Modeling and Analyzing Evaluation Cost of CUDA Kernels

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

General-purpose programming on GPUs (GPGPU) is becoming increasingly in vogue as applications such as machine learning and scientific computing demand high throughput in vector-parallel applications. NVIDIA's CUDA toolkit seeks to make GPGPU programming accessible by allowing programmers to write GPU functions, called kernels, in a small extension of C/C++. However, due to CUDA's complex execution model, the performance characteristics of CUDA kernels are difficult to predict, especially for novice programmers.

This paper introduces a novel quantitative program logic for CUDA kernels, which 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. 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 CUDA 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. 2021. Modeling and Analyzing Evaluation Cost of CUDA Kernels. *Proc. ACM Program. Lang.* 5, POPL, Article 25 (January 2021), 31 pages. https://doi.org/10.1145/3434306

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 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. 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 which 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.

Authors' addresses: Stefan K. Muller, Illinois Institute of Technology, USA, smuller2@iit.edu; Jan Hoffmann, Carnegie Mellon University, USA, jhoffmann@cmu.edu.

2475-1421/2021/1-ART25
https://doi.org/10.1145/3434306

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; Li and Gopalakrishnan 2010; Li et al. 2012; Singhania 2018] and dynamic [Boyer et al. 2008; 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 paper, 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 (e.g. [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 bugs 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*.

To automate the reasoning in our program logic, we have implemented the resource analysis tool RaCuda (**R**esource-**a**ware CUDA). If provided with the C implementation of a CUDA kernel and a resource metric, 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, and 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 doesn't 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. Although 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 (e.g., [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.

The contributions of this paper include:

- Two sets of operational cost semantics for a core calculus for CUDA kernels, 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 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.

The remainder of this paper is organized as follows. We begin with an introduction to the features of CUDA that will be relevant to this paper (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. Next, we describe in more detail our implementation of the analysis (Section 6) and evaluate it (Section 7). We conclude with a discussion of related work (Section 8).

```
__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.

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 paper 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 paper 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 SIMT (Single Instruction, Multiple Thread) execution. To reflect this, CUDA threads are organized into groups called *warps*[1]. The number of threads in

[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."

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, 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 $\texttt{addSub}_k$ 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 $\texttt{addSub}_k$ 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.

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, 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.

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

Fig. 2. Syntax of miniCUDA

3 THE 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 paper: 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 which 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 loses no 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 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, parameters to the kernel (these include arguments passed to the kernel function as well as CUDA parameters such as the block index and warp size), constants (of type int, bool and B) and a designated variable tid. 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. 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., $s_1;\ s_2$.

Our later results require certain "sanity checks" on code, namely that array indices be integers. We enforce these with a type system for miniCUDA, which is straightforward and therefore omitted for brevity. We write $\Sigma \vdash e : \tau$ to indicate that e has type τ under a signature Σ that gives the types of local variables, parameters, operators, functions and arrays. Statements do not have return values, but the judgment $\Sigma \vdash s$ is used to indicate that s is well-formed in that all of its subexpressions have the expected type.

Table 1. Resource constants and sample resource metrics. We use $F(n)$ to convert from integers to rationals.

Const.	Type	Param.	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	$\lambda\_.\ 0$	$\lambda n.\ F(n) - 1$	$\lambda\_.\ 0$	$\lambda n.\ F(n) - 1$

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 $M^{\mathsf{gread}}(i) \leq M^{\mathsf{gread}}(j)$, and similarly for M^{sread}, $M^{\mathsf{gwrite}}$and M^{swrite}.

The table 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.

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

$$\text{(OC:Var)}\quad \frac{}{\sigma; x \downarrow^{\mathcal{T}}_{M} (\sigma(x,t))_{t\in\mathcal{T}}; M^{\mathsf{var}}} \qquad \text{(EC:Op)}\quad \frac{\sigma; o_1 \downarrow^{\mathcal{T}}_{M} R_1; C_1 \qquad \sigma; o_2 \downarrow^{\mathcal{T}}_{M} R_2; C_2}{\sigma; o_1 \ \mathsf{op}\ o_2 \downarrow^{\mathcal{T}}_{M} (R_1(t) \ \mathsf{op}\ R_2(t))_{t\in\mathcal{T}}; C_1 + C_2 + M^{\mathsf{op}}}$$

$$\text{(EC:GArr)}\quad \frac{\sigma; o \downarrow^{\mathcal{T}}_{M} R; C}{\sigma; G[o] \downarrow^{\mathcal{T}}_{M} (\sigma(G, R(t)))_{t\in\mathcal{T}}; C + M^{\mathsf{gread}}(\mathrm{MemReads}(R))} \qquad \text{(OC:Tid)}\quad \frac{}{\sigma; \mathsf{tid} \downarrow^{\mathcal{T}}_{M} (Tid(t))_{t\in\mathcal{T}}; M^{\mathsf{var}}}$$

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

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

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

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

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

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

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

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

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

Fig. 3. Selected evaluation rules.

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 will 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 $R(t)$. 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

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

Selected evaluation rules for both judgments are presented in Figure 3. Some rules are omitted for space reasons. 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 $Tid(t)$ converts such an identifier to an integer thread ID. The domain of a store σ is $(Arrays \times \mathbb{Z}) \cup (LocalVars \times Threads)$. For an array A (regardless of whether it stored in global or shared memory), $\sigma(A, n)$ returns the n^{th} 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.

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]:

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

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. For assignment statements, the new state comes from updating the state with the new assignment. We write $\sigma[(x, t) \mapsto (v)_t \mid t \in \mathcal{T}]$ to indicate the state σ updated so that for all $t \in \mathcal{T}$, the binding (x, t) now maps to $(v)_t$. Array updates are written similarly.

Conditionals and while loops each have three rules. If a conditional evaluates to True for all threads (SC:IfT), 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), 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.

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 automated amortized resource analysis (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 6), 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 over the state of the store. 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, 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 $\sigma, \mathcal{T} \vDash P'$.

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 in order 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.

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 6, 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 $P' \Leftarrow P[x \leftarrow e]$ (resp. $Q' \Leftarrow Q[x \leftarrow e]$) 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 ${Q'}_X(\sigma) = Q_X(\sigma[(x,t) \mapsto R(t) \mid t \in \mathcal{T}])$

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 in Figure 4. These rules are similar to those of Figure 3, 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. Figure 4 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, we might start with $Q + C_1 + C_2$, have $Q + C_2$ left in the post-condition of s_1 and 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 the

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

(OQ:Var) $\dfrac{}{P \vdash_M x : M^{\mathsf{var}}}$ (OQ:Const) $\dfrac{}{P \vdash_M c : M^{\mathsf{const}}}$ (OQ:Param) $\dfrac{}{P \vdash_M p : M^{\mathsf{const}}}$ (OQ:Tid) $\dfrac{}{P \vdash_M \mathsf{tid} : M^{\mathsf{var}}}$ (EQ:Op) $\dfrac{P \vdash_M o_1 : C_1 \qquad P \vdash_M o_2 : C_2}{P \vdash_M o_1 \,\mathsf{op}\, o_2 : C_1 + C_2 + M^{\mathsf{op}}}$

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

(Q:Skip) $\dfrac{}{\vdash_M \{P;Q;X\}\ \mathsf{skip}\ \{P;Q;X\}}$

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

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

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

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

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

(Q:While1) $\dfrac{P \Rightarrow e\ \mathsf{unif} \qquad P \vdash_M e : C \qquad \vdash_M \{P \wedge e;Q;X\}\ s\ \{P;Q + M^{\mathsf{if}} + C;X\}}{\vdash_M \{P;Q + M^{\mathsf{if}} + C;X\}\ \mathsf{while}\ (e)\ s\ \{P \wedge \neg e;Q;X\}}$

(Q:While2) $\dfrac{P \vdash_M e : C \qquad X' = X \setminus W(s) \qquad \vdash_M \{P \wedge e;Q;X\}\ s\ \{P;Q + M^{\mathsf{if}} + M^{\mathsf{div}} + C;X\}}{\vdash_M \{P;Q + M^{\mathsf{if}} + M^{\mathsf{div}} + C;X\}\ \mathsf{while}\ (e)\ s\ \{P \wedge \neg e;Q;X'\}}$

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

(Q:VWrite2) $\dfrac{P \vdash_M e : C \qquad P \Leftarrow P'[x \leftarrow e]}{\vdash_M \{P;Q + M^{\mathsf{vwrite}} + C;X\}\ x \leftarrow e\ \{P';Q;X \setminus \{x\}\}}$

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

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

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

Fig. 4. Hoare logic rules for resource analysis.

conditional expression e does not vary across a warp: this is expressed with the premise $P \Rightarrow e$ 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, 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 ($M^{\mathsf{if}} + C$) 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" $M^{\mathsf{if}} + C$ 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, 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.

```
1                                                    L ≜ 2M^sread(1) + 2M^gwrite(5)
2                                                    {⊤; M^swrite(1) + M^gread(4) + h/2 L; {w, h, j}}
3  __global__ void addSub3 (int *A, int *B, int w, int h) {
4        __shared__ int As[32];
5    As[tid % 32] = A[tid];                          {⊤; h/2 L; {w, h, j}}
6    for (int j = 0; j < h; j += 2) {                {j < h; h−j/2 L; {w, h, j}}
7      B[j * w + tid] += As[tid];                    {j < h; M^sread(1) + M^gwrite(5) + h−j−2/2 L; {w, h, j}}
8      B[(j + 1) * w + tid] -= As[tid];              {j < h; h−j−2/2 L; {w, h, j}}
9    }
10                                                   {j ≥ h; 0; {w, h, j}}
   }
```

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

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 $\{P_2'; Q_2'; X\}$, 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 Derivation. Figure 5 steps through a derivation for the addSub3 kernel from Section 2, with the pre- and post-conditions interspersed in red. The code is simplified to more closely resemble miniCUDA. 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 $j = 0$). The initial constant potential is consumed by the write on line 5, which involves a global memory access with 4 separate reads (128 consecutive bytes with 32-byte reads[4] 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 6 how this information is encoded and used. On line 6, we establish the invariant of the loop body. On line 7, 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 7 and the rest on line 8. These require 5 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, in which the value of j will be decremented. After the loop, we conclude $j \geq h$ and have no remaining potential.

Soundness. We have proved 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 1. *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* $C' \leq C$.

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* $Q_X(\sigma) - C_s \geq Q'_{X'}(\sigma') \geq 0$.

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

All proofs are by induction on the derivation in the logic and are formalized in Coq. The case for while loops also includes an inner induction on the evaluation of the loop.

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 paper: 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, 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.

In order 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). 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. Given these two quantities, we can bound the execution time of a block assuming that the SM schedules warps *greedily*, that is, it issues instructions on as many ready warps as possible.

Theorem 2. *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 [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. □

We can, and will, use our analysis of the previous sections to independently calculate the work and span of a warp. Independently, we can compose the work and span of warps to obtain the work and span of a block: we sum over the work of the warps and take the maximum over the spans. This approach is not sound in general because warps of a block can synchronize with each other using the `__syncthreads()` built-in function[5], 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 taking the maximum span would result in an assumption that the span of the block includes 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 0-31 to perform their writes before proceeding.

Determining that, in the kernel above, each warp only performs one write, would require a sophisticated analysis that tracks costs separately for each thread: this is precisely what our analysis, to retain scalability, does not do. As a result, it is sound to simply compose the predicted spans of each warp in a block by taking the maximum. The remainder of this section will be devoted to proving this fact. In order to do so, we develop another cost semantics, this time a *parallel* semantics 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. 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 project 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 $((0,0)_{i \in Warps})$. We will use a shorthand for adding a cost onto a collection for a subset of warps:

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

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.

[5]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.

$$\text{(SC:Skip)}\quad \frac{}{\sigma; C; \mathsf{skip} \Downarrow^{\mathcal{T}}_{M} \sigma; C} \qquad \text{(SC:Sync)}\quad \frac{}{\sigma; (W, S); \mathsf{sync} \Downarrow^{\mathcal{T}}_{M} \sigma; (W, (\max S)_{Warps}) \oplus_{\mathcal{T}} M^{\mathsf{sync}}}$$

$$\text{(SC:Seq)}\quad \frac{\sigma; C; s_1 \Downarrow^{\mathcal{T}}_{M} \sigma_1; C' \qquad \sigma_1; C'; s_2 \Downarrow^{\mathcal{T}}_{M} \sigma_2; C''}{\sigma; C; s_1;\ s_2 \Downarrow^{\mathcal{T}}_{M} \sigma_2; C''} \qquad \text{(SC:VWrite)}\quad \frac{\sigma; C; e \downarrow^{\mathcal{T}}_{M} R; C'}{\sigma; C; x \leftarrow e \Downarrow^{\mathcal{T}}_{M} \sigma[(x, t) \mapsto R(t) \mid t \in \mathcal{T}]; C' \oplus_{\mathcal{T}} M^{\mathsf{vwrite}}}$$

$$\text{(SC:If)}\quad \frac{\begin{array}{c}\sigma; C; e \downarrow^{\mathcal{T}}_{M} R; C' \qquad \mathcal{T}_T = \{t \in \mathcal{T} \mid R_t\} \qquad \mathcal{T}_F = \{t \in \mathcal{T} \mid \neg R_t\} \\ \sigma; C'; s_1 \Downarrow^{\mathcal{T}_T}_{M} \sigma'; C'' \qquad \sigma'; C''; s_2 \Downarrow^{\mathcal{T}_F}_{M} \sigma''; C'''\end{array}}{\sigma; C; \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \Downarrow^{\mathcal{T}}_{M} \sigma''; C''' \oplus_{\mathcal{T}} M^{\mathsf{if}} \oplus_{\{w \mid \exists t_1 \in \mathcal{T}_T, t_2 \in \mathcal{T}_F\,.\, WarpOf(t_1) = WarpOf(t_2)\}} M^{\mathsf{div}}}$$

$$\text{(SC:While)}\quad \frac{\begin{array}{c}\sigma; C; e \downarrow^{\mathcal{T}}_{M} R; C' \qquad \sigma; C'; s \Downarrow^{\mathcal{T}_T}_{M} \sigma'; C'' \\ \mathcal{T}_T = \{t \in \mathcal{T} \mid R_t\} \qquad \sigma'; C''; \mathsf{while}\ (e)\ s \Downarrow^{\mathcal{T}_T}_{M} \sigma''; C'''\end{array}}{\sigma; C; \mathsf{while}\ (e)\ s \Downarrow^{\mathcal{T}}_{M} \sigma''; C''' \oplus_{\mathcal{T}} M^{\mathsf{if}} \oplus_{\{w \mid \exists t_1 \in \mathcal{T}_T, t_2 \in \mathcal{T} \setminus \mathcal{T}_T\,.\, WarpOf(t_1) = WarpOf(t_2)\}} M^{\mathsf{div}}}$$

Fig. 6. Selected rules for thread-level parallelism.

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:

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

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

Figure 6 gives selected rules for the cost semantics for statements, for which the judgment is

$$\sigma; C; s \Downarrow^{\mathcal{T}}_{M} \sigma'; C'$$

meaning that, on a set of threads $\mathcal{T}$, with heap σ, the statement s results in final heap σ' and if the cost collection is initially C, the cost extended with the cost of executing s is C'. The judgments for operands and expressions are similar; as with the lock-step cost semantics, they return a result family. The rule SC:Sync handles taking the maximum at synchronization points, and rules SC:If and SC:While add the cost of divergence only to warps that actually diverge. Otherwise, the rules (including rules omitted for space reasons) resemble those of the lock-step semantics.

We now show that the analysis of Section 4, when applied on work and span independently, soundly approximates the parallel cost semantics of Figure 6. 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 2 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 3 is the equivalent result for statements. Both proofs are straightforward inductions and are formalized in Coq.

Lemma 2. *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) $W(C') - W(C) \leq C_w \, |\{ \mathit{WarpOf}(t) \mid t \in \mathcal{T}\}|$
(4) $S(C') - S(C) \leq C_s$

Lemma 3. *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. Note that $W(\emptyset) = S(\emptyset) = 0$, so by Lemma 3, 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. □

6 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[6] 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 4, 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 8 for comparisons to related work). Still, for completeness, we briefly describe our approach here.

The abstraction domain is a pair $(C, \mathcal{M})$. 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 $\mathcal{M}_{\mathsf{tid}}(x)$ 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 $\mathcal{M}_{\mathsf{const}}(x)$ 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 $\mathcal{M}_{\mathsf{tid}}(x) = (0, 0, 0)$, then the value of x is guaranteed to be constant across threads. As another example, if $\mathcal{M}_{\mathsf{tid}}(x) = (1, 1, 1)$, then $x = \mathsf{tid} + c$, 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.

$$\begin{array}{rcl} \text{MemReads}(R) & \triangleq & \left|\left\{\left\lceil \frac{i}{m} \right\rceil \mid (i)_t \in R\right\}\right| \\ \text{Conflicts}(R) & \triangleq & \max\limits_{j \in [0, B-1]} \{a \equiv j \mod B \mid a \in Cod(R)\} \end{array}$$

Theorem 4 formalizes and proves the soundness of a bound on MemReads(x) given abstract information about x.

Theorem 4. *If* $\mathcal{M}_{\mathsf{tid}}(x) = k$ *and* $C \implies \mathsf{tid} \in [t, t']$ *and* $\sigma, \mathcal{T} \vDash (C, \mathcal{M})$ *and* $\sigma; x \downarrow_M^{\mathcal{T}} R; C$ *then* $\mathit{MemReads}(R) \leq \left\lceil \frac{k(t'-t)}{m} \right\rceil + 1$.

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

Proof. By the definition of $\sigma, \mathcal{T} \vDash (C, \mathcal{M})$, 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

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

□

Theorem 5 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 4, a stand-alone result about modular arithmetic.

Theorem 5. *If* $\mathcal{M}_{\mathsf{tid}}(x) = k > 0$ *and* $C \Rightarrow \mathsf{tid} \in [t, t']$ *and* $\sigma, \mathcal{T} \vDash (C, \mathcal{M})$ *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 $\sigma, \mathcal{T} \vDash (C, \mathcal{M})$, we have $Tid(t_0) \in [t, t']$. 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 $\left\lceil \frac{t'-t}{|Dom(R')|} \right\rceil$. The result follows from Lemma 4. □

Lemma 4. *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,$\{i \cdot a \mod n \mid i \in \{k, \ldots, k+m-1\}\} = A$. □

As an example of how the abstraction information is tracked and used in the resource analysis, we return to the code example in Figure 5. Figure 7 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 C 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 4, we can bound MemReads(`i`) by $\lceil \frac{32}{m} \rceil + 1$. Note that both t and t' in the statement of Theorem 4 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 5, we can bound Conflicts(`threadIdx.x`) by $\left\lceil \frac{32}{\min(32, \frac{32}{\gcd(1,32)})} \right\rceil = 1$.

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

```
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 }
```

$\mathcal{M}_{\mathsf{tid}}(\mathtt{blockDim.x}) = 32, \mathcal{M}_{\mathsf{tid}}(\mathtt{threadIdx.x}) \leq 32$

$\mathcal{M}_{\mathsf{tid}}(\mathtt{i}) = (1, 0, 0), \mathcal{M}_{\mathsf{const}}(\mathtt{i}) = 32\mathtt{blockIdx.x}$

$\mathcal{M}_{\mathsf{tid}}(\mathtt{j}) = (0, 0, 0), \mathcal{M}_{\mathsf{const}}(\mathtt{j}) = \top$

$\mathcal{M}_{\mathsf{tid}}(\mathtt{temp0}) = (1, 0, 0), \mathcal{M}_{\mathsf{const}}(\mathtt{temp0}) = \mathtt{j} * \mathtt{w}$

$\mathcal{M}_{\mathsf{tid}}(\mathtt{temp1}) = (1, 0, 0), \mathcal{M}_{\mathsf{const}}(\mathtt{temp1}) = (\mathtt{j} + 1) * \mathtt{w}$

Fig. 7. A sample abstract interpretation.

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 5). 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].

7 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 SDK, modified from such kernels by us, or written entirely by us). 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 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

Table 2. The benchmark suite used for our experiments. **SDK** = benchmarks distributed with CUDA SDK. **SDK*** = benchmarks derived from SDK by authors. **Us** = benchmarks written by authors.

Benchmark	Source	LoC	Params.	Block
matMul	SDK	26	N	32×32
matMulBad	SDK*		N	32×32
matMulTrans	SDK*	26	N	32×32
mandelbrot	SDK	78	N	32×1
vectorAdd	SDK	5	N	256×1
reduceN	SDK	14–18	N	256×1
histogram256	SDK	19	N	64×1
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	$w \times 1$

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 which deliberately introduces a number of performance bugs (matMulBad), and one (matMulTrans) which 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 4 in our benchmark suite; later iterations use advanced features of CUDA which we do not currently support. 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 also include the original versions. Finally, we include the examples from Section 2 (addSubN). We analyzed each benchmark under the four resource metrics defined in Table 1.

7.1 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 will 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 3 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. In order 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.

7.2 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

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

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

Table 4. Analysis time, inferred bound and average error for the "conflicts" metric. 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.

			Error			
Benchmark	Time (s)	Predicted Bound	$32N$	$32N+1$	Random	Average
matMul	0.11	$31(31+N)\lceil \frac{N}{32} \rceil$	—			—
matMulBad	8.13	0	0			0
matMulTrans	0.12	$1023\frac{31+N}{32}\lceil \frac{N}{32} \rceil$	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	$23715\lceil \frac{N}{32} \rceil$	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

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

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

Table 6. Analysis time, inferred bound and average error for the "divwarps" metric. An entry of "n/b" indicates that our analysis was unable to determine a bound.

			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

Table 7. Analysis time, inferred bound and average error for the "steps" metric. An entry of "n/b" indicates that our analysis was unable to determine a bound.

			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

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.

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 8 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 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[7]. In the worst case, all bytes handled by a warp map to the same memory bank, resulting in an 8-way conflict at each access (32 bins are accessed, but as there are only 256 bins, only $256/32 = 8$ map to each memory

[7]Of course, these would also result in data races in the absence of synchronization (which is present in the benchmark as originally written), but such synchronization would also be likely to result in performance impacts when such conflicts occur; for illustration purposes, we disable the synchronization so that this performance impact shows up as a bank conflict.

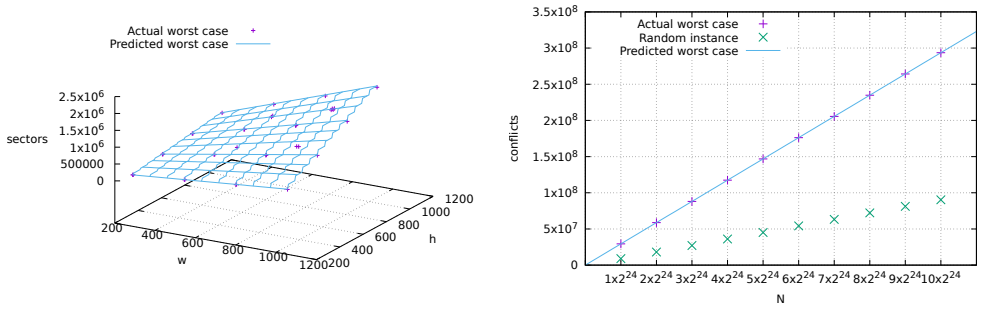

Fig. 8. Our inferred cost estimates (blue line) vs. actual worst-case costs (purple crosses) for various inputs. **Left:** addSub1, sectors; **Right:** histogram, conflicts (also includes a random input, green crosses)

bank). On the other hand, 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.

8 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 automatic amortized analysis (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.

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 [Li and Gopalakrishnan 2010; Pereira et al. 2016; 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 don't give examples of kernels with non-optimal implementations, and don't 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 in order to perform its analysis. 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 $\mathcal{M}_{\mathsf{tid}}(x)$ 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 two 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 paper 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.

9 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 8).

The "steps" metric of Section 7 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 paper, 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 would like to 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.

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 1801369, CAREER Award 1845514, and SHF Awards 1812876 and 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.

REFERENCES

Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and German Puebla. 2011. Cost Analysis of Concurrent OO Programs. In *9th Asian Symp. on Prog. Langs. 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

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*. 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 *Third 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 *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, USA, 467–478. https://doi.org/10.1145/2737924.2737955

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2012. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th Int. Conf. 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 *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 Transactions on Computing* 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 2017)*. ACM, New York, NY, USA, 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 *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 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 *36th ACM Symp. on Principles of Prog. Langs. (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 Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*. ACM, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/1964179.1964184

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (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 *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 *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 *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*. 185–197. https://doi.org/10.1145/640128.604148

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional Recurrence Analysis Revisited. In *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 *26th IEEE Symp. 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 *International Symposium on the Foundations of Software Engineering (FSE) (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 *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/2370036.2145844

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, USA, 71–82. https://doi.org/10.1145/2935764.2935793

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 2018)*. ACM, New York, NY, USA, 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 2018)*. ACM, New York, NY, USA, 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, USA, 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

Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. 2020. Sponge Examples: Energy-Latency Attacks on Neural Networks. arXiv:cs.LG/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 *Computer Aided Verification - 26th Int. Conf. (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. *CoRR* abs/1905.01833 (2019). arXiv:1905.01833 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, USA, 135–146. https://doi.org/10.1145/1941553.1941574