runtime in a way that is determined by runtime conditions would require additional research, because the thread-level parallelism results (Section 5) fundamentally assume that an entire block is synchronizing.

Next, we list features that could be added to the performance model that would increase the precision of the analysis:

— *Occupancy and instruction-level parallelism.* As discussed in Section 5, each SM can hold a number of warps. The precise number is determined by the kernel's demand for resources that are shared among warps on the same SM, for example registers and shared memory. This can lead to important performance trade-offs. For example, allowing each warp fewer

[16] Of course, a comprehensive list of features is infeasible. We focus on features common in modern CUDA kernels, especially in kernels we feel could benefit from the analysis of this article.

registers will allow the SM to fit more warps and increase thread-level parallelism (in our model, this increases the P value in Theorem 2), but may also add overhead and create artificial dependencies that reduce opportunities for instruction-level parallelism.

Although our resource analysis focuses on quantities related to execution time, AARA techniques have been used in the past to reason about high water marks for space usage (e.g., Reference [Hofmann and Jost 2003]). However, predicting bounds on quantities like register and shared memory usage uniformly for all inputs goes beyond the capabilities of current techniques. We speculate, however, that it may be possible to use extensions of such techniques, together with the formalism of this article, to reason about the shared memory requirements of kernels and, therefore, about one component of maximum occupancy.

To fully reason about the trade-off described in the first paragraph of this bullet point, we would need to model instruction-level parallelism. The dependency-graph-based parallelism models described in Section 5 could be used to model instruction-level parallelism if given information about data dependencies between instructions.

— *Caching*. We have, thus far, assumed a worst-case latency for memory accesses. This worst-case latency corresponds to the assumption that every access is a cache miss. We could relax this assumption in the presence of some model of cache behavior. This would make the model parametric over hardware parameters (cache size and layout) in a way that we have deliberately avoided. This type of lower-level reasoning is frequently done in the Worst-case Execution Time community, and we could use techniques from that community to introduce more precision into our analysis. As an example, we could extend the logical conditions of our program logic to track recently-accessed locations that are still in cache to reason about spacial and temporal locality.

Finally, we note some more fundamental limitations of our analysis (and, in some cases, of static analysis in general). Static analysis techniques such as ours neessarily rely on conservative approximations in cases where values may not be known ahead of time, to guarantee soundness. This is particularly important in cases where the branching behavior of the code depends upon features that cannot be predicted at compile time. If it cannot be proven, based on properties tracked in the program logic, that only one branch of a conditional (for example) can be taken, then the analysis must assume that either branch can be taken, resulting in a potential loss of precision. To maintain scalability, our analysis is fairly coarse-grained in its tracking of information, particularly about thread-local values of variables. If a variable whose value is not predictable or uniform across a warp is used as a loop variable, then the qualitative portion of our analysis will lose precision and the quantitative portion will generally be unable to find a bound. Symbolic and dynamic analyses take different approaches to the trade-off of soundness and precision. See Section 10 for a discussion of other related approaches.

10 RELATED WORK

Resource Bound Analysis. There exist many static analyses and program logics that (automatically or manually) derive sound performance information such as worst-case bounds for imperative [Carbonneaux et al. 2017; Gulwani et al. 2009; Kincaid et al. 2017; Sinn et al. 2014] and functional [Danner et al. 2012; Guéneau et al. 2018; Hoffmann et al. 2017; Hofmann and Jost 2003; Lago and Gaboardi 2011; Radiček et al. 2017] programs. However, there are very few tools for parallel [Hoffmann and Shao 2015] and concurrent [Albert et al. 2011; Das et al. 2018] execution and there are no such tools that take into account the aforementioned CUDA-specific performance bottlenecks.

Most closely related to our work is AARA for imperative programs and quantitative program logics. Carbonneaux et al. [2014] introduced the first imperative AARA in the form of a program

logic for verifying stack bounds for C programs. The technique has then been automated [Carbonneaux et al. 2017, 2015] using templates and LP solving and applied to probabilistic programs [Ngo et al. 2018]. A main innovation of this work is the development of an AARA for CUDA code: Previous work on imperative AARA cannot analyze parallel executions nor CUDA specific memory-access cost.

Parallel Cost Semantics. The model we use for reasoning about a CUDA block in terms of its work and span is derived from prior models for reasoning about parallel algorithms. The ideas of work and span (also known as *depth*) date back to work from the 1970s and 1980s showing bounds on execution time for a particular schedule [Brent 1974] and later for any greedy schedule [Eager et al. 1989]. Starting in the 1990s [Blelloch and Greiner 1995, 1996], parallel algorithms literature has used directed acyclic graphs (DAGs) to analyze the parallel structure of algorithms and calculate their work and span: the work is the total number of nodes in the DAG and the span is the longest path from source to sink. In the case of CUDA, we do not need the full generality of DAGs and so are able to simplify the notation somewhat, but our notation for costs of CUDA blocks in Section 5 remains inspired by this prior work. We build in particular on work by Muller and Acar [2016] that extended the traditional DAG models to account for latency (Muller and Acar were considering primarily costly I/O operations; we use the same techniques for the latency of operations such as memory accesses). Their model adds latencies as edge weights on the graph and redefines the span (but not work) to include these weights.

The idea of a bounded implementation for validating cost semantics was introduced by Blelloch and Greiner [1995, 1996]. The proof technique we use is inspired by Muller et al. [2018].

Analysis of CUDA Code. Given its importance in fields such as machine learning and high-performance computing, CUDA has gained a fair amount of attention in the program analysis literature in recent years. There exist a number of static [Alur et al. 2017; Cogumbreiro et al. 2021; Li and Gopalakrishnan 2010; Li et al. 2012; Liew et al. 2022; Pereira et al. 2016; Singhania 2018; Zheng et al. 2011] and dynamic [Boyer et al. 2008; Eizenberg et al. 2017; Peng et al. 2018; Wu et al. 2019] analyses for verifying certain properties of CUDA programs, but much of this work focused on *functional* properties, e.g., freedom from data races. Wu et al. [2019] investigate several classes of bugs, one of which is "non-optimal implementation," including several types of performance problems. They do not give examples of kernels with non-optimal implementations, and do not specify whether or how their dynamic analysis detects such bugs. PUG [Li and Gopalakrishnan 2010] and Boyer et al. [2008] focus on detecting data races but both demonstrate an extension of their race detectors designed to detect bank conflicts, with somewhat imprecise results. Kojima and Igarashi [2017] present a Hoare logic for proving functional properties of CUDA kernels. Their logic does not consider quantitative properties and, unlike our program logic, requires explicit reasoning about the sets of active threads at each program point, which poses problems for designing an efficient automated inference engine.

GKLEE [Li et al. 2012] is an analysis for CUDA kernels based on concolic execution, and targets both functional errors and performance errors (including warp divergence, non-coalesced memory accesses and shared bank conflicts). GKLEE requires some user annotations to perform its analysis. Horga et al. [2022] extended GKLEE to perform symbolic analysis for detecting, proposing inputs that trigger, or proving the absence of, bank conflicts. Alur et al. [2017] and Singhania [2018] have developed several static analyses for performance properties of CUDA programs, including uncoalesced memory accesses. Their analysis for detecting uncoalesced memory accesses uses abstract interpretation with an abstract domain similar to ours but simpler (in our notation, it only tracks $\mathcal{V}_{\text{tid}}(x)$ and only considers the values 0, 1, and -1 for it). Their work does not address shared bank conflicts or divergent warps. Moreover, they developed a separate analysis for each

type of performance bug. In this work, we present a general analysis that detects and quantifies several different types of performance bugs.

The systems described in the previous paragraph only detect performance errors (e.g., they might estimate what percentage of warps in an execution will diverge); they are not able to quantify the impact of these errors on the overall performance of a kernel. The analysis in this article has the full power of amortized analysis and is able to generate a resource bound, parametric in the relevant costs, that takes into account divergence, uncoalesced memory accesses and bank conflicts.

Other work has focused on quantifying or mitigating, but not detecting, performance errors. Bialas and Strzelecki [2016] use simple, tunable kernels to experimentally quantify the impact of warp divergence on performance using different GPUs. Their findings show that there is a nontrivial cost associated with a divergent warp even if the divergence involves some threads simply being inactive (e.g., threads exiting a loop early or a conditional with no "else" branch). This finding has shaped our thinking on the cost of divergent warps. Han and Abdelrahman [2011] present two program transformations that lessen the impact of warp divergence; they quantify the benefit of these optimizations but do not have a way of statically identifying whether a kernel contains a potential for divergent warps and/or could benefit from their transformations.

11 CONCLUSION

We have presented a program logic for proving qualitative and quantitative properties of CUDA kernels and proven it sound with respect to a model of GPU execution. We have used the logic to develop a resource analysis for CUDA as an extension to the Absynth tool, and shown that this analysis provides useful feedback on the performance characteristics of a variety of kernels.

This work has taken the first step toward automated static analysis tools for analyzing and improving performance of CUDA kernels. In the future, we plan to extend the logic to handle more advanced features of CUDA such as dynamic parallelism, by further embracing the connection to DAG-based models for dynamic parallelism (Sections 5 and 10).

The "steps" metric of Section 8 raises the question of whether it is possible to use our analysis to predict actual execution times by using appropriate resource metrics to analyze the work and span and combine them as in Section 5. Deriving such metrics is largely a question of careful profiling of specific hardware, which is outside the scope of this article, but in future work we hope to bring these techniques closer to deriving wall-clock execution time bounds on kernels. Doing so may involve further extending the analysis to handle *instruction-level parallelism*, which hides latency by beginning to execute instructions *in the same warp* that do not depend on data being fetched.

ACKNOWLEDGMENTS

The authors thank Aleksey Nogin and Michael Warren for helpful discussions and the initial suggestion to study bottlenecks in CUDA, Nikos Hardavellas for discussions about thread-level parallelism, and the anonymous reviewers for their valuable feedback.

REFERENCES

Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and German Puebla. 2011. Cost analysis of concurrent OO programs. In *Proceedings of the 9th Asian Symposium on Programming Languages and Systems (APLAS'11)*. https://doi.org/10.1007/978-3-642-25318-8_19

Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. 2017. GPUDrano: Detecting uncoalesced accesses in GPU programs. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 507–525. https://doi.org/10.1007/978-3-319-63387-9_25

Robert Atkey. 2010. Amortised resource analysis with separation logic. In *Proceedings of the 19th European Symposium on Programming (ESOP'10), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'10)*

(Lecture Notes in Computer Science), Andrew D. Gordon (Ed.), Vol. 6012. Springer, 85–103. https://doi.org/10.1007/978-3-642-11957-6_6

Piotr Bialas and Adam Strzelecki. 2016. Benchmarking the cost of thread divergence in CUDA. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, Konrad Karczewski, Jacek Kitowski, and Kazimierz Wiatr (Eds.). Springer International Publishing, Cham, 570–579. https://doi.org/10.1007/978-3-319-32149-3_53

Guy E. Blelloch and John Greiner. 1995. Parallelism in sequential functional languages. In *Functional Programming Languages and Computer Architecture*. Springer, 226–237. https://doi.org/10.1145/224164.224210

Guy E. Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*. ACM, 213–225. https://doi.org/10.1145/232629.232650

Michael Boyer, Kevin Skadron, and Westley Weimer. 2008. Automated dynamic analysis of CUDA programs. In *Proceedings of the 3rd Workshop on Software Tools for MultiCore Systems*.

Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (1974), 201–206. https://doi.org/10.1145/321812.321815

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *Proceedings of the 35th Conference on Programming Language Design and Implementation (PLDI'14)*. https://doi.org/10.1145/2594291.2594301

Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. 2017. Automated resource analysis with Coq proof objects. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 64–85. https://doi.org/10.1007/978-3-319-63390-9_4

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, New York, NY, 467–478. https://doi.org/10.1145/2737924.2737955

Tiago Cogumbreiro, Julien Lange, Dennis Liew Zhen Rong, and Hannah Zicarelli. 2021. Checking data-race freedom of GPU kernels, compositionally. In *Proceedings of the International Conference on Computer Aided Verification (CAV'21)*. Springer, 403–426. https://doi.org/10.1007/978-3-030-81685-8_19

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2012. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 29th International Conference on Functional Programming (ICFP'15)*. https://doi.org/10.1145/2858949.2784749

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel complexity analysis with temporal session types. In *Proceedings of the 23rd International Conference on Functional Programming (ICFP'18)*.

Derek L. Eager, John Zahorjan, and Edward D. Lazowska. 1989. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.* 38, 3 (1989), 408–423. https://doi.org/10.1109/12.21127

Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. BARRACUDA: Binary-level analysis of runtime RAces in CUDA programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, New York, NY, 126–140. https://doi.org/10.1145/3062341.3062342

Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *Proceedings of the 27th European Symposium on Programming (ESOP'18), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'18)*. 533–560. https://doi.org/10.1007/978-3-319-89884-1_19

Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)*. https://doi.org/10.1145/1480881.1480898

Tianyi David Han and Tarek S. Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'11)*. ACM, New York, NY, Article 3, 8 pages. https://doi.org/10.1145/1964179.1964184

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate amortized resource analysis. In *Proceedings of the 38th Symposium on Principles of Programming Languages (POPL'11)*. 357–370. https://doi.org/10.1145/2362389.2362393

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th Symposium on Principles of Programming Languages (POPL'17)*. https://doi.org/10.1145/3009837.3009842

Jan Hoffmann and Zhong Shao. 2015. Automatic static cost analysis for parallel programs. In *Proceedings of the 24th European Symposium on Programming (ESOP'15)*. https://doi.org/10.1007/978-3-662-46669-8_6

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. 185–197. https://doi.org/10.1145/640128.604148

Adrian Horga, Ahmed Rezine, Sudipta Chattopadhyay, Petru Eles, and Zebo Peng. 2022. Symbolic identification of shared memory based bank conflicts for GPUs. *J. Syst. Architect.* 127 (2022), 102518. https://doi.org/10.1016/j.sysarc.2022.102518

Ben Karsin, Volker Weichert, Henri Casanova, John Iacono, and Nodari Sitchinava. 2018. Analysis-driven engineering of comparison-based sorting algorithms on GPUs. In *Proceedings of the International Conference on Supercomputing (ICS'18)*. Association for Computing Machinery, New York, NY, 86–95. https://doi.org/10.1145/3205289.3205298

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional recurrence analysis revisited. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'17)*. https://doi.org/10.1145/3062341.3062373

Kensuke Kojima and Atsushi Igarashi. 2017. A hoare logic for GPU kernels. *ACM Trans. Comput. Logic* 18, 1, Article 3 (Feb. 2017), 43 pages. https://doi.org/10.1145/3001834

Ugo Dal Lago and Marco Gaboardi. 2011. Linear dependent types and relative completeness. In *Proceedings of the 26th IEEE Symposium on Logic in Computer Science (LICS'11)*. https://doi.org/10.1109/LICS.2011.22

Guodong Li and Ganesh Gopalakrishnan. 2010. SMT-based verification of GPU kernel functions. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE'10)*. https://doi.org/10.1145/1882291.1882320

Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic verification and test generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. https://doi.org/10.1145/2370036.2145844

Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2022. Provable GPU data-races in static race detection. In *PLACES: Electronic Proceedings in Theoretical Computer Science (EPTCS'22)*, Vol. 356. 36–45. https://doi.org/10.4204/EPTCS.356.4

Stefan K. Muller and Umut A. Acar. 2016. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'16)*. ACM, New York, NY, 71–82. https://doi.org/10.1145/2935764.2935793

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Competitive parallelism: Getting your priorities right. *Proc. ACM Program. Lang.* 2, ICFP, Article 95 (July 2018), 30 pages. https://doi.org/10.1145/3236790

Stefan K. Muller and Jan Hoffmann. 2021. Modeling and analyzing evaluation cost of CUDA kernels. *Proc. ACM Program. Lang.* 5, POPL, Article 25 (Jan. 2021), 31 pages. https://doi.org/10.1145/3434306

Stefan K. Muller and Jan Hoffmann. 2023. Modeling and Analyzing Evaluation Cost of CUDA Kernels. Retrieved from https://zenodo.org/records/10392393

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: Resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, 496–512. https://doi.org/10.1145/3192366.3192394

NVIDIA Corporation. 2019. *CUDA C Programming Guide v.10.1.168*.

Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A dynamic CUDA race detector. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, 390–403. https://doi.org/10.1145/3192366.3192368

Phillipe Pereira, Higo Albuquerque, Hendrio Marques, Isabela Silva, Celso Carvalho, Lucas Cordeiro, Vanessa Santos, and Ricardo Ferreira. 2016. Verifying CUDA programs using SMT-based context-bounded model checking. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC'16)*. ACM, New York, NY, 1648–1653. https://doi.org/10.1145/2851613.2851830

Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2017. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.* 2, POPL (2017). https://doi.org/10.1145/3158124

Dulma Rodriguez. 2010. A type system for amortised heap-space analysis of object-oriented programs. In *Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science, Algorithmic synthesis of reactive and discrete-continuous systems (AlgoSyn'10)*, Kai Bollue, Dominique Gückel, Ulrich Loup, Jacob Spönemann, and Melanie Winkler (Eds.). Verlagshaus Mainz, Aachen, Germany, 159.

Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. 2020. Sponge Examples: Energy-Latency Attacks on Neural Networks. Retrieved from https://arxiv.org/abs/2006.03463

Nimit Singhania. 2018. *Static Analysis for GPU Program Performance*. Ph.D. Dissertation. Computer and Information Science, University of Pennsylvania.

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A simple and scalable approach to bound analysis and amortized complexity analysis. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV'14)*. https://doi.org/10.1007/978-3-319-08867-9_50

Mingyuan Wu, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2019. Characterizing and detecting CUDA program bugs. Retrieved from http://arxiv.org/abs/1905.01833

Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, NY, 135–146. https://doi.org/10.1145/1941553.1941574

Received 8 June 2023; revised 18 October 2023; accepted 4 December 2023