

1 Modeling and Analyzing Evaluation Cost of CUDA Kernels

2 STEFAN K. MULLER, Illinois Institute of Technology, USA

3 JAN HOFFMANN, Carnegie Mellon University, USA

4 General-purpose programming on GPUs (GPGPU) is becoming increasingly in vogue as applications such as
 5 machine learning and scientific computing demand high throughput in vector-parallel applications. NVIDIA's
 6 CUDA toolkit seeks to make GPGPU programming accessible by allowing programmers to write GPU
 7 functions, called kernels, in a small extension of C/C++. However, due to CUDA's complex execution model,
 8 the performance characteristics of CUDA kernels are difficult to predict, especially for novice programmers.
 9

10 This paper introduces a novel quantitative program logic for CUDA kernels, which allows programmers to
 11 reason about both functional correctness and resource usage of CUDA kernels, paying particular attention to a
 12 set of common but CUDA-specific performance bottlenecks. The logic is proved sound with respect to a
 13 novel operational cost semantics for CUDA kernels. The semantics, logic and soundness proofs are formalized
 14 in Coq. An inference algorithm based on LP solving automatically synthesizes symbolic resource bounds by
 15 generating derivations in the logic. This algorithm is the basis of RACUDA, an end-to-end resource-analysis
 16 tool for kernels, which has been implemented using an existing resource-analysis tool for imperative programs.
 17 An experimental evaluation on a suite of CUDA benchmarks shows that the analysis is effective in aiding the
 18 detection of performance bugs in CUDA kernels.

19 **ACM Reference Format:**

20 Stefan K. Muller and Jan Hoffmann. 2021. Modeling and Analyzing Evaluation Cost of CUDA Kernels. *Proc.*
 21 *ACM Program. Lang.* 1, POPL, Article 1 (January 2021), 31 pages.

22 **1 INTRODUCTION**

23 Many of today's computational problems, such as training a neural network or processing images,
 24 are massively data-parallel: many steps in these algorithms involve applying similar arithmetic or
 25 logical transformations to a large, possibly multi-dimensional, vector of data. Such algorithms are
 26 naturally suitable for execution on Graphics Processing Units (GPUs), which consist of thousands
 27 of processing units designed for vector operations. Because of this synergy, general-purpose GPU
 28 (GPGPU) programming has become increasingly mainstream. With the rise of GPGPU programming
 29 has come tools and languages designed to enable this form of programming. Possibly the best-
 30 known such tool is CUDA, a platform for enabling general-purpose programs to run on NVIDIA
 31 GPUs. Among other features, CUDA provides an extension to C which allows programmers to
 32 write specialized functions, called *kernels*, for execution on the GPU. The language for writing
 33 kernels, called CUDA C or just CUDA, is very similar to C, enabling easy adoption by developers.
 34

35 Nevertheless, writing a kernel that executes efficiently on a GPU is not as simple as writing a
 36 C function: small changes to a kernel, which might be inconsequential for sequential CPU code,
 37 can have drastic impact on its performance. The CUDA C Programming Guide [29] lists three
 38 particularly pervasive performance bottlenecks to avoid: *divergent warps*, *uncoalesced memory*
 39 *accesses*, and *shared memory bank conflicts*. Divergent warps result from CUDA's execution model:
 40 a group of threads (often 32 threads, referred to as a *warp*) execute the same instruction on possibly
 41 different data. C functions, however, can perform arbitrary branching that can cause different
 42 threads of a warp to *diverge*, i.e., take different branches. CUDA is able to compile such code and
 43 execute it on a GPU, but at a fairly steep performance cost, as the two branches must be executed

44
 45 Authors' addresses: Stefan K. Muller, Illinois Institute of Technology, USA, smuller2@iit.edu; Jan Hoffmann, Carnegie
 46 Mellon University, USA, jhoffmann@cmu.edu.

47 2021. 2475-1421/2021/1-ART1 \$15.00

48 <https://doi.org/>

sequentially. Even if a conditional only has one branch, there is nontrivial overhead associated with divergence [3]. 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 [2, 25, 26, 34] and dynamic [6, 36] 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 [17]. 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 [33] 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) [18, 21] 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. [23]) 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. The cost semantics, the analysis, and the soundness proof 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

99 resource metric, RACUDA automatically derives a symbolic upper bound on the execution cost of
100 the kernel as specified by the metric. We have implemented, RACUDA on top of Absynth [9, 28], a
101 resource analysis tool for imperative programs.

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

107 The features of the analysis described so far are sufficient to analyze properties of a kernel such as
108 numbers of divergent warps or memory accesses. Analyzing the execution time of a kernel requires
109 more care because execution time doesn't compose in a straightforward way: threads are scheduled
110 onto available processors by a (deliberately) underspecified scheduling algorithm, which exploits
111 *thread-level parallelism* of kernels to hide the latency of operations such as memory accesses. While
112 we do not aim to develop a precise analysis of kernel execution time in this work (even for CPUs
113 where such Worst-case Execution Time analyses are well-studied, this is a separate and rich area of
114 research). However, we do take steps toward such an analysis by showing how our analysis can be
115 used to compute the *work* and *span* of kernels, two metrics derived from the literature on parallel
116 scheduling theory (e.g., [4, 5, 7, 13, 27]) that can be used to abstractly approximate the running
117 time of parallel algorithms.

118 The contributions of this paper include:

- 119 • Two sets of operational cost semantics for a core calculus for CUDA kernels, one for the
120 *lock-step* evaluation of individual warps and one for the *parallel* evaluation of many warps.
121 Both formalize the execution of kernels on a GPU under a given resource metric
- 122 • A novel Hoare-style program logic for miniCUDA, including both qualitative and quantitative
123 properties
- 124 • A Coq formalization of the cost semantics and soundness proofs of the program logic
- 125 • An analysis tool RACUDA that can parse kernels written in a sizable subset of CUDA C and
126 analyze them with respect to resource metrics such as number of bank conflicts and number
127 of divergent warps
- 128 • An empirical evaluation of our analysis tools on a suite of CUDA kernels.

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

138 2 A BRIEF INTRODUCTION TO CUDA

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

143 *Kernels and Threads.* A kernel is invoked on the GPU by calling it much like a regular function
144 with additional arguments specifying the number and layout of threads on which it should run. The

```

148     __global__ void addSubk (int *A, int *B, int w, int h) { addSk }
149     addS0 ≡
150         for (int i = 0; i < w; i++) {
151             int j = blockIdx.x * blockDim.x
152                 + threadIdx.x;
153             if (j % 2 == 0) {
154                 B[j * w + i] += A[i];
155             } else {
156                 B[j * w + i] -= A[i];
157             }
158         }
159
160     addS2 ≡
161         int i = blockIdx.x * blockDim.x
162             + threadIdx.x;
163         for (int j = 0; j < h; j += 2) {
164             B[j * w + i] += A[i];
165             B[(j + 1) * w + i] -= A[i];
166         }
167
168     addS1 ≡
169         for (int i = 0; i < w; i++) {
170             int j = blockIdx.x * blockDim.x
171                 + threadIdx.x;
172             B[2 * j * w + i] += A[i];
173             B[(2 * j + 1) * w + i] -= A[i];
174         }
175
176     addS3 ≡
177         __shared__ int As[blockDim.x];
178         int i = blockIdx.x * blockDim.x
179             + threadIdx.x;
180         As[threadIdx.x] = A[i];
181         for (int j = 0; j < h; j += 2) {
182             B[j * w + i] += As[i];
183             B[(j + 1) * w + i] -= As[i];
184         }
185
186
187
188
189
190
191
192
193
194
195
196

```

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

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 identifiers `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 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.

¹ Warp refers to the set of parallel threads stretched across a loom during weaving; according to the CUDA programming manual [29], “The term *warp* originates from weaving, the first parallel thread technology.”

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

201 Consider first the function addSub_0 that is given by addS_0 . The `for` loop iterates over the columns
 202 of the matrix, and each row is processed in parallel by separate threads. There is no need to iterate
 203 over the rows, because the main program instantiates the kernel for each row.

204 The implementation of addSub_0 contains a number of performance bugs. First, the conditional
 205 diverges at every iteration of the loop. The reason is that every warp contains thread identifiers
 206 (`threadIdx`) that result in both odd and even values for the variable j . A straightforward way to fix
 207 this bug is to remove the conditional, unroll the loop and perform both the addition of an even
 208 row and the subtraction of an odd row in one loop iteration. The resulting code, shown in addSub_1 ,
 209 does not have more parallelism than the original—the addition and subtraction are still performed
 210 sequentially—but will perform better because it greatly reduces the overhead of branching.

211 *Memory Accesses.* The next performance bottleneck we discuss relates to the way CUDA handles
 212 *global* memory accesses. CUDA warps can access up to 128 consecutive bytes of such memory at
 213 once. When threads in a warp access memory, such as the accesses to arrays A and B in the example,
 214 CUDA attempts to coalesce these accesses together into as few separate accesses as possible. If
 215 a warp accesses four consecutive 32-bit elements of an array, the memory throughput of that
 216 instruction is four times higher than if it performs four non-consecutive reads.
 217

218 The execution of the function addSub_1 is unable to coalesce accesses to B because, assuming w is
 219 larger than 32 and the arrays are stored in row-major order, no two threads within a warp access
 220 memory within 128 bytes of each other. This is fixed by instead iterating over the rows of the
 221 matrix and handling the columns in parallel. This way, all of the memory accesses by a warp are
 222 consecutive (e.g., threads 0 through 31 might access $A[0]$ through $A[31]$ and $B[w]$ through $B[w+63]$).
 223 The updated code is shown in the function body addSub_2 .

224 *Shared Memory.* In all of the kernels discussed so far, the arrays A and B reside in *global* memory,
 225 which is stored on the GPU and visible to all threads. CUDA also provides a separate *shared* memory
 226 space, which is shared only by threads within a block. Shared memory has a lower latency than
 227 global memory so we can, for example, use it to store the values of A rather than access global
 228 memory every time. In the function addsub_3 , we declare a shared array As and copy values of A
 229 into As before their first use.

230 Some care must be taken to ensure that the code of addsub_3 is performant because of how shared
 231 memory is accessed. Shared memory consists of a number, generally 32, of separate banks. Separate
 232 banks may be accessed concurrently, but multiple concurrent accesses to separate addresses in the
 233 same bank are serialized. It is thus important to avoid “bank conflicts”. Most GPUs ensure that 32
 234 consecutive 32-bit memory reads will not result in any bank conflicts. However, if a block accesses
 235 a shared array at a stride other than 1, bank conflicts can accumulate.

237 3 THE MINICUDA CORE CALCULUS

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

242 In designing the miniCUDA calculus, we have made a number of simplifying assumptions which
 243 make the presentation cleaner and more straightforward. One notable simplification is that a mini-
 244 CUDA program consists of a single kernel, without its function header. In addition, we collapse the
 245

```

246     Types       $\tau ::= \text{int} \mid \text{bool} \mid B \mid \text{arr}(\tau)$ 
247     Operands    $o ::= x \mid p \mid c \mid \text{tid}$ 
248     Arrays       $A ::= G \mid S$ 
249     Expressions  $e ::= o \mid o \text{ op } o \mid A[o]$ 
250     Statements    $s ::= \text{skip} \mid s; s \mid x \leftarrow e \mid A[o] \leftarrow e \mid \text{if } e \text{ then } s \text{ else } s \mid \text{while } (e) s$ 
251
252
253
254

```

Fig. 2. Syntax of miniCUDA

255 structure of thread and block indices into a single thread ID, denoted `tid`. This loses no generality as
 256 a three-dimensional thread index can be converted in a straightforward way to a one-dimensional
 257 thread ID, given the block dimension. The resource analysis will be parametric over the block index
 258 and other parameters, and will estimate the maximum resource usage of any warp in any block.

259 *Syntax.* The syntax of the miniCUDA calculus is presented in Figure 2. Two types of data are
 260 particularly important to the evaluation and cost analysis of the calculus: integers are used for both
 261 array indices (which determine the costs of memory accesses) and loop bounds (which are crucial
 262 for estimating the cost of loops), and booleans are used in conditionals. All other base types (e.g.,
 263 float, string) are represented by an abstract base type B . We also include arrays of any type.
 264

265 The terms of the calculus are divided into statements, which may affect control flow or the state
 266 of memory, and expressions, which do not have effects. We further distinguish *operands*, which
 267 consist of thread-local variables, parameters to the kernel (these include arguments passed to the
 268 kernel function as well as CUDA parameters such as the block index and warp size), constants (of
 269 type `int`, `bool` and B) and a designated variable `tid`. Additional expressions include $o_1 \text{ op } o_2$, which
 270 stands for an arbitrary binary operation, and array accesses $A[o]$. The metavariable A stands for
 271 a generic array. When relevant, we use metavariables that indicate whether the array is stored
 272 in (G)lobal or (S)hared memory. Note that subexpressions of expressions are limited to operands;
 273 more complex expressions must be broken down into binary ones by binding intermediate results
 274 to variables. This restriction simplifies reasoning about expressions without limiting expressivity.

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

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

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

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}	$\mathbb{N} \rightarrow \mathbb{Q}$	# of seq. reads	$\lambda n. F(n)$	$\lambda_. 0$	$\lambda_. 0$	$\lambda n. F(n)$
M^{sread}	$\mathbb{N} \rightarrow \mathbb{Q}$	# 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}	$\mathbb{N} \rightarrow \mathbb{Q}$	# of seq. reads	$\lambda n. F(n)$	$\lambda_. 0$	$\lambda_. 0$	$\lambda n. F(n)$
M^{swrite}	$\mathbb{N} \rightarrow \mathbb{Q}$	# of conflicts	$\lambda_. 0$	$\lambda n. F(n) - 1$	$\lambda_. 0$	$\lambda n. F(n) - 1$

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

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

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 requests².
- divwarps: Counts the number of times a warp diverges.
- steps: Counts the total number of evaluation steps.

Lock-step Operational Semantics. We now define an operational semantics for evaluating mini-CUDA kernels that also tracks the cost of evaluation given a resource metric. We use this semantic model as the basis for proving the soundness of our resource analysis in the next section. The operational semantics evaluates an expression or statement over an entire warp at a time to produce a result and the cost of evaluation. Because it only evaluates one warp and threads of a warp execute in lock-step, we refer to this as the “lock-step” semantics to differentiate it from the “parallel” semantics we 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.

²the term “sectors” comes from the NVIDIA profiling tools’ terminology for this metric

$\frac{(\text{OC:VAR})}{\sigma; x \downarrow_M^{\mathcal{T}} (\sigma(x, t))_{t \in \mathcal{T}}; M^{\text{var}}}$	$\frac{(\text{EC:OP})}{\sigma; o_1 \downarrow_M^{\mathcal{T}} R_1; C_1 \quad \sigma; o_2 \downarrow_M^{\mathcal{T}} R_2; C_2}{\sigma; o_1 \text{ op } o_2 \downarrow_M^{\mathcal{T}} (R_1(t) \text{ op } R_2(t))_{t \in \mathcal{T}}; C_1 + C_2 + M^{\text{op}}}$
$\frac{(\text{EC:GARR})}{\sigma; o \downarrow_M^{\mathcal{T}} R; C}$	$\frac{(\text{OC:TID})}{\sigma; \text{tid} \downarrow_M^{\mathcal{T}} (\text{Tid}(t))_{t \in \mathcal{T}}; M^{\text{var}}}$
$\frac{(\text{EC:SARR})}{\sigma; S[o] \downarrow_M^{\mathcal{T}} (\sigma(S, R(t)))_{t \in \mathcal{T}}; C + M^{\text{sread}}(\text{MemReads}(R))}$	
$\frac{(\text{SC:GWRITE})}{\sigma; o \downarrow_M^{\mathcal{T}} R_1; C_1 \quad \sigma; e \downarrow_M^{\mathcal{T}} R_2; C_2}{\sigma; G[o] \leftarrow e \Downarrow_M^{\mathcal{T}} \sigma[(G, R_1(t)) \mapsto R_2(t) \mid t \in \mathcal{T}]; C_1 + C_2 + M^{\text{gwrite}}(\text{MemReads}(R_1))}$	
$\frac{(\text{SC:SWRITE})}{\sigma; o \downarrow_M^{\mathcal{T}} R_1; C_1 \quad \sigma; e \downarrow_M^{\mathcal{T}} R_2; C_2}{\sigma; S[o] \leftarrow e \Downarrow_M^{\mathcal{T}} \sigma[(S, R_1(t)) \mapsto R_2(t) \mid t \in \mathcal{T}]; C_1 + C_2 + M^{\text{swrite}}(\text{Conflicts}(R_1))}$	$\frac{(\text{SC:SKIP})}{\sigma; \text{skip} \Downarrow_M^{\mathcal{T}} \sigma; 0}$
$\frac{(\text{SC:SEQ})}{\sigma; s_1 \Downarrow_M^{\mathcal{T}} \sigma_1; C_1 \quad \sigma_1; s_2 \Downarrow_M^{\mathcal{T}} \sigma_2; C_2}{\sigma; s_1; s_2 \Downarrow_M^{\mathcal{T}} \sigma_2; C_1 + C_2}$	$\frac{(\text{SC:VWRITE})}{\sigma; e \downarrow_M^{\mathcal{T}} R; C}{\sigma; x \leftarrow e \Downarrow_M^{\mathcal{T}} \sigma[(x, t) \mapsto R(t) \mid t \in \mathcal{T}]; C + M^{\text{vwrite}}}$
$\frac{(\text{SC:IFT})}{\sigma; e \downarrow_M^{\mathcal{T}} (\text{True})_{t \in \mathcal{T}}; C_1 \quad \sigma; s_1 \Downarrow_M^{\mathcal{T}} \sigma_1; C_2}{\sigma; \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow_M^{\mathcal{T}} \sigma_1; C_1 + M^{\text{if}} + C_2}$	$\frac{(\text{SC:IFD})}{\mathcal{T}_T = \{t \in \mathcal{T} \mid R(t)\} \neq \emptyset \quad \mathcal{T}_F = \{t \in \mathcal{T} \mid \neg R(t)\} \neq \emptyset}{\sigma; e \downarrow_M^{\mathcal{T}} R; C_1 \quad \sigma; s_1 \Downarrow_M^{\mathcal{T}_T} \sigma_1; C_2 \quad \sigma_1; s_2 \Downarrow_M^{\mathcal{T}_F} \sigma_2; C_3}{\sigma; \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow_M^{\mathcal{T}} \sigma_2; C_1 + M^{\text{if}} + C_2 + C_3 + M^{\text{div}}}$
$\frac{(\text{SC:WHILEALL})}{\sigma; e \downarrow_M^{\mathcal{T}} (\text{True})_{t \in \mathcal{T}}; C_1 \quad \sigma; s \Downarrow_M^{\mathcal{T}} \sigma_1; C_2 \quad \sigma_1; \text{while } (e) s \Downarrow_M^{\mathcal{T}} \sigma_2; C_3}{\sigma; \text{while } (e) s \Downarrow_M^{\mathcal{T}} \sigma_2; C_1 + C_2 + M^{\text{if}} + C_3}$	
$\frac{(\text{SC:WHILESOME})}{\sigma; e \downarrow_M^{\mathcal{T}} R; C_1 \quad \emptyset \neq \mathcal{T}_T \neq \mathcal{T} \quad \sigma; s \Downarrow_M^{\mathcal{T}_T} \sigma_1; C_2}{\mathcal{T}_T = \{t \in \mathcal{T} \mid R(t)\} \quad \sigma_1; \text{while } (e) s \Downarrow_M^{\mathcal{T}} \sigma_2; C_3}{\sigma; \text{while } (e) s \Downarrow_M^{\mathcal{T}} \sigma_2; C_1 + C_2 + M^{\text{if}} + M^{\text{div}} + C_3}$	$\frac{(\text{SC:WHILENONE})}{\sigma; e \downarrow_M^{\mathcal{T}} (\text{False})_{t \in \mathcal{T}}; C}{\sigma; \text{while } (e) s \Downarrow_M^{\mathcal{T}} \sigma; C + M^{\text{if}}}$

Fig. 3. Selected evaluation rules.

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:

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

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

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

395 Selected evaluation rules for both judgments are presented in Figure 3. Some rules are omitted
 396 for space reasons. We now discuss additional representation details and discuss some rules
 397 in more detail. As suggested above, the elements of \mathcal{T} are abstract thread identifiers t . The
 398 function $Tid(t)$ converts such an identifier to an integer thread ID. The domain of a store σ
 399 is $(\text{Arrays} \times \mathbb{Z}) \cup (\text{LocalVars} \times \text{Threads})$. For an array A (regardless of whether it stored in global or
 400 shared memory), $\sigma(A, n)$ returns the n^{th} element of A . Note that, for simplicity of presentation, we
 401 assume that out-of-bounds indices (including negative indices) map to some default value. For a
 402 local variable x , $\sigma(x, t)$ returns the value of x for thread t .

403 The evaluation of expressions is relatively straightforward: we simply evaluate subexpressions in
 404 parallel and combine appropriately, but with careful accounting of the costs. The cost of execution
 405 is generally obtained by summing the costs of evaluating subexpressions with the cost of the head
 406 operation given by the resource metric M . For example, Rule EC:OP evaluates the two operands
 407 and combines the results using the concrete binary operation represented by op. The cost is the
 408 cost of the two subexpressions, C_1 and C_2 , plus M^{op} . Array accesses evaluate the operand to a set
 409 of indices and read the value from memory at each index. The cost of these operations depends
 410 on $\text{MemReads}(R)$ or $\text{Conflicts}(R)$, where R is the set of indices. Recall that these functions give the
 411 number of global memory reads necessary to access the memory locations specified by R , and the
 412 number of bank conflicts resulting from simultaneously accessing the memory locations specified
 413 by R , respectively. As discussed before, we leave these functions as parameters because their exact
 414 definitions can change across versions of CUDA and hardware implementations. As examples
 415 of these functions, we give definitions consistent with common specifications in modern CUDA
 416 implementations [29]:

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

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

424 Statement evaluation is slightly more complex, as statements can update the state of mem-
 425 ory and also impact control flow: the former is represented by updating the store σ and the
 426 latter is represented by changing the thread set \mathcal{T} when evaluating subexpressions. For assign-
 427 ment statements, the new state comes from updating the state with the new assignment. We
 428 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)
 429 now maps to $(v)_t$. Array updates are written similarly.

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

442 4 QUANTITATIVE PROGRAM LOGIC

443 In this section, we present declarative rules for a Hoare-style logic that can be used to reason
 444 about the resource usage of a warp of a miniCUDA kernel. The analysis for resource usage is
 445 based on the ideas of automated amortized resource analysis (AARA) [18, 21]. The key idea of this
 446 analysis is to assign a non-negative numerical potential to states of computation. This potential
 447 must be sufficient to cover the cost of the following step and the potential of the next state. For
 448 imperative programs, the potential is generally a function of the values of local variables. Rules
 449 of a *quantitative Hoare logic* specify how this *potential function* changes during the execution of a
 450 statement. A derivation in the quantitative Hoare logic then builds a set of constraints on potential
 451 functions at every program point. In an implementation (Section 6), these constraints are converted
 452 to a linear program and solved with an LP solver.

453 As an example, consider the statement `for (int i = N; i >= 0; i--) { f(); }` and suppose we wish
 454 to bound the number of calls to `f`. This corresponds to a resource metric in which the cost of a
 455 function call is 1 and all other operations are free. The potential function at each point should be
 456 a function of the value of `i`: it will turn out to be the case that the correct solution is to set the
 457 potential to `i+1` in the body of the loop before the call to `f` and to `i` after the call. This difference
 458 “pays for” for the cost of 1 for the function call. It also sets up the proper potential for the next loop
 459 iteration: when `i` is decremented following the loop, the potential once again becomes `i+1`.

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

470 *Pre- and Post-Conditions.* Conditions of our logic have the form $\{P; Q; X\}$ and consist of the *logical*
 471 *condition* P and the *potential function* Q (both of which we describe below) as well as a set X of
 472 variables whose values are uniform across the warp and therefore can be used as potential-carrying
 473 variables as described above. We write $\sigma, \mathcal{T} \vdash X$ to mean that for all $x \in X$ and all $t_1, t_2 \in \mathcal{T}$,
 474 we have $\sigma(x, t_1) = \sigma(x, t_2)$. The *logical condition* P is a reasonably standard Hoare logic pre- or
 475 post-condition and contains logical propositions over the state of the store. We write $\sigma, \mathcal{T} \models P$ to
 476 indicate that the condition P is true under the store σ and values $t \in \mathcal{T}$ for the thread identifier.
 477 If either the store or the set of threads is not relevant in a particular context, we may use the
 478 shorthand $\sigma \models P$ to mean that there exists some \mathcal{T} such that $\sigma, \mathcal{T} \models P$ or the shorthand $\mathcal{T} \models P$ to
 479 mean that there exists some σ such that $\sigma, \mathcal{T} \models P$ ³. We write $P \Rightarrow P'$ to mean that P implies P' :
 480 that is, for all σ, \mathcal{T} such that $\sigma, \mathcal{T} \models P$, it is the case that $\sigma, \mathcal{T} \models P'$.

481 The second component of the conditions is a *potential function* Q , a mapping from stores and
 482 sets of variables X as described above to non-negative rational potentials. We use the potential
 483 function to track potential through a kernel in order to analyze resource usage. If $\sigma, \mathcal{T} \vdash X$,
 484 then $Q_X(\sigma)$ refers to the potential of σ under function Q , taking into account only the variables
 485 in X . Formally, we require (as a property of potential functions Q) that if for all $x \in X$ and $t \in \mathcal{T}$,
 486 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 .

488
 489 ³To aid in reasoning, you can read the shorthands as “ σ is *compatible* with P ” and “ \mathcal{T} is *compatible* with P .”

491 For a nonnegative rational cost C , we use the shorthand $Q + C$ to denote a potential function Q'
 492 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 σ, \mathcal{T}, X
 493 such that $\sigma, \mathcal{T} \models P$, we have $Q_X(\sigma) \geq Q'_X(\sigma)$.

494 In this section, we leave the concrete representation of the logical condition and the potential
 495 function abstract. In Section 6, we describe our implementation, including the representation of
 496 these conditions. For now, we make the assumptions stated above, as well as that logical conditions
 497 obey the standard rules of Boolean logic. We also assume that logical conditions and potential
 498 functions are equipped with an “assignment” operation $P' \Leftarrow P[x \leftarrow e]$ (resp. $Q' \Leftarrow Q[x \leftarrow e]$)
 499 such that if $\sigma, \mathcal{T} \models P'$ and $\sigma; e \downarrow_M^{\mathcal{T}} R; C$ then

- 500 • $\sigma[(x, t) \mapsto R(t) \mid t \in \mathcal{T}], \mathcal{T} \models P$
- 501 • If $x \in X$ and there exists v such that $R(t) = v$ for all $t \in \mathcal{T}$, then $Q'_X(\sigma) = Q_X(\sigma[(x, t) \mapsto$
 502 $R(t) \mid t \in \mathcal{T}])$

503 For simplicity, we also assume that the potential function depends only on the values of local
 504 variables in the store and not on the values of arrays. This is sufficient to handle the benchmarks
 505 we studied. We write $\sigma; \mathcal{T} \models \{P; Q; X\}$ to mean $\sigma, \mathcal{T} \models P$ and $\sigma, \mathcal{T} \vdash X$.

507 *Cost of Expressions.* Before presenting the Hoare-style logic for statements, we introduce a simpler
 508 judgment that we use for describing the resource usage of operands and expressions. The judgment
 509 is written $P \vdash_M e : C$ and indicates that, under condition P , the evaluation of e costs at most C .
 510 The rules for this judgment are presented in Figure 4. These rules are similar to those of Figure 3,
 511 with the exception that we now do not know the exact store used to evaluate the expression
 512 and must conservatively estimate the cost of array access based on the possible set of stores. We
 513 write $P \Rightarrow \text{MemReads}(o) \leq n$ to mean that for all σ and all \mathcal{T} such that $\sigma, \mathcal{T} \models P$, if $\sigma; o \downarrow_M^{\mathcal{T}} R; C$,
 514 then $\text{MemReads}(R) \leq n$. The meaning of $P \Rightarrow \text{Conflicts}(o) \leq n$ is similar.

515 *Inference Rules.* Figure 4 presents the inference rules for the Hoare-style logic for resource usage
 516 of statements. The judgment for these rules is written

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

519 which states that if (1) P holds, (2) we have Q resources and (3) all variables in X are thread-invariant,
 520 then if s terminates, it ends in a state where (1) P' holds, (2) we have Q' resources left over and
 521 (3) all variables in X' are thread-invariant. The simplest cases (e.g., Q:SKIP and Q:SEQ) simply
 522 thread conditions through without altering them (note that Q:SEQ feeds the post-conditions of s_1
 523 into the pre-conditions of s_2). Most other rules require additional potential in the pre-condition
 524 (e.g. $Q + C$), which is then discarded because it is used to pay for an operation. For example, if s_1
 525 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
 526 post-condition of s_1 and Q left in the post-condition of s_2 .

527 The most notable rules are for conditionals if e then s_1 else s_2 , which take into account the
 528 possibility of a divergent warp. There are four cases. First (Q:If1), we can statically determine that the
 529 conditional expression e does not vary across a warp: this is expressed with the premise $P \Rightarrow e \text{ unif}$,
 530 which is shorthand for

$$531 \forall \sigma, \mathcal{T}. \sigma, \mathcal{T} \models P \Rightarrow \exists c. \sigma; e \downarrow_M^{\mathcal{T}} (c)_{t \in \mathcal{T}}; C$$

532 That is, for any compatible store, e evaluates to a constant result family. In this case, only one
 533 branch is taken by the warp and the cost of executing the conditional is the maximum cost of
 534 executing the two branches (plus the cost M^{if} of the conditional and the cost C of evaluating the
 535 expression, which are added to the precondition). This is expressed by using Q' as the potential
 536 function in the post-condition for both branches. If the two branches do not use equal potential,
 537 the one that has more potential “left over” may use rule Q:WEAK (discussed in more detail later) to
 538 discard its extra potential and use Q' as a post-condition. We thus conservatively approximate the
 539

540	(OQ:VAR)	(OQ:CONST)	(OQ:PARAM)	(OQ:TID)	(EQ:OP)
541	$\frac{}{P \vdash_M x : M^{\text{var}}}$	$\frac{}{P \vdash_M c : M^{\text{const}}}$	$\frac{}{P \vdash_M p : M^{\text{const}}}$	$\frac{}{P \vdash_M \text{tid} : M^{\text{var}}}$	$P \vdash_M o_1 : C_1 \quad P \vdash_M o_2 : C_2 \quad \frac{}{P \vdash_M o_1 \text{ op } o_2 : C_1 + C_2 + M^{\text{op}}}$
542					
543					
544	(EQ:GARRAY)		(EQ:SARRAY)		(Q:SKIP)
545	$P \vdash_M o : C \quad P \Rightarrow \text{MemReads}(o) \leq n$		$P \vdash_M o : C \quad P \Rightarrow \text{Conflicts}(o) \leq n$		$\frac{}{\vdash_M \{P; Q; X\} \text{ skip } \{P; Q; X\}}$
546	$P \vdash_M G[o] : C + M^{\text{gread}}(n)$		$P \vdash_M S[o] : C + M^{\text{sread}}(n)$		
547					
548	(Q:IF1)				
549	$P \vdash_M e : C \quad \vdash_M \{P \wedge e; Q; X\} s_1 \{P'; Q'; X'\}$ $P \Rightarrow e \text{ unif} \quad \vdash_M \{P \wedge \neg e; Q; X\} s_2 \{P'; Q'; X'\}$				
550	$\vdash_M \{P; Q + M^{\text{if}} + C; X\} \text{ if } e \text{ then } s_1 \text{ else } s_2 \{P'; Q'; X'\}$		(Q:IF2)		
551			$P \vdash_M e : C \quad P \Rightarrow e \quad \vdash_M \{P; Q; X\} s_1 \{P'; Q'; X'\}$		
552			$\vdash_M \{P; Q + M^{\text{if}} + C; X\} \text{ if } e \text{ then } s_1 \text{ else } s_2 \{P'; Q'; X'\}$		
553		(Q:IF3)			
554		$P \vdash_M e : C \quad P \Rightarrow \neg e \quad \vdash_M \{P; Q; X\} s_2 \{P'; Q'; X'\}$			
555		$\vdash_M \{P; Q + M^{\text{if}} + C; X\} \text{ if } e \text{ then } s_1 \text{ else } s_2 \{P'; Q'; X'\}$			
556					
557	(Q:IF4)				
558	$P \vdash_M e : C \quad \vdash_M \{P \wedge e; Q; X\} s_1 \{P_1; Q_1; X_1\} \quad P \wedge \neg e \Rightarrow P' \quad P_1 \Rightarrow P''$ $\{P'; Q_1; X_1 \setminus W(s_1)\} s_2 \{P_2; Q_2; X_2\} \quad P_1 \Rightarrow P'' \quad P_2 \Rightarrow P'' \quad X' = X_2 \setminus W(s_2)$				
559	$\vdash_M \{P; Q + M^{\text{if}} + C + M^{\text{div}}; X\} \text{ if } e \text{ then } s_1 \text{ else } s_2 \{P''; Q_2; X'\}$				
560					
561	(Q:SEQ)		(Q:WHILE1)		
562	$\vdash_M \{P; Q; X\} s_1 \{P_1; Q_1; X_1\}$ $\vdash_M \{P_1; Q_1; X_1\} s_2 \{P'; Q'; X'\}$		$P \Rightarrow e \text{ unif} \quad P \vdash_M e : C$ $\vdash_M \{P \wedge e; Q; X\} s \{P; Q + M^{\text{if}} + C; X\}$		
563	$\vdash_M \{P; Q; X\} s_1; s_2 \{P'; Q'; X'\}$		$\vdash_M \{P; Q + M^{\text{if}} + C; X\} \text{ while } (e) s \{P \wedge \neg e; Q; X\}$		
564					
565		(Q:WHILE2)			
566	$P \vdash_M e : C \quad X' = X \setminus W(s)$	$\vdash_M \{P \wedge e; Q; X\} s \{P; Q + M^{\text{if}} + M^{\text{div}} + C; X\}$			
567	$\vdash_M \{P; Q + M^{\text{if}} + M^{\text{div}} + C; X\} \text{ while } (e) s \{P \wedge \neg e; Q; X'\}$				
568					
569	(Q:VWRITE1)		(Q:VWRITE2)		
570	$x \in X \quad P \Rightarrow e \text{ unif} \quad P \vdash_M e : C$ $P \Leftarrow P'[x \leftarrow e] \quad Q \Leftarrow Q'[x \leftarrow e]$		$P \vdash_M e : C \quad P \Leftarrow P'[x \leftarrow e]$		
571			$\vdash_M \{P; Q + M^{\text{vwrite}} + C; X\} x \leftarrow e \{P'; Q'; X\}$		
572					
573		(Q:GWRITE)			
574		$P \vdash_M o : C_1 \quad P \vdash_M e : C_2 \quad P \Rightarrow \text{MemReads}(o) \leq n$			
575		$\vdash_M \{P; Q + M^{\text{gwrite}}(n) + C_1 + C_2; X\} G[o] \leftarrow e \{P; Q; X\}$			
576					
577		(Q:SWRITE)			
578		$P \vdash_M o : C_1 \quad P \vdash_M e : C_2 \quad P \Rightarrow \text{Conflicts}(o) \leq n$			
579		$\vdash_M \{P; Q + M^{\text{swrite}}(n) + C_1 + C_2; X\} S[o] \leftarrow e \{P; Q; X\}$			
580					
581	(Q:WEAK)				
582	$\vdash_M \{P_2; Q_2; X_2\} s \{P'_2; Q'_2; X'_2\} \quad P_1 \Rightarrow P_2 \quad Q_1 \succeq Q_2 \quad X_1 \supset X_2 \quad P'_2 \Rightarrow P'_1 \quad Q'_2 \succeq Q'_1 \quad X'_2 \supset X'_1$				
583	$\vdash_M \{P_1; Q_1 + C; X_1\} s \{P'_1; Q'_1 + C; X'_1\}$				
584					

Fig. 4. Hoare logic rules for resource analysis.

589 potential left over after executing one branch. In the next two cases (Q:If2 and Q:If3), we are able
 590 to statically determine that the conditional expression is either true or false in any compatible store
 591 (i.e., either $P \Rightarrow e$ or $P \Rightarrow \neg e$), and we need only the “then” or “else” branch, so only the respective
 592 branch is considered in these rules.

593 In the final case (Q:If4), we consider the possibility that the warp may diverge. In addition to
 594 accounting for the case where we must execute s_1 followed by s_2 in sequence, this rule must also
 595 subsume the three previous cases, as it is possible that we were unable to determine statically that
 596 the conditional would not diverge (i.e., we were unable to derive the preconditions of Q:If1) but
 597 the warp does not diverge at runtime. To handle both cases, we require that the precondition of s_2
 598 is implied by:

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

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

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

611 The rules Q:WHILE1 and Q:WHILE2 charge the initial evaluation of the conditional ($M^{\text{if}} + C$) to
 612 the precondition. For the body of the loop, as with other Hoare-style logics, we must derive a loop
 613 invariant: the condition P must hold at both the beginning and end of each iteration of the loop
 614 body (we additionally know that e holds at the beginning of the body). In addition, the potential
 615 after the loop body must be sufficient to “pay” $M^{\text{if}} + C$ for the next check of the conditional, and still
 616 have potential Q remaining to execute the next iteration if necessary. Recall that Q is a function of
 617 the store. So this premise requires that the value of a store element (e.g. a loop counter) change
 618 sufficiently so that the corresponding decrease in potential $Q_X(\sigma)$ is able to pay the appropriate
 619 cost. The difference between the two rules is that Q:WHILE1 assumes the warp does not diverge, so
 620 we need not pay M^{div} and also need not remove variables assigned by the loop body from X .

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

626 Finally, as discussed above, Q:WEAK allows us to strengthen the preconditions and weaken
 627 the postconditions of a derivation. If s can execute with precondition $\{P_2; Q_2; X\}$ and postcondition
 628 $\{P'_2; Q'_2; X\}$, it can also execute with a precondition P_1 that implies P_2 and a potential function Q_1
 629 that is always greater than Q_2 . In addition, it can guarantee any postcondition implied by P'_2 and
 630 any potential function Q'_1 that is always less than Q'_2 . We can also take subsets of X as necessary in
 631 derivations. The rule also allows us to add a constant potential to both the pre- and post-conditions.

632 *Example Derivation.* Figure 5 steps through a derivation for the addSub3 kernel from Section 2,
 633 with the pre- and post-conditions interspersed in red. The code is simplified to more closely resemble
 634 miniCUDA. For illustrative purposes, we consider only the costs of array accesses (writes and reads)
 635 and assume all other costs are zero. The potential annotation consists of two parts: the *constant*
 636

```

638 1    $L \triangleq 2M^{\text{pread}}(1) + 2M^{\text{gwrite}}(5)$ 
639 2    $\{\top; M^{\text{swrite}}(1) + M^{\text{gread}}(4) + \frac{h}{2}L; \{w, h, j\}\}$ 
640 3   __global__ void addSub3 (int *A, int *B, int w, int h) {
641 4       __shared__ int As[32];
642 5       As[tid % 32] = A[tid];
643 6       for (int j = 0; j < h; j += 2) {
644 7           B[j * w + tid] += As[tid];
645 8           B[(j + 1) * w + tid] -= As[tid];
646 9       }
647 10  }
648

```

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

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 reads⁴ 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 the soundness of the analysis: 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 $\sigma; \mathcal{T} \models \{P; Q; X\}$, 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 $\sigma, \mathcal{T} \models P$ and $\sigma; e \downarrow_{\mathcal{T}}^{\mathcal{T}} R; C'$, then $C' \leq C$.*

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

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

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

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 [27], 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 [27], which shows the same bound for a general computation of work W and span S under a greedy schedule on P processors. \square

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 function⁵, which acts as a barrier forcing all warps to wait to proceed until all warps have reached the synchronization point. Consider the following code:

```
729 __global__ void tricky(int *A) {
730     if (threadIdx.x < 32) {
731         A[threadIdx.x] = 42;
732     }
}
```

⁵We 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.

```

736     __syncthreads();
737     if (threadIdx.x >= 32) {
738         A[threadIdx.x] = 42;
739     }
740 }
```

741 Assume that the latency of the write to global memory dominates the span of the computation. Each
 742 warp performs only one write, and so taking the maximum span would result in an assumption
 743 that the span of the block includes only one write. However, because of the synchronization in the
 744 middle, the span of the block must actually account for the latency of two writes: threads 32 and
 745 up must wait for threads 0-31 to perform their writes before proceeding.

746 Determining that, in the kernel above, each warp only performs one write, would require a
 747 sophisticated analysis that tracks costs separately for each thread: this is precisely what our analysis,
 748 to retain scalability, does not do. As a result, it is sound to simply compose the predicted spans
 749 of each warp in a block by taking the maximum. The remainder of this section will be devoted to
 750 proving this fact. In order to do so, we develop another cost semantics, this time a *parallel* semantics
 751 that models entire CUDA blocks and tracks the cost in terms of work and span.

752 The cost semantics tracks the work and span for each warp. At each synchronization, we take
 753 the maximum span over all warps to account for the fact that all warps in the block must wait
 754 for each other at that point. A cost is now a pair (c^w, c^s) of the work and span, respectively. A
 755 resource metric M maps resource constants to costs reflecting the number of instructions and
 756 latency required by the operation. We can take projections M_w and M_s of such a resource metric
 757 which project out the work and span components, respectively, of each cost. For the purposes of
 758 calculating the span, we assume that the span of an operation (the second component of the cost)
 759 reflects the time taken to process the instruction plus the latency (in other words, the latency is the
 760 span of the operation minus the work). We represent the cost of a block as a warp-indexed family
 761 of costs C . We use \emptyset to denote the collection $((0, 0)_{i \in Warps})$. We will use a shorthand for adding a
 762 cost onto a collection for a subset of warps:

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

763 We will overload the above notation to add a cost onto a collection for a subset of threads:
 764

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

765 where $WarpOf(t)$ is the warp containing thread t .

766 We denote the work of a collection by $W(C)$ and the span by $S(C)$. We can calculate the work
 767 and span of a block by summing and taking the maximum, respectively, over the warps:
 768

$$\begin{aligned} W(C) &\triangleq \sum_{i \in Warps} \text{fst } C_i \\ S(C) &\triangleq \max_{i \in Warps} \text{snd } C_i \end{aligned}$$

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

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

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

772 meaning that, on a set of threads \mathcal{T} , with heap σ , the statement s results in final heap σ' and if the
 773 cost collection is initially C , the cost extended with the cost of executing s is C' . The judgments for
 774 operands and expressions are similar; as with the lock-step cost semantics, they return a result
 775 family rather than a new heap. The rule SC:SYNC handles taking the maximum at synchronization
 776 points, and rules SC:IF and SC:WHILE add the cost of divergence only to warps that actually diverge.
 777

778

785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833	$\text{(SC:SKIP)} \quad \sigma; C; \text{skip } \Downarrow_M^{\mathcal{T}} \sigma; C$ $\text{(SC:SYNC)} \quad \sigma; (W, S); \text{sync } \Downarrow_M^{\mathcal{T}} \sigma; (W, (\max S) \text{warps}) \oplus_{\mathcal{T}} M^{\text{sync}}$ $\text{(SC:SEQ)} \quad \frac{\sigma; C; s_1 \Downarrow_M^{\mathcal{T}} \sigma_1; C' \quad \sigma_1; C'; s_2 \Downarrow_M^{\mathcal{T}} \sigma_2; C''}{\sigma; C; s_1; s_2 \Downarrow_M^{\mathcal{T}} \sigma_2; C''}$ $\text{(SC:VWRITE)} \quad \frac{}{\sigma; C; x \leftarrow e \Downarrow_M^{\mathcal{T}} \sigma[(x, t) \mapsto R(t) \mid t \in \mathcal{T}]; C' \oplus_{\mathcal{T}} M^{\text{vwrite}}}$ $\text{(SC:IF)} \quad \frac{\sigma; C; e \Downarrow_M^{\mathcal{T}} R; C' \quad \mathcal{T}_T = \{t \in \mathcal{T} \mid R_t\} \quad \mathcal{T}_F = \{t \in \mathcal{T} \mid \neg R_t\} \quad \sigma; C'; s_1 \Downarrow_M^{\mathcal{T}_T} \sigma'; C'' \quad \sigma'; C''; s_2 \Downarrow_M^{\mathcal{T}_F} \sigma''; C'''}{\sigma; C; \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow_M^{\mathcal{T}} \sigma''; C''' \oplus_{\mathcal{T}} M^{\text{if}} \oplus_{\{w \mid \exists t_1 \in \mathcal{T}_T, t_2 \in \mathcal{T}_F. \text{WarpOf}(t_1) = \text{WarpOf}(t_2)\}} M^{\text{div}}}$ $\text{(SC:WHILE)} \quad \frac{\sigma; C; e \Downarrow_M^{\mathcal{T}} R; C' \quad \sigma; C'; s \Downarrow_M^{\mathcal{T}} \sigma'; C'' \quad \mathcal{T}_T = \{t \in \mathcal{T} \mid R_t\} \quad \sigma'; C''; \text{while } (e) s \Downarrow_M^{\mathcal{T}} \sigma''; C'''}{\sigma; C; \text{while } (e) s \Downarrow_M^{\mathcal{T}} \sigma''; C''' \oplus_{\mathcal{T}} M^{\text{if}} \oplus_{\{w \mid \exists t_1 \in \mathcal{T}_T, t_2 \in \mathcal{T} \setminus \mathcal{T}_T. \text{WarpOf}(t_1) = \text{WarpOf}(t_2)\}} M^{\text{div}}}$
---	--

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

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 essentially 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 these semantics. 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 show formally, these two features of the warp-level cost semantics cancel each other 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 $\sigma; C; e \Downarrow_M^{\mathcal{T}} R; C'$ then there exist C_w and C_s such that

- (1) $\sigma; e \Downarrow_{M_w}^{\mathcal{T}} R; C_w$
- (2) $\sigma; e \Downarrow_{M_s}^{\mathcal{T}} R; C_s$
- (3) $W(C') - W(C) \leq C_w |\{ \text{WarpOf}(t) \mid t \in \mathcal{T} \}|$
- (4) $S(C') - S(C) \leq C_s$

LEMMA 3. If $\Sigma \vdash s$ and $\Sigma \vdash_{\mathcal{T}} \sigma$ and $\sigma; C; s \Downarrow_M^{\mathcal{T}} \sigma'; C'$ then there exist C_w and C_s such that

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

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

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

840 THEOREM 3. If $\Sigma \vdash s$ and $\Sigma \vdash_{\mathcal{T}} \sigma$ and $\vdash_M \{P; Q_w; X_w\} s \{P'; Q'_w; X'_w\}$ and $\vdash_M \{P; Q_s; X_s\} s \{P'; Q'_s; X'_s\}$
 841 and $\sigma; \mathcal{T} \vDash \{P; Q_w; X_w\}$ and $\sigma; \mathcal{T} \vDash \{P; Q_s; X_s\}$ and $\sigma; \emptyset; s \Downarrow_M^{\mathcal{T}} \sigma'; C$ then

$$842 \quad 843 \quad W(C) \leq (Q_{wX_w}(\sigma) - Q'_{wX'_w}(\sigma')) |\{WarpOf(t) \mid t \in \mathcal{T}\}|$$

844 and

$$845 \quad S(C) \leq Q_{sX_s}(\sigma) - Q'_{sX'_s}(\sigma')$$

846 PROOF. Note that $W(\emptyset) = S(\emptyset) = 0$, so by Lemma 3, we have C_w and C_s such that

- 847 (1) $\sigma; s \Downarrow_{M_w}^{\mathcal{T}} \sigma'; C_w$
 848 (2) $\sigma; s \Downarrow_{M_s}^{\mathcal{T}} \sigma'; C_s$
 849 (3) $W(C) \leq C_w |\{WarpOf(t) \mid t \in \mathcal{T}\}|$
 850 (4) $S(C) \leq C_s$

852 and by Theorem 1, we have $Q_{wX_w}(\sigma) - C_w \geq Q'_{wX'_w}(\sigma') \geq 0$ and $Q_{sX_s}(\sigma) - C_s \geq Q'_{sX'_s}(\sigma') \geq 0$.
 853 The result follows. \square

855 6 INFERENCE AND IMPLEMENTATION

856 In this section, we discuss the implementation of the logical conditions and potential functions of
 857 Section 4 and the techniques used to automate the reasoning in our tool RACUDA. The automation
 858 is based on instantiations of the boolean conditions and potential functions similar to existing
 859 work [9, 10]. We have implemented the inference algorithms as an extension to the Absynth
 860 tool [9, 28]. We begin by outlining our implementation, and then detail the instantiations of the
 861 potential annotations and logical conditions.

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

869 RACUDA uses a modified version of Front-C⁶ to parse CUDA, which we then lower into a
 870 representation similar to miniCUDA. Another set of transformations converts this representation
 871 into IMP, a simple resource-annotated imperative language that serves as a front-end to Absynth.
 872 Part of this process is adding annotations that express the cost of each operation in the given
 873 resource metric.

874 We have extended the abstract interpretation pass to gather CUDA-specific information that
 875 allows us to bound the values of the functions MemReads(\cdot) and Conflicts(\cdot) (recall that these
 876 functions were used to select which rules to apply in the program logic of Figure 4, but their
 877 definitions were left abstract). We describe the extended abstraction domain in the next subsection.

878 For the most part, we did not modify the representation of potential functions, but we briefly
 879 discuss this representation at the end of this section. In addition, we implemented a “simulated

881 ⁶<https://github.com/BinaryAnalysisPlatform/FrontC>

evaluation” mode that interprets the CUDA representation according to the cost semantics of Section 3.

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 $\mathcal{M}_{tid}(x)$ 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 $\mathcal{M}_{const}(x)$ 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 $\mathcal{M}_{tid}(x) = (0, 0, 0)$, then the value of x is guaranteed to be constant across threads. As another example, if $\mathcal{M}_{tid}(x) = (1, 1, 1)$, then $x = tid + c$, where c does not depend on the thread, and so the array access $A[x]$ has a stride of 1.

This information can directly be extended to expressions, and therefore to update the variable-specific information at assignments and determine whether expressions used in conditionals might be divergent. The use of this information to predict uncoalesced memory accesses and bank conflicts is more interesting. We assume the following definitions of $\text{MemReads}(\cdot)$ and $\text{Conflicts}(\cdot)$, now generalized to use m as the number of array elements accessed by a global read and B as the number of shared memory banks.

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

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

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

PROOF. By the definition of $\sigma, \mathcal{T} \models (C, \mathcal{M})$, 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 $\text{Conflicts}(x)$ given abstract information about x . This bound assumes that x is divergent; for non-divergent operands o , it is the case by assumption that $\text{Conflicts}(o) = 1$. The proof relies on Lemma 4, a stand-alone result about modular arithmetic.

932 THEOREM 5. If $\mathcal{M}_{\text{tid}}(x) = k > 0$ and $C \Rightarrow \text{tid} \in [t, t']$ and $\sigma, \mathcal{T} \models (C, \mathcal{M})$ and $\sigma; x \downarrow_M^{\mathcal{T}} R; C$ then

$$933 \\ 934 \quad \text{Conflicts}(R) \leq \left\lceil \frac{t' - t}{\min(t' - t, \frac{B}{\gcd(k, B)})} \right\rceil \\ 935$$

936 PROOF. Let $t_0 \in \mathcal{T}$. By the definition of $\sigma, \mathcal{T} \models (C, \mathcal{M})$, we have $\text{Tid}(t_0) \in [t, t']$. We have $R(t_0) =$
 937 $kt_0 + c$. Let $R' = (kt \bmod B)_{t \in \mathcal{T}}$. The accesses in R access banks from R' at uniform stride, and
 938 so the maximum number of times any such bank is accessed in R is $\left\lceil \frac{t' - t}{|\text{Dom}(R')|} \right\rceil$. The result follows
 939 from Lemma 4. \square

941 LEMMA 4. Let $k, m, n, a \in \mathbb{N}$ and $m \leq n$. Then $|\{i \cdot a \bmod n \mid i \in \{k, \dots, k + m - 1\}\}| =$
 942 $\min(m, \frac{n}{\gcd(a, n)})$.

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

951 As an example of how the abstraction information is tracked and used in the resource analysis, we
 952 return to the code example in Figure 5. Figure 7 steps through the abstract interpretation of the same
 953 code. For the purposes of this example, we have declared two variables `temp0` and `temp1` to hold
 954 intermediate computations. This reflects more closely the intermediate representation on which
 955 the abstract interpretation is done. We establish C from parameters provided to the analysis that
 956 specify that `blockDim.x` is 32, which also bounds `threadIdx.x`. The assignment to `i` on line 3 then
 957 establishes that `i` is a multiple of `threadIdx.x` and has a constant component of `32blockIdx.x`.
 958 This information is then used on line 4 to bound `MemReads(i)` and `Conflicts(threadIdx.x)`. By
 959 Theorem 4, we can bound `MemReads(i)` by $\lceil \frac{32}{m} \rceil + 1$. Note that both t and t' in the statement of
 960 Theorem 4 are multiples of 32 (and, in practice, m will divide 32), so we can improve the bound
 961 to $\lceil \frac{32}{m} \rceil$. By Theorem 5, we can bound `Conflicts(threadIdx.x)` by $\left\lceil \frac{32}{\min(32, \frac{32}{\gcd(1, 32)})} \right\rceil = 1$.

962 When `j` is declared, it is established to have no thread-dependence. Its constant component is
 963 initially zero, but the loop invariant sets $\mathcal{M}_{\text{const}}(j) = \top$. The assignments on lines 6 and 8 propagate
 964 the information that `i` depends on `threadIdx.x` as well as some additional information about the
 965 constant components. This information is used in the two global loads to bound `MemReads(temp0)`
 966 and `MemReads(temp1)` by $\lceil \frac{32}{m} \rceil + 1$. In this case, without further information about the value of `w`,
 967 we are unable to make any assumptions about alignment and cannot derive a more precise bound.
 968 As above, we can determine $\text{Conflicts}(i) \leq 1$ for the loads on both lines.

969 *Potential functions.* Our implementation of potential functions is taken largely from prior work on
 970 AARA for imperative programs [9, 28]. We instantiate a potential function Q as a linear combination
 971 of a fixed set I of *base functions* from stores to rational costs, each depending on a portion of the
 972 state. A designated base function b_0 is the constant function and tracks constant potential. For each
 973 program, we select a set of N base functions, plus the constant function, denoted b_0, b_1, \dots, b_N , that
 974 capture the portions of the state relevant to calculating potential. A potential function Q is then a
 975 linear combination of the selected base functions:

$$976 \\ 977 \quad Q(\sigma) = q_0 + \sum_{i=1}^N q_i b_i(\sigma) \\ 978 \\ 979$$

```

981 1 __global__ void addSub3 (int *A, int *B, int w, int h) { blockDim.x = 32, threadIdx.x ≤ 32
982 2   __shared__ int As[blockDim.x];
983 3   int i = blockIdx.x * blockDim.x + threadIdx.x;  $M_{tid}(i) = (1, 0, 0)$ ,  $M_{const}(i) = 32blockIdx.x$ 
984 4   As[threadIdx.x] = A[i];
985 5   for (int j = 0; j < h; j += 2) {  $M_{tid}(j) = (0, 0, 0)$ ,  $M_{const}(j) = \top$ 
986 6     int temp0 = j * w + i;
987 7     B[temp0] += As[i];
988 8     int temp1 = (j + 1) * w + i;  $M_{tid}(temp0) = (1, 0, 0)$ ,  $M_{const}(temp0) = j * w$ 
989 9     B[temp1] -= As[i];
99010   }
99111 }
```

Fig. 7. A sample abstract interpretation.

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

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

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 [9].

7 EVALUATION

We evaluated the range 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 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

	Benchmark	Source	LoC	Params.	Block
1030	matMul	SDK	26	N	32×32
1031	matMulBad	SDK*		N	32×32
1032	matMulTrans	SDK*	26	N	32×32
1033	mandelbrot	SDK	78	N	32×1
1034	vectorAdd	SDK	5	N	256×1
1035	reduceN	SDK	14–18	N	256×1
1036	histogram256	SDK	19	N	64×1
1037	addSub0	Us	9	h, w	$h \times 1$
1038	addSub1	Us	7	h, w	$\frac{h}{2} \times 1$
1039	addSub2	Us	7	h, w	$w \times 1$
1040	addSub3	Us	8	h, w	$w \times 1$
1041					
1042					

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.

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: “conflicts”, “sectors”, “divwarps” and “steps”.

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 $N \bmod 32$ 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

Benchmark	Error on metric “sectors”				Error on metric “conflicts”			
	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	4.9×10^{-3}	4.7×10^{-3}	0.01	4.98×10^{-3}	0	0	0	0
addSub1	4.9×10^{-3}	0.01	0.01	0.01	0	0	0	0
addSub2	0	0.01	0.03	0.01	0	0	0	0
addSub3	0	0.03	0.04	0.02	0	0	0	0

Table 3. Error of the cost semantics with respect to profiled values for the “sectors” and “conflicts” metrics.

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. This calculation of the error reflects the fact that, 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

Benchmark	Time (s)	Predicted Bound	Error			
			32N	32N + 1	Random	Average
matMul	0.11	$31(31 + N) \lceil \frac{N}{32} \rceil$	—	—	—	—
matMulBad	8.13	0	0	0	0	0
matMulTrans	0.12	$1023 \frac{31+N}{32} \lceil \frac{N}{32} \rceil$	33.4	33.4	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	0	0
reduce1	0.02	$23715 \lceil \frac{N}{32} \rceil$	1806	1806	1806	1806
reduce2	1.07	n/b	n/b	n/b	n/b	n/b
reduce2a	0.03	0	0	0	0	0
reduce3	1.54	n/b	n/b	n/b	n/b	n/b
reduce3a	0.03	0	0	0	0	0
histogram	1.19	$56 \frac{63+N}{64}$	0	0	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

Table 4. Analysis time, inferred bound and average error for the “conflicts” metric. For matMul, the correct value is 0 for all inputs, so no error is reported. An entry of “n/b” indicates that our analysis was unable to determine a bound.

Benchmark	Time (s)	Predicted Bound	Error			
			32N	32N + 1	Random	Average
matMul	1.05	$(4 + 10 \frac{31+N}{32}) \lceil \frac{N}{32} \rceil$	0.30	0.30	0.30	0.30
matMulBad	8.31	$15(31 + N) \lceil \frac{N}{32} \rceil$	0.20	0.20	0.20	0.20
matMulTrans	1.05	$(4 + 10 \frac{31+N}{32}) \lceil \frac{N}{32} \rceil$	0.30	0.30	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}	7.23×10^{-5}	1.4×10^{-4}
reduce0	0.03	$5 \lceil \frac{N}{32} \rceil$	0.21	0.21	0.21	0.21
reduce1	0.02	$5 \lceil \frac{N}{32} \rceil$	0.21	0.21	0.21	0.21
reduce2	0.09	$5 \lceil \frac{N}{32} \rceil$	0.21	0.21	0.21	0.21
reduce3	0.10	$9 \lceil \frac{N}{32} \rceil$	1.22	1.22	1.22	1.22
histogram	1.14	$\frac{1}{64}(5740 + 10(63 + N))$	0.25	0.25	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 5. Analysis time, inferred bound and average error for the “sectors” metric.

Benchmark	Time (s)	Predicted Bound	Error			
			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	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 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.

show that RACUDA is quite efficient, with analysis times usually under 1 second; analysis times on the order of minutes are seen for exceptionally complex kernels. Recall that RACUDA produces bounds for a single warp. To obtain bounds for the kernel, we multiplied by the number of warps required to process the entire input (often $\lceil \frac{N}{32} \rceil$ if the input size is N). Several of the kernels perform internal loops with a stride of 32. Precise bounds of such loops would, like the number of warps, be of the form $\lceil \frac{N}{32} \rceil$. However, Absynth can only produce polynomial bounds and so must approximate this bound by $\frac{N+31}{32}$, which is the tightest possible polynomial bound.

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

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

neglecting inputs that would cause a division by zero.

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

Benchmark	Time (s)	Predicted Bound	Error			
			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	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

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.

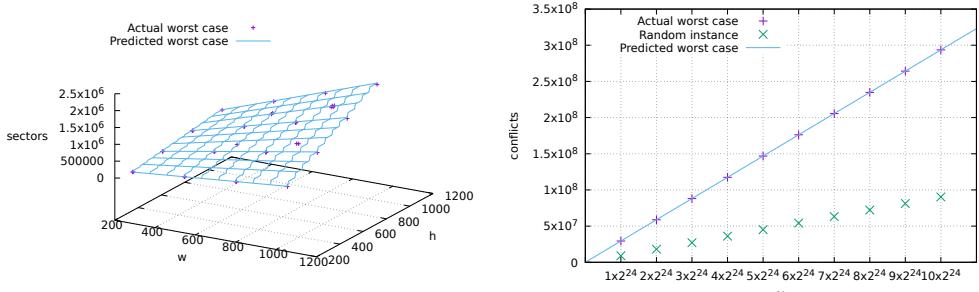


Fig. 8. Our inferred cost estimates (blue line) vs. actual worst-case costs (purple crosses) for various inputs. **Left:** addSub1, sectors; **Right:** histogram, conflicts (also includes a random input, green crosses)

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

1275 the worst-case input. The benchmark used for this figure is histogram, whose shared memory
 1276 performance displays interesting behavior depending on the input. The benchmark computes a
 1277 256-bin histogram of the number of occurrences of each byte (0x0-0xff) in the input array. The bins
 1278 are stored in shared memory, and so occurrences of bytes that map to the same shared memory
 1279 bank (e.g. 0x00 and 0x20) in the same warp might result in bank conflicts⁷. In the worst case, all
 1280 bytes handled by a warp map to the same memory bank, resulting in an 8-way conflict at each
 1281 access (32 bins are accessed, but as there are only 256 bins, only $256/32 = 8$ map to each memory
 1282 bank). On the other hand, in a random input, bytes are likely to be much more evenly distributed.
 1283 This figure shows the benefit of static analysis over random testing in safety-critical applications
 1284 where soundness is important: at least in this benchmark, random testing is highly unlikely to
 1285 uncover, or even approach, the true worst case.

1286 We present the results for the “divwarps” and “steps” metrics in Tables 6 and 7. For the purposes
 1287 of these tables, the bounds are shown per warp rather than for the entire kernel (composing the
 1288 “steps” metric across multiple warps is not so straightforward, as we discuss in Section 5). Again,
 1289 the analysis is fairly efficient, though in this table we see that the performance of RACUDA is
 1290 harmed most by nesting of divergent conditionals and loops, as in the benchmarks SYN-BRDIS
 1291 and SYN-BRDIS-OPT. Still, analysis times remain at most on the order of minutes (and are still
 1292 under 1 second for most benchmarks). For the “steps” and “divwarps” metrics, we do not compare
 1293 to a profiled GPU execution because NVIDIA’s profiling tools do not collect metrics directly related
 1294 to these. Instead, we compare to RACUDA’s “evaluation mode”.

1295 These experiments show the utility of RACUDA over tools that merely identify one type of
 1296 performance bug. Often, there is a tradeoff between two performance bottlenecks. For example,
 1297 reduce3 has worse global memory performance than reduce2, but performs the first level of the
 1298 reduction immediately from global memory, reducing shared memory accesses. By combining these
 1299 into a metric (e.g. “steps”) that takes account the relative cost of each operation, we can explore the
 1300 tradeoff: we see that in terms of “steps”, reduce2 is more efficient than reduce3, but the situation will
 1301 likely be reversed depending on the actual runtime costs of each operation, which we do not attempt
 1302 to profile for the purposes of this evaluation. As another example, the SYN-BRDIS-OPT kernel was
 1303 designed to reduce the impact of divergent warps over SYN-BRDIS using a transformation called
 1304 *branch distribution*. Branch distribution factors out code common to two branches of a divergent
 1305 conditional (for example, *if e then (A; B; C) else (A'; B; C')* would become *(if e then A else A')*;
 1306 *B*; (*if e then C else C'*). In this code example (and in the benchmarks), the
 1307 transformation actually *increases* the *number* of divergences: we can see this in the “divwarps”
 1308 metric for the two benchmarks. However, the total amount of code that must execute sequentially
 1309 is decreased (in the small example above, code section *B* is not sequentialized), as evidenced by the
 1310 “steps” metric.

1312 8 RELATED WORK

1313 *Resource Bound Analysis.* There exist many static analyses and program logics that (automatically
 1314 or manually) derive sound performance information such as provably-correct worst-case bounds
 1315 for imperative [9, 16, 22, 35] and functional [11, 15, 19, 21, 24, 32] programs. However, there are
 1316 very few tools for parallel [20] and concurrent [1, 12] execution and there are no such tools that
 1317 take into account the aforementioned CUDA-specific performance bottlenecks.

1318
 1319
 1320 ⁷Of course, these would also result in data races in the absence of synchronization (which is present in the benchmark as
 1321 originally written), but such synchronization would also be likely to result in performance impacts when such conflicts
 1322 occur; for illustration purposes, we disable the synchronization so that this performance impact shows up as a bank conflict.

1324 Most closely related to our work is automatic amortized analysis (AARA) for imperative programs
 1325 and quantitative program logics. Carboneaux et al. [8] introduced the first imperative AARA in
 1326 the form of a program logic for verifying stack bounds for C programs. The technique has then
 1327 been automated [9, 10] using templates and LP solving and applied to probabilistic programs [28].
 1328 A main innovation of this work is the development of an AARA for CUDA code: Previous work on
 1329 imperative AARA cannot analyze parallel executions nor CUDA specific memory-access cost.

1330 *Parallel Cost Semantics.* The model we use for reasoning about a CUDA block in terms of its
 1331 work and span is derived from prior models for reasoning about parallel algorithms. The ideas of
 1332 work and span (also known as *depth*) date back to work from the 1970s and 1980s showing bounds
 1333 on execution time for a particular schedule [7] and later for any greedy schedule [13]. Starting in
 1334 the 1990s [4, 5], parallel algorithms literature has used directed acyclic graphs (DAGs) to analyze
 1335 the parallel structure of algorithms and calculate their work and span: the work is the total number
 1336 of nodes in the DAG and the span is the longest path from source to sink. In the case of CUDA, we
 1337 do not need the full generality of DAGs and so are able to simplify the notation somewhat, but our
 1338 notation for costs of CUDA blocks in Section 5 remains inspired by this prior work. We build in
 1339 particular on work by Muller and Acar [27] that extended the traditional DAG models to account
 1340 for latency (Muller and Acar were considering primarily costly I/O operations; we use the same
 1341 techniques for the latency of operations such as memory accesses). Their model adds latencies as
 1342 edge weights on the graph and redefines the span (but not work) to include these weights.
 1343

1344 *Analysis of CUDA Code.* Given its importance in fields such as machine learning and high-
 1345 performance computing, CUDA has gained a fair amount of attention in the program analysis
 1346 literature in recent years. There exist a number of static [25, 31, 37] and dynamic [6, 14, 30, 36]
 1347 analyses for verifying certain properties of CUDA programs, but much of this work focused on
 1348 functional properties, e.g., freedom from data races. Wu et al. [36] investigate several classes of bugs,
 1349 one of which is “non-optimal implementation”; this class includes several types of performance
 1350 problems. They don’t give examples of kernels with non-optimal implementations, and don’t
 1351 specify whether or how their dynamic analysis detects such bugs. PUG [25] and Boyer et al. [6]
 1352 focus primarily on detecting data races but both demonstrate an extension of their race detectors
 1353 designed to detect bank conflicts, albeit with somewhat imprecise results. Kojima and Igarashi
 1354 [23] present a Hoare logic for proving functional properties of CUDA kernels. Their logic does not
 1355 consider quantitative properties and, unlike our program logic, requires explicit reasoning about
 1356 the sets of active threads at each program point, which poses problems for designing an efficient
 1357 automated inference engine.

1358 GKLEE [26] is an analysis for CUDA kernels based on concolic execution, and targets both
 1359 functional errors and performance errors (including warp divergence, non-coalesced memory
 1360 accesses and shared bank conflicts). GKLEE requires some user annotations in order to perform its
 1361 analysis. Alur et al. [2] and Singhania [34] have developed several static analyses for performance
 1362 properties of CUDA programs, including uncoalesced memory accesses. Their analysis for detecting
 1363 uncoalesced memory accesses uses abstract interpretation with an abstract domain similar to ours
 1364 but simpler (in our notation, it only tracks $M_{tid}(x)$ and only considers the values 0, 1, and -1 for it).
 1365 Their work does not address shared bank conflicts or divergent warps. Moreover, they developed a
 1366 separate analysis for each type of performance bug. In this work, we present a general analysis
 1367 that detects and quantifies several different types of performance bugs.

1368 The two systems described in the previous paragraph only detect performance errors (e.g., they
 1369 might estimate what percentage of warps in an execution will diverge); they are not able to quantify
 1370 the impact of these errors on the overall performance of a kernel. The analysis in this paper has
 1371 the full power of amortized resource analysis and is able to generate a resource bound, parametric
 1372

1373 in the relevant costs, that takes into account warp divergence, uncoalesced memory accesses and
1374 shared bank conflicts.

1375 Other work has focused on quantifying or mitigating, but not detecting, performance errors.
1376 Bialas and Strzelecki [3] use simple, tunable kernels to experimentally quantify the impact of warp
1377 divergence on performance using different GPUs. Their findings show that there is a nontrivial cost
1378 associated with a divergent warp even if the divergence involves some threads simply being inactive
1379 (e.g. threads exiting a loop early or a conditional with no “else” branch). This finding has shaped
1380 our thinking on the cost of divergent warps. Han and Abdelrahman [17] present two program
1381 transformations that lessen the performance impact of warp divergence; they experimentally
1382 analyze the benefit of these optimizations but do not have a way of statically identifying whether a
1383 kernel contains a potential for divergent warps and/or could benefit from their transformations.
1384

1385 9 CONCLUSION

1386 We have presented a program logic for proving qualitative and quantitative properties of CUDA
1387 kernels, and proven it sound with respect to a model of the cost of executing kernels on GPUs.
1388 We have used the logic to develop a resource analysis for CUDA as an extension to the Absynth
1389 tool, and shown that this analysis provides useful feedback on the performance characteristics of a
1390 variety of CUDA kernels.

1391 This work has taken the first step toward automated static analysis tools for analyzing and
1392 improving performance of CUDA kernels. In the future, we plan to extend the logic to handle more
1393 advanced features of CUDA such as dynamic parallelism, by further embracing the connection to
1394 DAG-based models for dynamic parallelism (Sections 5 and 8).

1395 The “steps” metric of Section 7 raises the tantalizing question of whether it is possible to use
1396 our analysis to predict actual execution times of kernels by using appropriate resource metrics
1397 to analyze the work and span and combine them as in Section 5. Deriving such metrics is largely
1398 a question of careful profiling of specific hardware, which is outside the scope of this paper, but
1399 in future work we hope to bring these techniques closer to deriving wall-clock execution time
1400 bounds on kernels. Doing so may involve further extending the analysis to handle *instruction-level*
1401 *parallelism*, which hides latency by beginning to execute instructions *in the same warp* that do not
1402 depend on data being fetched.

1403 1404 ACKNOWLEDGMENTS

1405 This article is based on research supported by DARPA under AA Contract FA8750-18-C-0092 and
1406 by the National Science Foundation under SaTC Award 1801369, CAREER Award 1845514, and SHF
1407 Awards 1812876 and 2007784. Any opinions, findings, and conclusions contained in this document
1408 are those of the authors and do not necessarily reflect the views of the sponsoring organizations.
1409

1410 1411 REFERENCES

- [1] 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)*.
- [2] 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.
- [3] 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.
- [4] Guy E. Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *Functional Programming Languages and Computer Architecture*. 226–237.
- [5] 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.

- [6] M. Boyer, K. Skadron, and W. Weimer. 2008. Automated Dynamic Analysis of CUDA Programs. In *Third Workshop on Software Tools for MultiCore Systems*.
- [7] Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (1974), 201–206.
- [8] 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)*.
- [9] 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.
- [10] 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>
- [11] 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)*.
- [12] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel Complexity Analysis with Temporal Session Types. In *23rd International Conference on Functional Programming (ICFP’18)*.
- [13] 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.
- [14] 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>
- [15] 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
- [16] 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)*.
- [17] 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>
- [18] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL’11)*. 357–370.
- [19] 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)*.
- [20] Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *24th European Symposium on Programming (ESOP’15)*.
- [21] 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>
- [22] 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)*.
- [23] 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>
- [24] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS’11)*.
- [25] 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)*.
- [26] 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)*.
- [27] 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.
- [28] 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>
- [29] NVIDIA Corporation. 2019. *CUDA C Programming Guide v.10.1.168*.

- 1471 [30] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A Dynamic CUDA Race Detector. In *Proceedings of*
1472 *the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York,
1473 NY, USA, 390–403. <https://doi.org/10.1145/3192366.3192368>
- 1474 [31] Phillippe Pereira, Hugo Albuquerque, Hendrio Marques, Isabela Silva, Celso Carvalho, Lucas Cordeiro, Vanessa Santos,
1475 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.
1476 <https://doi.org/10.1145/2851613.2851830>
- 1477 [32] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2017. Monadic Refinements for
1478 Relational Cost Analysis. *Proc. ACM Program. Lang.* 2, POPL (2017).
- 1479 [33] Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. 2020. Sponge Examples:
1480 Energy-Latency Attacks on Neural Networks. arXiv:cs.LG/2006.03463
- 1481 [34] Nimit Singhania. 2018. *Static Analysis for GPU Program Performance*. Ph.D. Dissertation. Computer and Information
1482 Science, University of Pennsylvania.
- 1483 [35] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Approach to Bound Analysis and
1484 Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*.
- 1485 [36] Mingyuan Wu, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2019. Characterizing and Detecting
1486 CUDA Program Bugs. *CoRR* abs/1905.01833 (2019). arXiv:1905.01833 <http://arxiv.org/abs/1905.01833>
- 1487 [37] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A Low-overhead Mechanism for Detecting
1488 Data Races in GPU Programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel
1489 Programming (PPoPP '11)*. ACM, New York, NY, USA, 135–146. <https://doi.org/10.1145/1941553.1941574>
- 1490
- 1491
- 1492
- 1493
- 1494
- 1495
- 1496
- 1497
- 1498
- 1499
- 1500
- 1501
- 1502
- 1503
- 1504
- 1505
- 1506
- 1507
- 1508
- 1509
- 1510
- 1511
- 1512
- 1513
- 1514
- 1515
- 1516
- 1517
- 1518
- 1519