Automatic Resource Analysis for CUDA Kernels

Stefan K. Muller
Carnegie Mellon University

Jan Hoffmann
Carnegie Mellon University

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

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

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

Nevertheless, writing a kernel that executes efficiently on a GPU is not as simple as writing a C function: small changes to a kernel, which might be inconsequential for sequential CPU code, can have drastic impact on its performance. The CUDA C Programming Guide [29] lists three particularly pervasive performance bottlenecks to avoid: divergent warps, uncoalesced memory accesses, and shared memory bank conflicts. Divergent warps result from CUDA’s execution model: a group of threads (often 32 threads, referred to as a warp) execute the same instruction on possibly different data. C functions, however, can perform arbitrary branching that can cause different threads of a warp to diverge, i.e., take different branches. CUDA is able to compile such code and execute it on a GPU, but at a fairly steep performance cost, as the two branches must be executed sequentially. Even if a conditional only has one branch, there is nontrivial overhead associated with divergence [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, 23, 24, 33] and dynamic [4, 35] 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 diverging conditionals since 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 [15].

In this paper, we present an automated amortized resource analysis (AARA) [16, 19] which 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. For example, a simple resource metric could count the number of divergent warps during an execution. More precise resource metrics can quantify the performance gains that are, for example, achieved by the aforementioned refactoring.

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 32 copies, one for each
thread. Reasoning about the contents of variables then requires reasoning independently about each thread, which doesn’t scale, or reasoning statically about which subset of threads might perform every assignment to every variable. Some existing program logics for CUDA (e.g. [21]) 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. The bounds given by the program logic can then be proven sound with respect to this cost semantics. The cost semantics, the analysis, and the soundness proof are formalized in the Coq Proof Assistant. We have implemented the CUDA analysis on top of Absynth [7, 28], a resource analysis tool for imperative programs. We evaluated our implementation for both precision and performance on a number of CUDA kernels derived from various sources including prior work and sample code distributed with CUDA. The evaluation shows our tool to be useful in identifying the presence and quantifying the impact of performance bottlenecks on CUDA kernels, and shows promise as a tool for novice and intermediate CUDA programmers to debug the performance of kernels.

The contributions of this paper include:

- An operational cost semantics for a core calculus for CUDA kernels that formalizes the execution of kernels on a GPU under a given resource metric
- A novel Hoare-style program logic for miniCUDA, including both qualitative and quantitative properties
- A Coq formalization of the cost semantics and soundness proofs of the program logic
- An analysis tool that can parse kernels written in a sizeable subset of CUDA C and analyze them with respect to a given resource metric
- An empirical evaluation of our analysis tools on a suite of CUDA kernels.

2 A Brief Introduction to CUDA

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

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

Threads within a block and blocks within a grid may be organized in one, two or three dimensions, which are specified when the kernel is invoked. A thread running CUDA code may access the x, y and z coordinates of its thread index using the designated identifiers threadIdx.x, threadIdx.y and threadIdx.z. CUDA also defines the 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 and not participate in executing the instruction).

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

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

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

---

\(^1\) 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.”
The implementation of addSub0 contains a number of performance bugs. First, the conditional diverges at every iteration of the loop. The reason is that every warp contains thread identifiers (threadIdx) that result in both odd and even values for the variable j. A straightforward way to fix this bug is to remove the conditional, unravel the loop and perform both the addition of an even row and the subtraction of an odd row in one loop iteration. The resulting code, shown in addSub1, does not have more parallelism than the original—the addition and subtraction are still performed sequentially—but will perform better because it greatly reduces the overhead of branching.

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

The execution of the function addSub1 is unable to coalesce accesses to B because, assuming w and h are larger than 4 and the arrays are stored in row-major order, no two threads within a warp access memory within 128 bits of each other. This is fixed by instead iterating over the rows of the matrix and handling the columns in parallel. This way, all of the memory accesses by a warp are consecutive (e.g., threads 0 through 31 might access A[0] through A[31] and B[w] through B[w+63]). The updated code is shown in the function body addSub2.

Shared Memory. In all of the kernels discussed so far, the arrays A and B reside in global memory, which is stored on the GPU and visible to all threads. CUDA also provides a separate shared memory space, which is shared only by threads within a block. Shared memory has a lower latency than global memory so we can, for example, use it to store the values of A rather than access global memory every time. In the function addSub3, we declare a shared array As and copy values of A into As before their first use.

Some care must be taken to ensure that the code of addSub3 is performant because of how shared memory is accessed. Shared memory consists of a number, generally 32, of separate banks. Separate banks may be accessed concurrently, but multiple concurrent accesses to the same bank are serialized. It is thus important to avoid "bank conflicts". Most GPUs ensure that 32 consecutive 32-bit memory reads will not result in any bank conflicts. However, if a block accesses a shared array at a stride other than 1, bank conflicts can accumulate. In addSub3, if blockDim.x is a multiple of 32, as will often be the case for a one-dimensional block, each thread in each warp will access a separate bank, minimizing bank conflicts. However, if blockDim.x is 16, for example, each bank will be accessed by two threads in each warp and shared memory throughput will decrease.

3 The miniCUDA Core Calculus

In this section, we present a core calculus, called miniCUDA, that captures the features of CUDA that are of primary interest in this paper: control flow (to allow for loops and to study the cost of divergent warps) and memory accesses. We will use this calculus to present the theory of our resource analysis for CUDA kernels.
Operands  \( o \) ::= \( x \mid p \mid c \mid \text{tid} \)
Arrays  \( A \) ::= \( G \mid S \)
Expressions  \( e \) ::= \( o \mid o \oplus o \mid A[o] \)
Statements  \( s \) ::= \( \text{skip} \mid s \mid x \leftarrow e \mid A[o] \leftarrow e \)

| if \( e \) then \( s \) else \( s \) | while \( (e) \) \( s \)

**Figure 2. Syntax of miniCUDA**

In designing the miniCUDA calculus, we have made a number of simplifying assumptions which make the presentation cleaner and more straightforward. One notable simplification is that a miniCUDA program consists of a single kernel, without its function header. In addition, we collapse the structure of thread and block indices into a single thread ID, denoted \( \text{tid} \). This loses no generality as a three-dimensional thread index can be converted in a straightforward way to a one-dimensional thread ID, given the block dimension. The resource analysis will be parametric over the block index and other parameters, and will estimate the maximum resource usage of any warp in any block.

**Syntax** The syntax of the miniCUDA calculus is presented in Figure 2. The terms of the calculus are divided into statements, which may affect control flow or the state of memory, and expressions, which do not have effects. We further distinguish operands, which consist of thread-local variables, parameters to the kernel (these include arguments passed to the kernel function as well as CUDA parameters such as the block index and warp size), constants (of type int, bool and float) and a designated variable tid. Additional expressions include \( o_1 \oplus o_2 \), which stands for an arbitrary binary operation, and array accesses \( A[o] \). The metavariable \( A \) stands for a generic array. When relevant, we use metavariabes that indicate whether the array is stored in (G)lobal or (S)hared memory. Note that subexpressions of expressions are limited to operands; more complex expressions must be broken down into binary ones by binding intermediate results to variables. This restriction simplifies reasoning about expressions without limiting expressivity.

Statements include two types of assignment: assignment to a local variable and to an array element. Statements also include conditionals and while loops. The keyword skip represents the “empty” statement. Statements may be sequenced without limiting expressivity. In addition, we collapse the variables, parameters, operators, functions and arrays. Statements do not have return values, but the judgment \( \Sigma \vdash s \) is used to indicate that \( s \) is well-formed in that all of its subexpressions have the expected type.

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

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 metric, 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 \( M^{\text{read}}(i) \leq M^{\text{read}}(j) \), and similarly for \( M^{\text{write}} \).

**Operational Semantics** We now define an operational semantics for evaluating miniCUDA 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. The evaluation judgments are parameterized by the set \( T \) of currently active threads in the warp. Its elements \( t \) are abstract thread identifiers. The function \( \text{tid}(t) \) converts such an identifier to an integer thread ID.

The operational semantics also requires a store \( \sigma \), representing the values stored in memory. The domain of \( \sigma \) is \((\text{Arrays} \times \Sigma) \cup (\text{LocalVars} \times \text{Threads})\). For an array \( A \) (regardless of whether it stored in global or shared memory), \( \sigma(A, n) \)
returns the \( n^{th} \) element of \( A \). Note that, for simplicity of presentation, we assume that out-of-bounds indices (including negative indices) map to some default value. For a local variable \( x \), \( \sigma(x, t) \) returns the value of \( x \) for thread \( t \).

Selected evaluation rules are presented in Figure 3. Some rules are omitted for space reasons and can be found in the attached supplementary material. Operands and expressions are evaluated using a high-step semantics with the judgment \( \sigma; e \downarrow_T R; C \), meaning that, under store \( \sigma \), the expression \( e \) evaluates on threads \( T \) to \( R \) with cost \( C \). The result \( R \) is a family indexed by \( T \) consisting of the result of evaluation at each thread. The rules for expressions are slightly more complex. Array accesses evaluate the operand to a set of indices and read the value from memory at each index. The determination of the cost includes evaluating \( \text{MemReads}(R) \) or \( \text{Conflicts}(R) \) on the set of indices \( R \). As discussed before, we leave these functions as parameters because their exact definitions can change across versions of CUDA and hardware implementations. As examples of these functions, we give definitions consistent with common specifications in modern CUDA implementations [29]:

\[
\text{MemReads}(R) \triangleq \left\{ (i, j) \mid \text{Dom}(R) \right\}
\]

\[
\text{Conflicts}(R) \triangleq \max_{i \in [k, 31]} \{ R(t) \equiv j \mod 32 \mid t \in \text{Dom}(R) \}
\]

Above, we assume that global reads are 128 bits in size and array elements are 32 bits. In reality, and in our implementation, \( \text{MemReads}(R) \) depends on the type of the array.

Figure 3. Selected evaluation rules.

Statements are evaluated with the judgment \( \sigma; s \downarrow_T \sigma'; C \). Unlike with operands and expressions, there is no return value; instead, evaluation results in a new state. For assignment statements, the new state comes from updating the state with the new assignment. We write \( \sigma[(x, t) \mapsto (v), t \in T] \) to indicate the state \( \sigma \) updated so that for all \( t \in T \), the binding \( (x, t) \) now maps to \( (v), t \). Array updates are written similarly. Conditionals and while loops each have three rules. If a conditional evaluates to True for all threads (SC:IfT), we simply evaluate the “if” branch with the full set of threads \( T \), and similar if all threads evaluate to False. If, however, there are non-empty sets of threads where the conditional evaluates to True and False (\( T_F \) and \( T_T \), respectively), we must evaluate both branches. We evaluate the “if” branch with \( T_F \) and the “else” branch with \( T_T \). Note that the resulting state of the “if” branch is passed to evaluation of the “else” branch; this corresponds to CUDA executing the two branches in sequence. This rule, SC:IfD, also adds the cost \( M^{\text{div}} \) of a divergent warp. The three rules for while loops similarly handle the cases in which all, some or none of the threads in \( T \) evaluate the condition to be True. The first two rules both evaluate the body under the set of threads for which the condition is true and then reevaluate the loop. Rule SC:WhileSome also indicates that we must pay \( M^{\text{div}} \) because the warp diverges.
4 Quantitative Program Logic

In this section, we present declarative rules for a Hoare-style logic that can be used to reason about the resource usage of a miniCUDA kernel for a given resource metric. The analysis for resource usage is based on the ideas of automated amortized resource analysis [16, 19]. The key idea of this analysis is to assign a non-negative numerical potential to states of computation. This potential must be sufficient to cover the cost of the following step and the potential of the next state.

The particular challenge of designing such a logic for CUDA is that each thread in a warp has a distinct local state. To keep inference tractable and scalable, we wish to reason about only one copy of each variable, but must then be careful about what exactly is meant by any predicate on a state. In particular, the potential of a state in an imperative amortized resource analysis is typically a function of the values of local variables. However, such a function is not well-defined for CUDA local variables, which have a value for each thread.

To solve this problem, we make an observation about CUDA programs: There is often a separation between local program variables which carry data (e.g., are used to store data loaded from memory or intermediate results of computation) and those that carry potential (e.g., are used as indices in for loops). To develop a sound and useful quantitative logic, it suffices to track potential for the latter set of variables, which generally hold the same value across all active threads.

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

The second component of the conditions is a potential function \(Q\), a mapping from stores and sets of variables \(X\) as described above, to non-negative rational potentials. We use the potential function to track potential through a kernel in order to analyze resource usage. If \(σ, T ⊨ X\), then \(Q_X(σ)\) refers to the potential of \(σ\) under function \(Q\), taking into account only the variables in \(X\). Formally, we require (as a property of potential functions \(Q\)) that if for all \(x ∈ X\) and \(t ∈ T\), we have \(σ_1(x, t) = σ_2(x, t)\), then \(Q_X(σ_1) = Q_X(σ_2)\). That is, \(Q\) can only take the variables in \(X\) into consideration.

For a nonnegative rational cost \(C\), we use the shorthand \(Q + C\) to denote a potential function \(Q′\) such that for all \(σ\) and \(X\), we have \(Q′_X(σ) = Q_X(σ) + C\). We write \(Q ≥ Q′\) to mean that for all \(σ\), \(T\) such that \(σ, T ⊨ P\), we have \(Q_X(σ) ≥ Q′_X(σ)\).

In this section, we leave the concrete representation of the logical condition and the potential function abstract. In Section 5, we describe our implementation, including the representation of these conditions. For now, we make the assumptions stated above, as well as that logical conditions obey the standard rules of Boolean logic. We also assume that logical conditions and potential functions are equipped with an “assignment” operation \(P′ ≔ P[x ← e]\) (resp. \(Q′ ≔ Q[x ← e]\)) such that if \(σ, T ⊨ P\) and \(σ; e ⊑ T; R; C\) then

- \(σ | (x, t) → R(t) \quad | t ∈ T\), \(T ⊨ P\)
- \(σ | x ∈ X\) and there exists \(v\) such that \(R(t) = v\) for all \(t ∈ T\), then \(Q′_X(σ) = Q_X(σ)(x, t) → R(t) \quad | t ∈ T\)

For simplicity, we also assume that the potential function depends only on the values of local variables in the store and not on the values of arrays. This is sufficient to handle the benchmarks we studied. We write \(σ; T ⊨ \{P; Q; X\}\) to mean \(σ, T ⊨ P\) and \(σ, T ⊨ X\).

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

Inference Rules Figure 4 presents the inference rules for the Hoare-style logic for resource usage of statements. The judgment for these rules is written \(\{P; Q; X\} s \{P′; Q′; X′\}\), which states that, assuming precondition \(\{P; Q; X\}\), if \(s\) terminates, the state after executing \(s\) satisfies \(\{P′; Q′; X′\}\). The most notable rules are for conditionals if \(e\) then \(s_1\) else \(s_2\), which take into account the possibility of a divergent warp. There are four cases. First (Q:If1), we can statically determine that the conditional expression \(e\) does not vary across a warp: this is expressed with the premise \(P ⇒ e\) uniq, which is shorthand for

\[∀σ, T. \ σ, T ⊨ P ⇒ ∃c. \ σ; e ⊑ T \ (c)_{t ∈ T}; C\]

That is, for any compatible store, \(e\) evaluates to a constant result family. In this case, only one branch is taken by the warp and the cost of executing the conditional is the maximum cost of executing the two branches (plus the cost \(M^d\) of...
the conditional and the cost \( C \) of evaluating the expression, which are added to the precondition). This is expressed by using \( Q' \) as the potential function in the post-condition for both branches. In this way, we require the potential to remain after executing the two branches. This can be achieved by using rule Q:Weak, which allows derivations to discard potential (this rule will be discussed in more detail later). In the next two cases (Q:If2 and Q:If3), we are able to statically determine that the conditional expression is either true or false in any compatible store (i.e., either \( P \Rightarrow e \) or \( P \Rightarrow \neg e \)), and we need only the “then” or “else” branch.

In the final case (Q:If4), we consider the possibility that the warp may diverge. In addition to accounting for the case where we must execute \( s_1 \) followed by \( s_2 \) in sequence, this rule must also subsume the three previous cases, as it is possible that we were unable to determine statically that the conditional would not diverge (i.e., we were unable to derive the preconditions of Q:If1) but the warp does not diverge at runtime. To handle both cases, we require that the precondition of \( s_2 \) is implied by both \( P \land \neg e \) and \( P_1 \) (the postcondition of \( s_1 \)), and that the postcondition of the whole conditional is implied by the individual postconditions of both branches. In addition, we remove from \( X_1 \) the set of variables possibly written to by \( s_1 \) (denoted \( W(s_1) \)), this can be determined syntactically) because if the warp diverged, variables written to by \( s_1 \) no longer have consistent values across the entire warp. We similarly remove \( W(s_2) \) from \( X_2 \).

Note that it is sound to use rule Q:If4 instead of the other rules for conditionals. However, Q:If4 would produce a conservative over-estimate of the cost by assuming a warp diverges even if it can be shown that it does not. Our inference algorithm will maximize precision by choosing the most precise rule that applies.

The rules Q:While1 and Q:While2 charge the initial evaluation of the conditional \( (M^d + C) \) to the precondition. For the body of the loop, as with other Hoare-style logics, we must derive a loop invariant: the condition \( P \) must hold at both the beginning and end of each iteration of the loop body (we additionally know that \( e \) holds at the beginning of the body). In addition, the potential after the loop body must be sufficient to “pay” \( M^d + C \) for the next check of the conditional, and still have potential \( Q \) remaining to execute the next iteration if necessary. Recall that \( Q \) is a function of the store. So this premise requires that the value of a store
element (e.g. a loop counter) change sufficiently so that the corresponding decrease in potential \(Q_X(\sigma)\) is able to pay the appropriate cost. The difference between the two rules is that Q WHILE1 assumes the warp does not diverge, so we need not pay \(M_{dx}\) and also need not remove variables assigned by the loop body from \(X\).

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

Finally, as discussed above, Q WEAK allows us to strengthen the preconditions and weaken the postconditions of a derivation. If we can execute with precondition \(\{P_1; Q_1; X\}\) and post-condition \(\{P'_1; Q'_1; X\}\), it can also execute with a precondition \(P_i\) that implies \(P_1\) and a potential function \(Q_i\) that is always greater than \(Q_1\). In addition, it can guarantee any postcondition implied by \(P'_i\) and any potential function \(Q'_i\) that is always less than \(Q'_1\). We can also take subsets of \(X\) as necessary in derivations. The rule also allows us to add a constant potential to both the pre- and post-conditions.

**Example Derivation** Figure 5 steps through a derivation for the addSub3 kernel from Section 2, with the pre- and post-conditions interspersed in red. The code is simplified to more closely resemble miniCUDA. For illustrative purposes, we consider only the costs of array accesses (writes and reads) and assume all other costs are zero. The potential annotation consists of two parts: the constant potential and a component that is proportional to the value of \(h - j\) (initially we write this as just \(h\) because \(j = 0\)). The initial constant potential is consumed by the write on line 5, which involves 8 global reads and 1 shared read. On line 6, we establish the invariant of the loop body. On line 9, we transfer \(L\) to the constant potential (this is accomplished by Rule Q WEAK). We then expend part of this on the assignment on line 9 and the rest on line 11 (which requires 9 global reads because the first index is not aligned to a 128-bit 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 \(\sigma\) and any set of threads \(T\) such that \(\sigma; T \vdash \{P; Q; X\}\), the cost of executing the program under \(\sigma\) and threads \(T\) is at most \(Q_X(\sigma)\). We first show the soundness of the resource analysis for operands and expressions.

**Lemma 1.** If \(\Sigma \vdash e : \tau\) and \(\Sigma \vdash T \sigma\) and \(P + e : C\) and \(T \vdash \sigma; T \vdash \{P; C\}\), then \(C \leq C\).

**Theorem 1.** If \(\Sigma \vdash s\) and \(\Sigma \vdash \tau\) and \(\{P; Q; X\}\) with \(P; Q; X\) and \(\sigma; T \vdash \{P; Q; X\}\) and \(\sigma; T \vdash \{P; Q; X\}\) and \(Q_X(\sigma) - C_i \geq Q_X(\sigma') \geq 0\).

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

## 5 Inference and Implementation

In this section, we discuss the implementation of the logical conditions and potential functions of Section 4 and the techniques used to automate the reasoning. The automation is based on instantiations of the boolean conditions and potential functions similar to existing work [7, 8]. We have implemented the inference algorithms as an extension to the Absynth tool [7, 28]. We begin by outlining our implementation, and then detail the instantiations of the potential annotations and logical conditions.

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

Our extension uses a modified version of Front-C² to parse CUDA, which we then lower into a representation similar to miniCUDA. Another set of transformations converts this

---

1. **Figure 5.** A derivation using the program logic. We define \(L\) to be \(2M_{spread}(1) + M_{gwrite}(8) + M_{gwrite}(9)\).

2. **Lemma 1.** If \(\Sigma \vdash e : \tau\) and \(\Sigma \vdash T \sigma\) and \(P + e : C\) and \(T \vdash \sigma; T \vdash \{P; C\}\), then \(C \leq C\).

3. **Theorem 1.** If \(\Sigma \vdash s\) and \(\Sigma \vdash \tau\) and \(\{P; Q; X\}\) with \(P; Q; X\) and \(\sigma; T \vdash \{P; Q; X\}\) and \(\sigma; T \vdash \{P; Q; X\}\) and \(Q_X(\sigma) - C_i \geq Q_X(\sigma') \geq 0\).

4. [https://github.com/BinaryAnalysisPlatform/FrontC](https://github.com/BinaryAnalysisPlatform/FrontC)
representation into IMP, a simple resource-annotated imperative language that serves as a front-end to Absynth. Part of this process is adding annotations that express the cost of each operation in the given resource metric.

We have extended the abstract interpretation pass to gather CUDA-specific information that allows us to determine the costs of array accesses and whether conditionals might diverge. This information is then used to select the most precise rules from our program logic. We describe the extended abstraction domain in the next subsection.

For the most part, we did not modify the representation of potential functions, but we briefly discuss this representation at the end of this section. In addition, we implemented a “simulated evaluation” mode that interprets the CUDA representation according to the cost semantics of Section 3. We use this mode as a “ground truth” to which we compare the predictions of our analysis in the evaluation of Section 6.

Logical Conditions The logical conditions of the declarative rules of Section 4 correspond to the abstraction domain we use in the abstract interpretation. The abstraction domain is a pair (C, M). The first component is a set of constraints of the form \( \sum_{x \in \text{var}} k_x x + k \leq 0 \), where \( k_x, k \in \mathbb{N} \). These form a constraint system on the runtime values of variables with 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 that tracks its dependence on tid and one that tracks its constant component. We write \( M_{\text{id}}(x) \) for the first projection and \( M_{\text{const}}(x) \) for the second projection. 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 \( M_{\text{id}}(x) = 0 \), then the value of \( x \) is guaranteed to be constant across threads. As another example, if \( M_{\text{id}}(x) = 1 \), then \( x = \text{tid} + c \), where \( c \) does not depend on the thread, and so the array access \( A[a] \) 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.

---

\[ \text{MemReads}(R) \triangleq \left\lvert \left\{ \left\lfloor \frac{i}{m} \right\rfloor \mid (i)_t \in R \right\} \right\rceil \]

\[ \text{Conflicts}(R) \triangleq \max_{j \in [0, B - 1]} \{ R(t) \equiv j \mod B \mid t \in \text{Dom}(R) \} \]

Theorem 2 formalizes and proves the soundness of a bound on \( \text{MemReads}(x) \) given abstract information about \( x \) (for a constant \( c \), \( \text{MemReads}(c) = 1 \), so only variables require a non-trivial analysis).

**Theorem 2.** If \( M_{\text{id}}(x) = k \) and \( C \Rightarrow \text{tid} \in [t, t'] \) and \( \sigma, T \models (C, M) \) and \( \sigma; x \downarrow_T R; C \) then \( \text{MemReads}(R) \leq \left\lfloor \frac{k(t' - t)}{m} \right\rfloor + 1 \).

**Proof.** By the definition of \( \sigma, T \models (C, M) \), we have \( T \subset [t, t'] \).

Let \( a = \min_{t \in T} R(t) \) and \( b = \max_{t \in T} R(t) \). We have

\[ \text{MemReads}(R) \leq \left\lfloor \frac{b - a}{m} \right\rfloor + 1 \leq \left\lfloor \frac{t' - t}{m} \right\rfloor + 1 \]

\[ \square \]

Theorem 3 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 \( \text{Conflicts}(o) = |T| \). The proof relies on Lemma 2, a standalone result about modular arithmetic, which we prove in the supplementary material.

**Theorem 3.** If \( M_{\text{id}}(x) = k > 0 \) and \( C \Rightarrow \text{tid} \in [t, t'] \) and \( \sigma, T \models (C, M) \) and \( \sigma; x \downarrow_T R; C \) then

\[ \text{Conflicts}(R) \leq \left\lfloor \frac{t' - t}{\min(t' - t, B \cdot \gcd(k, B))} \right\rfloor \]

**Proof.** Let \( t_0 \in T \). By the definition of \( \sigma, T \models (C, M) \), we have \( \text{Tid}(t_0) \in [t, t'] \). We have \( R(t_0) = kt_0 + C \). Let \( R' = (kt \mod B)_{t \in T} \). The accesses in \( R \) access banks from \( R' \) at uniform stride, and so the maximum number of times any such bank is accessed in \( R \) is \( \left\lfloor \frac{t' - t}{\min(t' - t, B \cdot \gcd(k, B))} \right\rfloor \). The result follows from Lemma 2.

**Lemma 2.** Let \( k, m, n, a \in \mathbb{N} \) and \( m \leq n \). Then \( \left\lfloor \frac{i - a \mod n}{i \in \{k, \ldots, k + m - 1\}} \right\rfloor = \min(m, \frac{n}{\gcd(a, n)}) \).

Potential functions Our implementation of potential functions is taken largely from prior work on AAARA for imperative programs [7, 28]. We instantiate a potential function \( Q \) as a linear combination of a fixed set \( I \) of base functions from stores to rational costs, each depending on a portion of the state. A designated base function \( b_0 \) is the constant function and tracks constant potential. For each program, we select a set of \( N \) base functions, plus the constant function, noted \( b_0, b_1, \ldots, b_N \), that capture the portions of the state relevant to calculating potential. A potential function \( Q \) is then a linear combination of the selected base functions:

\[ Q(\sigma) = q_0 + \sum_{i=1}^{N} a_i b_i(\sigma) \]
In the analysis we use, base functions are generated by the following grammar:

\[
M ::= 1 \mid x \cdot M \cdot M \mid ||P, P||
\]

\[
P ::= k \cdot M \mid P + P
\]

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 [7].

6 Evaluation

We evaluated the range and precision of our analysis on a set of benchmarks drawn from various sources. Table 1 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, written entirely by us or derived from previous work). 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. 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 several of our own modifications to it: one which computes on \(8 \times 8\) blocks (matMul8), one which deliberately introduces a number of performance bugs (matMulBad), and one (matMulTrans) which transposes one of the input matrices in a (as it happens, misguided) attempt to improve shared memory performance. 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, so we performed slight manual refactoring on these examples (kernels reduce2a and reduce3a) so our algorithm can derive bounds for them. We also include the original versions. The benchmarks BRDIS and BRDIS-OPT were written by the authors based on outlines of synthetic examples from Han and Abdelrahman [15] of kernels before and after (respectively) a transformation they proposed to reduce the impacts of divergent warps. Finally, we include the examples from Section 2 (addSubN).

We analyzed each benchmark under 3 resource metrics:

- divwarp: Counts the number of times a warp diverges (i.e., \(M^{\text{div}} = 1\)).
- shared: Counts the cost of shared memory reads and writes (i.e., \(M^{\text{read}}(n) = M^{\text{write}}(n) = n\)).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>LoC</th>
<th>Params.</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>matMul</td>
<td>SDK</td>
<td>26</td>
<td>wA, wB</td>
<td>32 \times 1</td>
</tr>
<tr>
<td>matMul8</td>
<td>SDK</td>
<td>26</td>
<td>wA, wB</td>
<td>8 \times 8</td>
</tr>
<tr>
<td>matMulBad</td>
<td>SDK</td>
<td>26</td>
<td>wA, wB</td>
<td>32 \times 1</td>
</tr>
<tr>
<td>matMulTrans</td>
<td>SDK</td>
<td>26</td>
<td>wA, wB</td>
<td>8 \times 8</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>SDK</td>
<td>78</td>
<td>N</td>
<td>32 \times 1</td>
</tr>
<tr>
<td>vectorAdd</td>
<td>SDK</td>
<td>5</td>
<td>–</td>
<td>32 \times 1</td>
</tr>
<tr>
<td>reduceN</td>
<td>SDK</td>
<td>14–18</td>
<td>–</td>
<td>32 \times 1</td>
</tr>
<tr>
<td>seam</td>
<td>Us</td>
<td>43</td>
<td>h, w</td>
<td>32 \times 1</td>
</tr>
<tr>
<td>histogram256</td>
<td>SDK</td>
<td>19</td>
<td>N</td>
<td>32 \times 1</td>
</tr>
<tr>
<td>BRDIS</td>
<td>[15]*</td>
<td>31</td>
<td>M, N</td>
<td>32 \times 1</td>
</tr>
<tr>
<td>BRDIS-OPT</td>
<td>[15]*</td>
<td>29</td>
<td>M, N</td>
<td>32 \times 1</td>
</tr>
<tr>
<td>addSubN</td>
<td>Us</td>
<td>7–14</td>
<td>h, w</td>
<td>32 \times 1</td>
</tr>
</tbody>
</table>

Table 1. The benchmark suite used for our experiments.

Us = benchmarks written by authors.

\(X^*\) = benchmarks derived from \(X\) by authors.

Figure 6. Our inferred cost estimates (blue line) vs. simulated worst-case costs (red stars) for various input sizes. **Top:** matMul, shared; **Bottom:** BRDIS, steps.
We note, however, that despite any imprecisions, more efficient, but this will likely depend on hardware details). The marking — indicates that our analysis reported that it failed to find a bound. The timing results show that our analysis is quite efficient for almost all benchmarks (with analysis times usually under 1 second and never more than 5 minutes). The performance of our algorithm is harmed most by nesting of divergent conditionals and loops, as in the benchmarks seam, BRDIS and BRDIS-OPT.

We also ran most benchmarks in simulated evaluation mode with a set Inputs of inputs of different sizes and report the average error

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

neglecting inputs that would cause a division by zero. For some benchmarks, marked with ** in the table, evaluation cost depends on the contents of memory, which our simulator does not currently model. Figure 6 plots our predicted cost versus the simulated worst-case for two representative benchmarks. In many cases, the biggest source of imprecision in our analysis is that CUDA kernels often perform operations in groups of, e.g., 32 elements per loop iteration resulting in bounds of the form \( \frac{N}{32} \). The Absynth tool can only produce polynomial bounds and so must approximate this bound by \( \frac{N + 31}{32} \), which is the tightest possible polynomial bound. In other cases, additional imprecisions are introduced, usually because our abstraction domain is insufficiently complex to show, e.g., that memory accesses are properly aligned. We note, however, that despite any imprecisions, 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., matrixMul, reduce3a, addSub3). Both of these properties 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.

The experiments also show the utility of our parametric tool over tools that merely identify one type of performance bug. Often, there is a tradeoff between two performance bottlenecks. For example, reduce3a has worse global memory performance than reduce2a, but better shared memory performance. By combining these into a metric that takes account the relative cost of each operation, we can explore the tradeoff (in the case of the steps metric, reduce2a is more efficient, but this will likely depend on hardware details). As another example, the BRDIS-OPT kernel was designed

Table 2. Inferred evaluation cost, average error and analysis time. *For vectorAdd/divwarps, the correct value is 0 for all inputs, so no error is reported. **Worst-case inputs could not be generated for these benchmarks.
to reduce the impact of divergent warps over BRDIS using a transformation called branch distribution. Table 2 makes clear that the transformation actually increases the number of divergences (because it splits divergent code into two sections). However, the total amount of code that must execute sequentially is decreased, as evidenced by the steps metric.

7 Related Work

Resource Bound Analysis. There exist many static analyses and program logics that (automatically or manually) derive sound performance information such as provably-correct worst-case bounds for imperative [7, 14, 20, 34] and functional [9, 13, 17, 19, 22, 32] programs. However, there are very few tools for parallel [18] and concurrent [1, 10] execution and there are no such tools that take into account the aforementioned CUDA-specific performance bottlenecks.

Most closely related to our work is automatic amortized analysis (AARA) for imperative programs and quantitative program logics. Carbonneaux et al. [6] introduced the first imperative AARA in form of a program logic for verifying stack bounds for C programs. The technique has then been automated [7, 8] using templates and LP solving and applied to probabilistic programs [28]. As discussed, a main innovation of this work are the development of an AARA for CUDA code. Previous work on imperative AARA cannot analyze parallel executions nor CUDA specific memory-access cost.

Analysis of CUDA Code. Given its importance in fields such as machine learning and high-performance computing, CUDA has gained a fair amount of attention in the program analysis literature in recent years. There exist a number of static [23, 31, 36] and dynamic [4, 12, 30, 35] analyses for verifying certain properties of CUDA programs, but much of this work focused on functional properties, e.g., freedom from data races. Wu et al. [35] investigate several classes of bugs, one of which is “non-optimal implementation”; this class includes several types of performance problems. They don’t give examples of kernels with non-optimal implementations, and don’t specify whether or how their dynamic analysis detects such bugs. PUG [23] and Boyer et al. [4] focus primarily on detecting data races but both demonstrate an extension of their race detectors designed to detect bank conflicts, albeit with somewhat imprecise results. Kojima and Igarashi [21] present a Hoare logic for proving functional properties of CUDA kernels. Their logic does not consider quantitative properties and, unlike our program logic, requires explicit reasoning about the sets of active threads at each program point, which poses problems for designing an efficient automated inference engine.

GKLEE [24] is an analysis for CUDA kernels based on concolic execution, and targets both functional errors and performance errors (including warp divergence, non-coalesced memory accesses and shared bank conflicts). GKLEE requires some user annotations in order to perform its analysis. Alur et al. [2] and Singhania [33] have developed several static analyses for performance properties of CUDA programs, including uncoalesced memory accesses. Their analysis for detecting uncoalesced memory accesses uses abstract interpretation with an abstract domain similar to ours but simpler (in our notation, it only tracks $M_{id}(x)$ and only considers the values 0, 1, and -1 for it). Their work does not address shared bank conflicts or divergent warps. Moreover, they developed a separate analysis for each type of performance bug. In this work, we present a general analysis that detects and quantifies several different types of performance bugs.

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

Other work has focused on quantifying or mitigating, but not detecting, performance errors. Bialas and Strzelecki [3] use simple, tunable kernels to experimentally quantify the impact of warp divergence on performance using different GPUs. Their findings show that there is a nontrivial cost associated with a divergent warp even if the divergence involves some threads simply being inactive (e.g. threads exiting a loop early or a conditional with no “else” branch). This finding has shaped our thinking on the cost of divergent warps. Han and Abdelrahman [15] present two program transformations that lessen the performance impact of warp divergence; they experimentally analyze the benefit of these optimizations but do not have a way of statically identifying whether a kernel contains a potential for divergent warps and/or could benefit from their transformations.

8 Conclusion

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

This work has taken the first step toward automated static analysis tools for analyzing and improving performance of CUDA kernels. In the future, we plan to extend this logic to handle more advanced features of CUDA such as synchronization and dynamic parallelism by combining the lockstep execution analysis of this paper with dependence-graph cost models for dynamic parallelism (e.g., [5, 11, 25–27]). These features are largely orthogonal to the performance bottlenecks we have considered in this paper, but integrating them.
into our analysis will bring our model closer to predicting the real-time execution characteristics of kernels.

References


ACM, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/1964179.1964184


