A Modular Static Cost Analysis for GPU Warp-Level Parallelism

GREGORY BLIKE, University of Massachusetts Boston, USA
HANNAH ZICARELLI, University of Massachusetts Boston, USA
UDAYA SATHIYAMOORTHY, University of Massachusetts Boston, USA
JULIEN LANGE, Royal Holloway, University of London, UK
TIAGO COGUMBREIRO, University of Massachusetts Boston, USA

Graphics Processing Units (GPUs) are the accelerator of choice for performance-critical applications, yet optimizing for performance requires mastery of the complex interactions between its memory architecture and its execution model. Existing static analysis tools for GPU kernels either identify performance bugs without quantifying costs or cannot handle thread-divergent control flow, leading to significant over-approximations. We present the first static relational-cost analysis for GPU warp-level parallelism that can give exact bounds even in the presence of thread divergence. Our analysis is general and flexible, as it is parametric on the resource metric (uncoalesced accesses, bank conflicts) and on the cost relation (=, ≤, ≥). We establish a soundness theorem for our technique, provide mechanized proofs in Rocq and implement our theory in a tool called Pico. In a reproducibility experiment, Pico produced the tightest bounds in every input, outperforming the state-of-the-art tool RaCUDA in 10 kernels (1.7× better), while RaCUDA produced 4 incorrect bounds and crashed on 2 kernels. In an experiment to measure the accuracy of Pico, we studied the impact of thread-divergence in control-flow in a dataset of 226 kernels. We found that at least 75.3% of conditionals and 85.4% of loops can be captured exactly, without introducing approximation.

CCS Concepts: • **Theory of computation → Parallel computing models**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: cost analysis, relational verification, GPU programming

1 Introduction

Graphics Processing Units (GPUs) have become essential accelerators for performance-critical applications including machine learning [48], scientific simulations [23, 55], and high-performance computing [54]. Extracting optimal performance from GPUs requires navigating a complex landscape of interacting optimization requirements that span from a deep understanding of the memory architecture to the execution model of GPU devices. To maximize performance threads must [22]: be continuously busy to maximize parallelism, perform sequential and contiguous accesses (coalesced data-movements) to minimize memory latency [7, 17, 59], avoid bank-conflicts [27, 32, 58, 60], and minimize divergent behavior [10, 52, 53].

Symbolic, dynamic analysis [15, 30, 31, 39], and profiling tools, such as Nvidia's nSight, can identify performance bottlenecks for a specific input and run, but cannot provide guarantees about

Authors' Contact Information: Gregory Blike, gregory.blike001@umb.edu, University of Massachusetts Boston, Boston, Massachusetts, USA; Hannah Zicarelli, University of Massachusetts Boston, Boston, Massachusetts, USA; Udaya Sathiyamoorthy, University of Massachusetts Boston, Boston, Massachusetts, USA; Julien Lange, Julien.Lange@rhul.ac.uk, Royal Holloway, University of London, UK; Tiago Cogumbreiro, Tiago.Cogumbreiro@umb.edu, University of Massachusetts Boston, Boston, Massachusetts, USA.

worst-case behavior or ensure the absence of performance bugs across all possible inputs. For safety-critical applications such as autonomous vehicles and real-time machine learning systems, static worst-case bounds on resource usage of GPU programs (known as kernels) [33] are essential to ensure system reliability and predictable performance. There is a need for static analysis approaches that can provide formal guarantees about GPU program performance.

Despite this clear need, most existing static analysis tools give limited feedback to users. PUG [37, 38] and GPUDrano [1, 2] can only identify the presence of performance bugs without quantifying their total cost. RaCUDA [45, 46] quantifies GPU kernel costs, but is unable to account for thread-divergence patterns that give rise to many GPU performance bottlenecks [10, 52, 53]. Of special interest to GPU programmers [18] are bank conflicts and uncoalesced accesses, that turn a single memory operation (transaction) into multiple ones. Crucially, ignoring thread-divergence can produce approximation errors that compound multiplicatively with the nesting of loops. To understand why, we briefly explain thread-divergence. In GPUs, threads execute in groups called *warps*, also known as wavefronts, where all threads in a warp must execute the same instruction simultaneously (in lockstep). Threads can be suspended according to the test conditions of a branch, so when different threads of the warp evaluate the test differently, then the warp executes both branches, this is known as *thread divergence*. RaCUDA cannot analyze thread-divergent loops (rejecting such programs entirely) and ignores the effect of thread-divergent conditionals, causing it to overestimate costs.

We propose Pico, the first cost analysis for GPU kernels that derives *exact costs* with support for thread divergence. Our analysis is modular by being parametric on a resource metric (*e.g.*, uncoalesced accesses, bank conflicts) to target different GPU performance bottlenecks, and on a cost relation (*e.g.*, $=$, $\leq$, $\geq$) to enable exact costs, over-approximations, and under-approximations. Symbolic costs can depend on kernel inputs, and resource metrics can depend on exactly which threads are active, directly addressing thread-divergence challenges. Pico translates a GPU kernel into a sequential program whose resource usage can be quantified symbolically without approximation. Our technique is a variation on relational cost analysis [16, 49–51], which establishes bounds on cost relationships between programs, achieving greater precision than non-relational analysis by exploiting shared program structures.

This paper presents the following contributions:

- §3 A **dynamic cost model** of GPU warp execution, parametric on a cost metric. The model provides the foundation for relational-cost analysis between GPU and sequential models.
- §4 The first **static relational-cost analysis** supporting thread-divergent control flow. This analysis is parametric on the cost relation, providing a powerful framework to reason about exact costs, over-approximations, and under-approximations.
- §5 A **fully mechanized meta-theory** for our parametric analysis. We prove the static analysis sound relative to the dynamic model. We state our result for any cost relation and cost metric, which then precisely captures how cost metrics and loops are analyzed, which can be leveraged in the implementation.
- §6 An **implementation of our cost analysis**, called Pico, for bank conflicts and uncoalesced accesses, which we formally define in Section 6.4. Our tool leverages the relational analysis to separate capturing the cost (as a sequential program) from reducing the cost to a symbolic arithmetic expression. Our implementation features a modular design with alternative ways to calculate the symbolic cost: our bespoke algorithm using Maxima [40], or translators to Absynth [47], CoFloCo [25], and KoAT [26].
- §7 A **comprehensive evaluation** of Pico, verifying the precision and applicability of our model. A first experiment reproduces an evaluation of 25 kernels from [45] and shows a

significant improvement over RaCUDA in 10 kernels. Using Pico, we identified two unsound results given by RaCUDA. A second experiment demonstrates the precision of our approach on 226 kernels from a well-known dataset [4]. We measured the effect of thread-divergence on the accuracy of our analysis wrt. control-flow. Pico captures precisely at least 75.3% of conditionals and at least 85.4% of loops.

In the following section, we give an overview our approach and contributions. Section 8 discusses related work and positions our contributions within the broader landscape of GPU analysis techniques. Section 9 summarizes our contributions and outlines directions for future work. The Rocq mechanization of the theory and are included in the paper's artifact [13]. Our tool is maintained in the Faial project [56] and the Rocq mechanization is maintained in [57].

2 Overview

This section demonstrates our relational-cost analysis for GPU programs with thread-divergent control flow. We show how our approach achieves exact bounds where existing tools over-approximate, and how exact metrics can be composed to derive tighter bounds for complex cost analyses.

Background: memory access protocols. In this paper, we use Memory Access Protocols [19, 20, 42, 43] as an abstraction over GPU kernels, that we formally define in the following section. Practically, protocols serve as an intermediate representation for static analysis. Theoretically, protocols are a form of behavioral type codifying how threads interact with memory. Henceforth, we refer to kernels as protocols.

Background: thread-divergent behavior. To understand our running examples, we explain the execution of protocol $\mathsf{for}\ \mathsf{x} \in 0..\mathsf{tid}\ \{\mathsf{A}[0]\}$. The upper bound of the loop is a special variable tid, which corresponds to a numeric identifier that uniquely identifies each thread with an integer (ranged over by meta-variable $i \in \mathbb{N}$), so for thread i the loop range is $0..i$. This example depicts *thread-divergence, i.e.*, when the execution of a warp proceeds with suspended (inactive) threads. Since every thread in a warp executes as a unit in lockstep, when the warp executes this loop, an increasing number of threads becomes inactive as the warp executes each iteration, until all threads are inactive, which makes the warp terminate looping. We represent the activity state of all threads as a vector of Booleans, where t is true and f is false. Let W be the thread count in a warp. For instance, with $\mathsf{W} = 4$ threads, at iteration 0 all threads are active $(\mathsf{t}, \mathsf{t}, \mathsf{t}, \mathsf{t})$, at iteration 1 thread 0 becomes inactive $(\mathsf{f}, \mathsf{t}, \mathsf{t}, \mathsf{t})$, and so on. This thread divergence creates complex cost patterns that our analysis targets.

Example 2.1 (Thread-divergent loop bounds). To highlight the expressiveness of our formalism in contrast to a worst-case cost analysis, we show that the cost of a protocol with a thread-divergent loop equals that of a sequential program, (1) by reasoning about the number of iterations of both loops, and (2) by reasoning about the metric locally. In this example, we will consider a cost metric that counts how many threads are active when accessing an array $\mathsf{A}[0]$. Let $\mathsf{tid} = (0, 1, 2, \ldots, \mathsf{W} - 1)$ represent a unique thread identifier per thread.

$$\emptyset; \mathsf{t} \vdash (\mathsf{for}\ \mathsf{x} \in 0..\mathsf{tid}\ \{\mathsf{A}[0]\}) = (\mathsf{for}\ \mathsf{x} \in 0..\mathsf{W} - 1\ \{\mathsf{tick}(\mathsf{W} - \mathsf{x})\}) \tag{1}$$

The judgment states that under the empty typing context $\emptyset; \mathsf{t}$, the protocol (left side) has the same cost (=) as the sequential program (right side). The sequential program executes a loop from 0 to $\mathsf{W} - 1$, where at each iteration x, instruction tick consumes exactly $\mathsf{W} - \mathsf{x}$ resources.

Let us verify the correspondence between the number of active threads (our metric) and the scalar expression $\mathsf{W} - \mathsf{x}$ (the argument of tick). For instance, when $\mathsf{W} = 4$, then we would have the 4 iterations for both ranges. In iteration 0, all 4-threads are active $(\mathsf{t}, \mathsf{t}, \mathsf{t}, \mathsf{t})$, and we have $\mathsf{W} - \mathsf{x} = 4 - 0$.

In iteration 1, thread 0 is inactive (3 active) and we have $4 - 1 = 3$. In iteration 2, threads 0 and 1 are inactive and we have $4 - 2 = 2$. In the last iteration, only thread 3 is active and we have $4 - 3 = 1$. Thus, we can conclude that there are always $\mathsf{W} - \mathsf{x}$ threads active. A worst-case analysis like RaCUDA provides only a constant bound W, whereas our relational cost analysis can capture the precise relationship between iteration number and active thread count ($\mathsf{W} - \mathsf{x}$).

Pico implements our relational cost as follows. Given a protocol $\mathsf{for}\ \mathsf{x} \in 0..\mathsf{tid}\ \{\mathsf{A}[0]\}$, Pico uses our type system to derive a program $\mathsf{for}\ \mathsf{x} \in 0..\mathsf{W} - 1\ \{\mathsf{tick}(\mathsf{W} - \mathsf{x})\}$, outputs a symbolic cost of $\sum_{0 \le \mathsf{x} \le \mathsf{W}-1} \mathsf{W} - \mathsf{x}$, and then simplifies such cost to $(\mathsf{W}^2 + \mathsf{W})/2$ using Maxima (528 when $\mathsf{W} = 32$). A worst case cost analysis would yield a symbolic cost of W^2 (1024 when $\mathsf{W} = 32$).

Our ability to statically analyze thread-divergent loops addresses two fundamental limitations in the state of the art [45, 46]. Existing static analysis tools cannot handle thread-divergent loops, forcing them to reject such programs entirely. Additionally, existing theoretical frameworks require loop invariants that tools cannot automatically provide, whereas our analysis does not.

Key takeaways: *This example showcases two novel contributions of our formalism: support for thread-divergent loop ranges, and a metric analysis that expresses precise (=) costs symbolically* ($\mathsf{W} - \mathsf{x}$).

Example 2.2 (Kernel addS_0*).* We show a practical application of our approach, by studying the cost of a kernel [46, addS_0] that alternately adds and subtracts from rows of a matrix. In this example, we establish an upper bound program for the *uncoalesced accesses metric*. When threads access non-adjacent memory locations, the GPU cannot combine the accesses into a single memory transaction, and issues multiple transactions instead, which are bounded by the total number of active threads. Test $\mathsf{tid} \mathbin{\%} 2 = 0$ is thread-divergent, because half of the threads evaluate the test as true (even thread identifiers), and the other half evaluates the test to false (odd thread identifiers).

$$\emptyset; \mathsf{t} \vdash\ (\mathsf{for}\ \mathsf{x} \in 0..i\ \{\mathsf{if}\ (\mathsf{tid} \mathbin{\%} 2 = 0)\ \{\mathsf{A}[\mathsf{tid} \times i + \mathsf{x}]\}\})\ \le\ (\mathsf{for}\ \mathsf{x} \in 0..i\ \{\mathsf{tick}(\lfloor \mathsf{W}/2 \rfloor)\}) \tag{2}$$

We observe that the loop of the protocol (left-hand side) is *thread-uniform*, *i.e.*, every thread evaluates the loop bounds equally, and the same loop range is used in the program (right-hand side). Thus, variable x admits the *same* values in either side of the relation. The judgment also establishes that uncoalesced accesses never exceed $\lfloor \mathsf{W}/2 \rfloor$ per iteration, since the number of uncoalesced accesses are bounded by the number of active threads, and the conditional $\mathsf{tid} \mathbin{\%} 2 = 0$ ensures exactly half the threads are active. Our approach yields tighter bounds than the state of the art's worst-case analysis [45, 46], which assumes all $\mathsf{W} = 32$ threads create uncoalesced accesses.

Key takeaways: *This example showcases two novel contributions of our formalism: support for thread-divergent conditionals (*$\mathsf{tid} \mathbin{\%} 2 = 0$*); and our cost-analysis generalizes the cost relation (here* $\le$*) with support for exact (=), upper (*$\le$*), and lower (*$\ge$*) relational bounds, among others.*

3 Dynamic Cost Model

To establish the soundness of our static analysis, we need to introduce dynamic evaluation so that we can relate the cost of a protocol with a sequential program. In this section we introduce a dynamic analysis to establish cost relations between executing protocols and statements. We define dynamic semantics for both domains, then formally relate their costs: first giving semantics to vector-based expressions (Section 3.1), then providing dynamic semantics for warps in Memory Access Protocols (Section 3.2) and sequential Resource Calculus statements (Section 3.3), and finally introducing a dynamic relational-cost judgment that formally relates costs between protocols and statements (Section 3.4).

3.1 Expressions of Vectors

Figure 1 gives the syntax and semantics of expressions that yield vectors. In warp semantics, since each thread may have a different view over its own local data, we represent the warp's data as

Syntax

$$
\begin{array}{llllllllll}
\mathbb{N}_{>0} & \ni & \mathsf{W} & \textit{(thread count per warp)} & & \sigma & \ni & \mathcal{V} \to \mathbb{N}^{\mathsf{W}} & & \\
\mathbb{N} & \ni & i, j, k & ::= & 0 \mid 1 \mid \cdots & \mathcal{E}^{\mathsf{W}} & \ni & n, m, o & ::= & i \mid x \mid \mathsf{tid} \mid n \star n \\
\mathcal{B} & \ni & a, b & ::= & \mathsf{t} \mid \mathsf{f} & \mathcal{C}^{\mathsf{W}} & \ni & c, d & ::= & b \mid n \diamond n \mid c \circ c \\
\mathbb{N}^{\mathsf{W}} & \ni & I, J, K & & & & & r & ::= & n..n \\
\mathcal{B}^{\mathsf{W}} & \ni & C, B & & & \mathbb{N}^{\mathsf{W}} & \ni & I_{\mathbb{N}} & \overset{\text{def}}{=} & (0, 1, \ldots, \mathsf{W}-1)
\end{array}
$$

Big-step semantics for numeric expressions n $\boxed{\langle n, \sigma \rangle \Downarrow I}$

$$
\langle i, \sigma \rangle \Downarrow i^{\mathsf{W}} \qquad \langle x, \sigma \rangle \Downarrow \sigma(x) \qquad \langle \mathsf{tid}, \sigma \rangle \Downarrow I_{\mathbb{N}} \qquad \frac{\langle n, \sigma \rangle \Downarrow I \qquad \langle m, \sigma \rangle \Downarrow J}{\langle n \star m, \sigma \rangle \Downarrow I \star J}
$$

Big-step semantics for boolean expressions c and ranges $n..n$ $\boxed{\langle c, \sigma \rangle \Downarrow B}$ $\boxed{\langle r, \sigma \rangle \Downarrow (I, B)}$

$$
\langle b, \sigma \rangle \Downarrow b^{\mathsf{W}} \qquad \frac{\langle n, \sigma \rangle \Downarrow I \qquad \langle m, \sigma \rangle \Downarrow J}{\langle n \diamond m, \sigma \rangle \Downarrow I \diamond J}
$$

$$
\frac{\langle c, \sigma \rangle \Downarrow C \qquad \langle d, \sigma \rangle \Downarrow B}{\langle c \circ d, \sigma \rangle \Downarrow C \circ B} \qquad \frac{\langle n, \sigma \rangle \Downarrow I \qquad \langle n \leq m, \sigma \rangle \Downarrow B}{\langle n..m, \sigma \rangle \Downarrow (I, B)}
$$

Fig. 1. Syntax and semantics of expressions.

a vector and define the evaluation of the same expression for each thread in a warp. Expression evaluation outputs a vector of values, where the vector's components represent the value that each respective thread evaluated.

Numbers, sets, and vectors. Let i, j, k be meta-variables over non-negative integers picked from the set $\mathbb{N}$. Let $\star$ be a meta-variable over the usual arithmetic operators $\{+, \times, \bmod\}$. We define a *vector* X of dimension k as a sequence of k components α_i, written as $(\alpha_1, \ldots, \alpha_k)$. Let $|\mathcal{X}|$ (resp. $|X|$) denote the cardinal of a set $\mathcal{X}$ (resp. vector X). We write $X(i) = \alpha_i$ if $X = (\alpha_1, \ldots, \alpha_k)$ and $0 \leq i < k$. We write $\alpha \in X$ if $X(i) = \alpha$ for some $0 \leq i < |X|$. We say that a vector X is uniform, notation $\mathrm{unif}(X)$, when $X(i) = X(j)$ for all $0 \leq i < k$ and $0 \leq j < k$. For instance, vector $(2, 2, 2, 2)$ is uniform, and vector $(0, 1, 2, 3)$ is non-uniform. We define a *componentwise operation* f on vectors as $f(X, X')(i) = f(X(i), X'(i))$ for any vectors X and X'. For any α, we write α^k to denote vector X of length k where $X(i) = \alpha$ for any $0 \leq i < k$. For instance, $0^4 = (0, 0, 0, 0)$. Trivially, we have that $\mathrm{unif}(\alpha^k)$ for any k.

Numeric vectors. We say that a vector is *numeric* when its components are all in $\mathbb{N}$. Let I, J, K be meta-variables over numeric vectors of length W. Vector $I_{\mathbb{N}}$ represents a vector of the first W numbers. We lift every operation in $\star$ componentwise. For instance, the following equality holds.

$$
(2, 2, 2, 2) \times (0, 1, 2, 3) = (2 \times 0, 2 \times 1, 2 \times 2, 2 \times 3) = (0, 2, 4, 6) \tag{3}
$$

Numeric expressions. A numeric expression n evaluates down to a numeric vector I, where the components represent the value of each thread. A numeric expression $n \in \mathcal{E}^{\mathsf{W}}$ has dimension $\mathsf{W} \in \mathbb{N}$. A numeric expression is either a numeric vector I of length W, a number i, a loop variable x, a unique thread identifier tid, or a binary operation $n \star m$. Figure 1 (middle) gives the operational big-step semantics, denoted by $\langle n, \sigma \rangle \Downarrow I$, where $n \in \mathcal{E}^{\mathsf{W}}$ is a numeric expression and σ is an environment that maps (loop) variables to numeric vectors, the result is a numeric vector I of length W.

A number i evaluates down to a uniform vector i^{W}. Since $\mathrm{unif}(i^{\mathsf{W}})$, then all threads have a uniform view over the evaluation of i. We evaluate tid to a vector that maps each thread identifier to itself. Finally, we lift the binary operation $\star$ for numeric expressions.

For instance, let $\mathsf{W} = 4$. We now show that expression $2 \times \mathsf{tid}$ evaluates to vector $2^{\mathsf{W}} \times I_{\mathbb{N}} = (0, 2, 4, 6)$, as shown below. We often omit the multiplication sign in numeric expressions when it is not ambiguous.

$$\frac{\langle 2, \emptyset\rangle \Downarrow 2^{\mathsf{W}} \qquad \langle \mathsf{tid}, \emptyset\rangle \Downarrow I_{\mathbb{N}} \quad (3)}{\langle 2 \times \mathsf{tid}, \emptyset\rangle \Downarrow 2^{\mathsf{W}} \times I_{\mathbb{N}}} \tag{4}$$

Booleans and boolean vectors. Let a, b be meta-variables over booleans $\mathcal{B} = \{\mathsf{t}, \mathsf{f}\}$. We say that a vector is *boolean* when its components are picked from $\mathcal{B}$. Let $\mathrm{count}(B, b)$ return the number of occurrences of b in vector B. We may also omit the parenthesis and commas when denoting vectors of booleans, for instance we write fttt for $(\mathsf{f}, \mathsf{t}, \mathsf{t}, \mathsf{t})$. For instance, $\mathrm{count}(\mathsf{fttt}, \mathsf{t}) = 3$. Let $\diamond$ be a meta-variable over the set of usual relational operators $\{<, >, \le, \ge, =\}$. Let $\circ$ be a meta-variable over the set of usual boolean operators $\{\wedge, \vee\}$. We lift $\diamond$ and $\circ$ componentwise.

$$\big((1, 1, 1, 1) \le (0, 1, 2, 3)\big) = (1 \le 0, 1 \le 1, 1 \le 2, 1 \le 3) = \mathsf{fttt}$$

Boolean expressions. The set of boolean expressions C^{W} is parameterized by the number of dimensions W, similar to the set of numeric expressions. A boolean expression $c \in C^{\mathsf{W}}$ is either a boolean vector B, a boolean literal b, a relational operator $\diamond$ over numeric expressions, or a boolean operator $\circ$ over boolean expressions. Figure 1 (bottom) gives a big-step operational semantics for boolean expressions executing under warp semantics with judgment $\langle c, \sigma\rangle \Downarrow B$, where an expression c is executed by W threads down to a boolean vector B of length W. A boolean vector B evaluates down to itself, as it is a value. A boolean literal b evaluates down to a uniform vector with a copy of b for each thread. The semantics of relational and boolean operators are standard.

Range expressions. A range $r ::= n..m$, used in loops, declares the lower bound n and the upper bound m of an iteration space. Ranges are inclusive in the upper and lower bounds. Judgment $\langle r, \sigma\rangle \Downarrow (I, B)$ defines an evaluation of range r down to the value of the lower bound of the range I and whether the range is nonempty ($n \le m$, where $r = n..m$). Hence, range $0..0$ evaluates to pair $(0^{\mathsf{W}}, \mathsf{t}^{\mathsf{W}})$ where vector 0^{W} is the first value of the range and t^{W} represents that the range is nonempty (*i.e.*, not terminated) for all threads. Range $1..0$ evaluates to pair $(1^{\mathsf{W}}, \mathsf{f}^{\mathsf{W}})$ that states that the first element is vector 1^{W} yet the range has terminated for all threads. We define the abbreviation $i + r \overset{\text{def}}{=} (i + n)..m$ when $r = n..m$. For instance, range $1 + (0..0)$ is an abbreviation for $(0 + 1)..0$.

3.2 Dynamic Semantics for Memory Access Protocols (Vector)

In this section, we introduce the semantics of a Memory Access Protocols for a single warp. Our semantics is parameterized by a metric M, that can be instantiated to measure bank conflicts and uncoalesced accesses.

Evaluating protocols. Figure 2 gives an operational big-step semantics, notation $\langle p, B, \sigma\rangle \Downarrow_M i$, to evaluate a Memory Access Protocol p, predicated (or masked) on a vector of enabled threads B, a vector environment σ, and a metric M that yields a cost i. Protocol skip yields a run-time cost of 0 abstract resources (*e.g.*, conflicts). Accessing memory with $\mathsf{A}[n]$ yields a cost of $M(I, B)$, which takes a vector of indices I (obtained from evaluating n) and a vector of enabled threads B. Sequencing protocols amounts to adding the costs. A conditional restricts the existing the set of enabled threads I with the result of the condition C (from evaluating c). The for-loop yields a cost of 0

Syntax

$$\mathcal{P} \ni \quad p, q \quad ::= \quad \mathsf{skip} \mid \mathsf{A}[n] \mid p\,;p \mid \mathsf{if}\ (c)\ \{p\} \mid \mathsf{for}\ x \in r\ \{p\}$$

Big-step semantics for protocols $\mathcal{P}$ $\boxed{\langle p, B, \sigma\rangle \Downarrow_M i}$

$$\textsc{Skip}\ \frac{}{\langle \mathsf{skip}, B, \sigma\rangle \Downarrow_M 0} \qquad \textsc{Acc}\ \frac{\langle n, \sigma\rangle \Downarrow I}{\langle \mathsf{A}[n], B, \sigma\rangle \Downarrow_M M(I, B)} \qquad \textsc{Seq}\ \frac{\langle p_1, B, \sigma\rangle \Downarrow_M i \qquad \langle p_2, B, \sigma\rangle \Downarrow_M j}{\langle p_1\,; p_2, B, \sigma\rangle \Downarrow_M i + j}$$

$$\textsc{If-T}\ \frac{\langle c, \sigma\rangle \Downarrow C \qquad C \neq \mathsf{f}^{\mathsf{W}} \qquad \langle p, B \wedge C, \sigma\rangle \Downarrow_M i}{\langle \mathsf{if}\ (c)\ \{p\}, B, \sigma\rangle \Downarrow_M i} \qquad \textsc{If-F}\ \frac{\langle c, \sigma\rangle \Downarrow \mathsf{f}^{\mathsf{W}}}{\langle \mathsf{if}\ (c)\ \{p\}, B, \sigma\rangle \Downarrow_M 0}$$

$$\textsc{For-1}\ \frac{\langle r, \sigma\rangle \Downarrow (I, \mathsf{f}^{\mathsf{W}})}{\langle \mathsf{for}\ x \in r\ \{p\}, B, \sigma\rangle \Downarrow_M 0} \qquad \textsc{For-2}\ \frac{\begin{array}{c}\langle r, \sigma\rangle \Downarrow (I, C) \qquad C \neq \mathsf{f}^{\mathsf{W}} \\ \langle p, B \wedge C, \sigma[x \mapsto I]\rangle \Downarrow_M j \qquad \langle \mathsf{for}\ x \in 1 + r\ \{p\}, B, \sigma\rangle \Downarrow_M k\end{array}}{\langle \mathsf{for}\ x \in r\ \{p\}, B, \sigma\rangle \Downarrow_M j + k}$$

Fig. 2. Syntax and semantics memory access protocols.

when range r is empty f^{W}. Otherwise ($C \neq \mathsf{f}^{\mathsf{W}}$), we add the cost of running one iteration i with the rest of the loop j. To obtain i, we run the loop body p and assign the value I of the lower bound of range r to loop variable x, only enabling the threads C that are still activated for this iteration. To obtain j, we continue the execution of the loop with one fewer iteration, by incrementing the lower bound $(1 + r)$.

Example 3.1 (Thread-divergent loop cost). Let $M_{\mathsf{E}}(I, B)$ return $\mathrm{count}(B, \mathsf{t})$, *i.e.*, returns the number of occurrences of t in B. We now show that the evaluation of the protocol in Example 2.1 yields $10 = 4 + 3 + 2 + 1$. We have

$$\frac{\langle 1..\mathsf{tid}, \emptyset\rangle \Downarrow (1^4, \mathsf{fttt}) \quad \langle \mathsf{A}[0], \mathsf{fttt}, \{\mathsf{x}\colon 1^4\}\rangle \Downarrow_{M_{\mathsf{E}}} 3 \quad \langle \mathsf{for}\ \mathsf{x} \in 2..\mathsf{tid}\ \{\mathsf{A}[0]\}, \mathsf{t}^4, \emptyset\rangle \Downarrow_{M_{\mathsf{E}}} (2 + 1)}{\langle \mathsf{for}\ \mathsf{x} \in 1..\mathsf{tid}\ \{\mathsf{A}[0]\}, \mathsf{t}^4, \emptyset\rangle \Downarrow_{M_{\mathsf{E}}} (3 + 2 + 1)} \tag{5}$$

and

$$\frac{\langle 0..\mathsf{tid}, \emptyset\rangle \Downarrow (0^4, \mathsf{t}^4) \qquad \langle \mathsf{A}[0], \mathsf{t}^4, \{\mathsf{x}\colon 0^4\}\rangle \Downarrow_{M_{\mathsf{E}}} 4 \qquad (5)}{\langle \mathsf{for}\ \mathsf{x} \in 0..\mathsf{tid}\ \{\mathsf{A}[0]\}, \mathsf{t}^4, \emptyset\rangle \Downarrow_{M_{\mathsf{E}}} (4 + 3 + 2 + 1)}$$

3.3 Dynamic Semantics for Resource Calculus (Scalar)

Our scalar abstract cost models are programs that operate on numeric and boolean expressions of scalars. We give the syntax and semantics of our sequential programs in Figure 3. A statement s is either $\mathsf{tick}(n)$ that uses n resources, a sequence, a conditional, or a loop. The judgment $\langle n, S\rangle \downarrow i$ states that numeric expression n evaluates to a number i and judgment $\langle c, S\rangle \downarrow b$ states that boolean expression c evaluates to a boolean b. Both judgments use an environment S that maps variables to numbers (scalars). Judgment $\langle s, S\rangle \downarrow i$ states that statement s evaluates to a cost i under environment S. The rule Tick states that statement $\mathsf{tick}(n)$ produces i resources, which result from evaluating the argument n and obtaining a value of i. Sequencing adds results from the cost of statements s_1 and s_2. Conditionals take the cost of the guarded statement s when the test c evaluates to true, otherwise the statement produces 0 resources. A for-loop becomes a summation of the cost of the loop body s.

Syntax

$$\begin{array}{llll} \mathcal{S} \ni & s, t, u & ::= & \mathsf{tick}(n) \mid s\,;s \mid \mathsf{if}\ (c)\ \{s\} \mid \mathsf{for}\ x \in r\ \{s\} \\ S \ni & \mathcal{V} \to \mathbb{N} & & \\ & E & ::= & (B, \sigma, S). \end{array}$$

Big-step semantics for numeric expressions n $\boxed{\langle n, S\rangle \downarrow i}$

$$\langle i, S\rangle \downarrow i \qquad \langle x, S\rangle \downarrow S(x) \qquad \frac{\langle n, S\rangle \downarrow i \quad \langle m, S\rangle \downarrow j}{\langle n \star m, S\rangle \downarrow i \star j}$$

Big-step semantics for boolean expressions c and ranges $n..m$ $\boxed{\langle c, S\rangle \downarrow b}$ $\boxed{\langle n..n, S\rangle \downarrow (i, b)}$

$$\langle b, S\rangle \downarrow b \qquad \frac{\langle n, S\rangle \downarrow i \quad \langle m, S\rangle \downarrow j}{\langle n \diamond m, S\rangle \downarrow i \diamond j} \qquad \frac{\langle c, S\rangle \downarrow a \quad \langle d, S\rangle \downarrow b}{\langle c \circ d, S\rangle \downarrow a \circ b} \qquad \frac{\langle n, S\rangle \downarrow i \quad \langle n \leq m, S\rangle \downarrow b}{\langle n..m, S\rangle \downarrow (i, b)}$$

Big-step semantics for statements s $\boxed{\langle s, S\rangle \downarrow i}$

$$\textsc{tick}\ \frac{\langle n, S\rangle \downarrow i}{\langle \mathsf{tick}(n), S\rangle \downarrow i} \qquad \textsc{seq}\ \frac{\langle s_1, S\rangle \downarrow i_1 \quad \langle s_2, S\rangle \downarrow i_2}{\langle s_1; s_2, S\rangle \downarrow i_1 + i_2} \qquad \textsc{if-t}\ \frac{\langle c, S\rangle \downarrow \mathsf{t} \quad \langle s, S\rangle \downarrow i}{\langle \mathsf{if}\ (c)\ \{s\}, S\rangle \downarrow i} \qquad \textsc{if-f}\ \frac{\langle c, S\rangle \downarrow \mathsf{f}}{\langle \mathsf{if}\ (c)\ \{s\}, S\rangle \downarrow 0}$$

$$\textsc{for-skip}\ \frac{\langle r, S\rangle \downarrow (i, \mathsf{f})}{\langle \mathsf{for}\ x \in r\ \{s\}, S\rangle \downarrow 0} \qquad \textsc{for-iter}\ \frac{\langle n..m, S\rangle \downarrow (i, \mathsf{t}) \quad \langle s, S[x \mapsto i]\rangle \downarrow j \quad \langle \mathsf{for}\ x \in 1 + r\ \{s\}, S\rangle \downarrow k}{\langle \mathsf{for}\ x \in r\ \{s\}, S\rangle \downarrow j + k}$$

Dynamic relational cost judgment $\boxed{E \models p \sim s}$

$$\frac{\langle p, B, \sigma\rangle \Downarrow_M i \quad \langle s, S\rangle \downarrow j \quad i \sim j}{(B, \sigma, S) \models p \sim s}$$

Fig. 3. Syntax and semantics of Resource calculus.

Example 3.2 (Sequential loop cost). We show that the statement in Example 2.1 yields a cost of 10 when W = 4, which equals the cost of the thread-divergent (vector) counterpart (Example 3.1).

$$\frac{\langle 0..(4-1), \emptyset\rangle \Downarrow (0, \mathsf{t}) \quad \langle \mathsf{tick}(4 - \mathsf{x}), \{\mathsf{x}\colon 0\}\rangle \downarrow 4 \quad \langle \mathsf{for}\ \mathsf{x} \in 1..(4-1)\ \{\mathsf{tick}(4 - \mathsf{x})\}, \emptyset\rangle \downarrow 6}{\langle \mathsf{for}\ \mathsf{x} \in 0..(4-1)\ \{\mathsf{tick}(4 - \mathsf{x})\}, \emptyset\rangle \downarrow (4 + 6)}$$

3.4 Dynamic Relational-Cost Analysis

The dynamic relational-cost provides the true cost measurement—in the logical sense—that serves as the basis for establishing relational bounds of protocols with sequential programs. In this section, we introduce a *dynamic* relational cost judgment that compares resource usage: protocols may use exactly (=), at most (≤), or at least (≥) the same resources as sequential programs. We first define the cost relation that parameterizes our analysis, then introduce our dynamic relational-cost judgment $\cdot \models p \sim s$. Our judgment is parameterized by a cost relation, $\sim$, which we define following.

Definition 3.3 (Cost relation). We define a *cost relation* as a standard additive relation, ranged over by $\sim$, as a binary relation on $\mathbb{N} \times \mathbb{N}$ where $0 \sim 0$ and relation $\sim$ is closed under addition, *i.e.*, for any i_1, i_2, i_3, i_4 we have that $i_1 \sim i_3$ and $i_2 \sim i_4$ implies $i_1 + i_2 \sim i_3 + i_4$.

We have that $=$, $\le$, and $\ge$ are all cost relations.

Let E denote a dynamic (relational) environment, defined as $E ::= (B, \sigma, S)$ where B denotes a vector of enabled threads, σ denotes a vector environment, and S denotes a scalar environment. The vector of enabled threads B and vector environment σ are both required to evaluate a protocol, while scalar environment S is required to evaluate a statement. This combined environment enables our relational judgment to evaluate both the protocol and the statement within the same context, allowing direct cost comparison. For instance, the relational environment $(\mathrm{t}^4, \emptyset, \emptyset)$ contains empty stores and enabled threads t^4 for Example 2.1.

Judgment $E \models p \sim s$ (Figure 3, bottom) states that the execution cost of protocol p is related to the execution cost of program s by relation $\sim$ under environment E. We now apply this judgment to demonstrate how the dynamic relational cost analysis validates the static cost equality from Example 2.1. The vector evaluation takes the enabled threads t^4 as a parameter and yields cost 10, while the scalar evaluation takes the scalar store and also yields cost 10, establishing the equality relation. The following inference rule formalizes this relational judgment:

$$\frac{\langle \mathsf{for}\ \mathsf{x} \in 0..\mathsf{tid}\ \{\mathsf{A}[0]\}, \mathrm{t}^4, \emptyset\rangle \Downarrow_{M_{\mathrm{E}}} 10 \qquad \langle \mathsf{for}\ \mathsf{x} \in 0..(4-1)\ \{\mathsf{tick}(4-\mathsf{x})\}, \emptyset\rangle \downarrow 10}{(\mathrm{t}^4, \emptyset, \emptyset) \models (\mathsf{for}\ \mathsf{x} \in 0..\mathsf{tid}\ \{\mathsf{A}[0]\}) = (\mathsf{for}\ \mathsf{x} \in 0..(4-1)\ \{\mathsf{tick}(4-\mathsf{x})\})}$$

4 Static Relational-Cost Analysis

We present the static type system that derives the cost relationships shown in the running examples (Examples 2.1 and 2.2). Our type system uses relational types to reason about both vector and scalar interpretations of expressions, enabling static cost analysis of thread-divergent GPU code. We first present the type syntax and explain relational types (Section 4.1), then demonstrate the analysis on the running examples (Section 4.2), and state the main soundness result (Section 4.3).

4.1 Type System

We define our relational type system in three parts: type syntax and environments, typing judgments for expressions, and the main cost relation judgment.

Types and typing environments. The syntax of types τ is given in Figure 4. In our relational system, the same variable has two meanings: a vector meaning (for GPU protocols) and a scalar meaning (for sequential programs). Intuitively, a relational type can be thought of as a set of pairs, each pair containing a vector and a scalar. The *uniform* type U describes a uniform value paired with its scalar equivalent, say 4^{W} and 4. The *divergent* type D describes a vector (whose elements vary) paired with a scalar, say $(0, 1, 2, 3)$ and 0. We now introduce our typing environments. A typing environment Γ refines types with ranges to enforce correspondence between vector and scalar interpretations; values assigned to each variable must be picked from the same i-th iteration of both ranges. Typing $\emptyset$ denotes the empty context. Constructor $\Gamma, x \colon \tau_{(r,r')}$ assigns type τ to variable x and expresses that variable x admits a vectorized value picked from range r and a scalar value picked from range r'. We may just omit range annotations when that information is unnecessary and write $\Gamma, x \colon \tau$. To lookup a variable's type, $\Gamma(x)$ returns τ if there is an element $x \colon \tau_{\_}$ in Γ. The domain of Γ is written as $\mathrm{dom}(\Gamma)$.

Typing expressions. We define typing judgments that assign uniform and divergent types to numeric expressions, boolean expressions, and loop ranges. Capturing uniform expressions statically provides the guarantee that the same expression has identical interpretations in vector and scalar contexts. Judgment $\Gamma \vdash n \colon \tau$ states that, under typing context Γ, a numeric expression n has type τ. Literals i are uniform; the thread identifier tid is thread divergent, as each thread holds a unique identifier number; variables are looked up in the typing context Γ. Binary operations combine

Syntax

$$\tau ::= \mathsf{U} \mid \mathsf{D} \qquad \Gamma ::= \emptyset \mid \Gamma, x\colon \tau_{(r,r)} \qquad \Phi ::= \mathsf{t} \mid \Phi \wedge c \qquad \mathcal{I} ::= \approx \mid \leq \mid \geq$$

Typing expressions $\boxed{\Gamma \vdash n\colon \tau}$ $\boxed{\Gamma \vdash c\colon \tau}$ $\boxed{\Gamma \vdash r\colon \tau}$

$$\Gamma \vdash i\colon \mathsf{U} \qquad \Gamma \vdash \mathsf{tid}\colon \mathsf{D} \qquad \Gamma \vdash x\colon \Gamma(x) \qquad \frac{\Gamma \vdash n\colon \tau \qquad \Gamma \vdash m\colon \tau'}{\Gamma \vdash n \star m\colon \tau + \tau'} \qquad \Gamma \vdash b\colon \mathsf{U}$$

$$\frac{\Gamma \vdash n\colon \tau \qquad \Gamma \vdash m\colon \tau'}{\Gamma \vdash n \diamond m\colon \tau + \tau'} \qquad \frac{\Gamma \vdash c\colon \tau \qquad \Gamma \vdash d\colon \tau'}{\Gamma \vdash c \circ d\colon \tau + \tau'} \qquad \frac{\Gamma \vdash n\colon \tau \qquad \Gamma \vdash m\colon \tau'}{\Gamma \vdash n..m\colon \tau + \tau'}$$

Relational cost judgment $\boxed{\Gamma; \Phi \vdash p \sim s}$

$$\textsc{skip}\ \frac{}{\Gamma; \Phi \vdash \mathsf{skip} \sim \mathsf{tick}(0)} \qquad \textsc{acc}\ \frac{\Gamma; \Phi \vdash M(n) \sim m}{\Gamma; \Phi \vdash \mathsf{A}[n] \sim \mathsf{tick}(m)} \qquad \textsc{seq}\ \frac{\Gamma; \Phi \vdash p_1 \sim s_1 \qquad \Gamma; \Phi \vdash p_2 \sim s_2}{\Gamma; \Phi \vdash p_1 ; p_2 \sim s_1 ; s_2}$$

$$\textsc{if-u}\ \frac{\Gamma \vdash c\colon \mathsf{U} \qquad \Gamma; \Phi \vdash p \sim s}{\Gamma; \Phi \vdash \mathsf{if}\ (c)\ \{p\} \sim \mathsf{if}\ (c)\ \{s\}} \qquad \textsc{if-d}\ \frac{\Gamma \vdash c\colon \mathsf{D} \qquad \Gamma; \Phi \wedge c \vdash p \sim s}{\Gamma; \Phi \vdash \mathsf{if}\ (c)\ \{p\} \sim s}$$

$$\textsc{for}\ \frac{\Gamma; \Phi \vdash_{\sim} r\ \mathcal{I}\ r'\colon \tau \qquad \Gamma, x\colon \tau_{(r,r')}; \Phi \vdash p \sim s}{\Gamma; \Phi \vdash \mathsf{for}\ x \in r\ \{p\} \sim \mathsf{for}\ x \in r'\ \{s\}}$$

Fig. 4. Relational-cost analysis of vectorized protocols vs scalar programs.

uniformity with $\tau_1 + \tau_2$ defined as follows $\mathsf{U} + \tau = \tau + \mathsf{U} = \tau + \tau = \tau$, which states that the result is uniform only when the type of both operands is uniform, otherwise the result is divergent D. Judgment $\Gamma \vdash c\colon \tau$ states boolean expression c has a type τ under typing context Γ. The rules are similar to those of numeric expressions. Finally, judgment $\Gamma \vdash n..m\colon \tau$ assigns a type to a loop range. When both bounds of a range are uniform, the range is uniform.

Typing protocols with programs. We now describe our judgment to relate protocols with programs. Judgment $\Gamma; \Phi \vdash p \sim s$ relates the execution costs of protocol p and program s under cost relation $\sim$ and metric M. Judgment $\Gamma; \Phi \vdash p = s$ states that protocol p has the same execution cost as that of program s, while $\Gamma; \Phi \vdash p \leq s$ states that the execution cost of p is lower than the execution cost of program s. The judgment uses typing environment Γ for variables and constraint environment Φ to control which threads are active. For instance, constraint $\mathsf{t} \wedge x = 3$ only activates threads for which the condition $x = 3$ evaluates to true.

We now explain the rules of judgment $\Gamma; \Phi \vdash p \sim s$. Rule Skip states that protocol skip has an execution cost of 0 ticks. Rule Acc states that the cost of a memory access is given by the judgment $\Gamma; \Phi \vdash M(n) \sim m$, called the *metric analysis*. We postpone the full definition to Section 5.3, but illustrate with an example from Example 2.1: $\emptyset, \mathsf{x}\colon \mathsf{D}_{(0..\mathsf{tid}, 0..\mathsf{W}-1)}; \mathsf{t} \vdash \mathsf{M}_{\mathsf{E}}(0) = \mathsf{W} - \mathsf{x}$ establishes that the number of active threads accessing index 0 is given by the scalar expression $\mathsf{W} - \mathsf{x}$. In this example, the type $\mathsf{D}_{(0..\mathsf{tid}, 0..\mathsf{W}-1)}$ being assigned to x gives us the required information to conclude that the scalar expression $\mathsf{W} - \mathsf{x}$ is correct (see Lemma 5.8). Since variable x is picked from the *same* iteration from both ranges, when we are at iteration i, we have $\mathsf{x} = i^{\mathsf{W}}$ (vector) and $\mathsf{x} = i$ (scalar). This means there are $\mathsf{W} - i$ active threads.

Rule SEQ is straightforward. We have two rules for conditionals that depend on the type of the test c. Rule IF-U states that when the test c is uniform U and the conditional's body is related $\Gamma; \Phi \vdash p \sim s$, then the conditional preserves the cost. When c is thread divergent (Rule IF-D), then the scalar program cannot express the conditional directly, so we constrain the active threads, by extending the typing environment Φ with the test c. Indeed, by extending the constraint, we delegate capturing the divergent condition c to the metric analysis.

Rule FOR states that when the two iteration spaces are related, $\Gamma; \Phi \vdash r \, \mathcal{I} \, r' : \tau$, and we can relate the cost of the loop bodies p with s under a typing context extended with $x : \tau$, then the two loops are related. Iteration type $\approx$ states that both ranges must have the same number of iterations, iteration type $\preceq$ represents that the left-hand side range must terminate before or at the same time of the right-hand side, and iteration type $\succeq$ represents the converse. Judgment $\Gamma; \Phi \vdash_{\sim} r \, \mathcal{I} \, r' : \tau$, called the *loop analysis*, lets us relate two iteration spaces with an iteration type $\mathcal{I}$ for a cost relation $\sim$. We postpone the full definition to Section 5.2, but illustrate with an example. For instance, from Example 2.1, the judgment $\emptyset; \mathsf{t} \vdash_{=} 0..\mathsf{tid} \approx 0..\mathsf{W}-1 : \mathsf{D}$ states that ranges have the same ($\approx$) number of iterations (here there are W iterations). The loop analysis yields a divergent type D because threads become inactive at different iterations, *i.e.*, we have thread divergence.

4.2 Running Examples

We demonstrate our type system by working through the derivations from the running examples (Examples 2.1 and 2.2). Each derivation demonstrates the expressiveness of our type system: reasoning about thread-divergent loops with precise cost relationships and thread-divergent conditionals with bounded costs. In both cases we postpone the proofs of loop and metric analyses to Sections 5.2 and 5.3, respectively.

Example 4.1. The following derivation proves Equation (1), from Example 2.1.

$$\dfrac{(6)\ \emptyset; \mathsf{t} \vdash_{=} 0..\mathsf{tid} \approx 0..\mathsf{W}-1 : \mathsf{D} \qquad \dfrac{(7)\ \emptyset, \mathsf{x} : \mathsf{D}_{(0..\mathsf{tid}, 0..\mathsf{W}-1)}; \mathsf{t} \vdash M_{\mathsf{E}}(0) = \mathsf{W}-\mathsf{x}}{\emptyset, \mathsf{x} : \mathsf{D}_{(0..\mathsf{tid}, 0..\mathsf{W}-1)}; \mathsf{t} \vdash \mathsf{A}[0] = \mathsf{tick}(\mathsf{W}-\mathsf{x})}\ \text{ACC}}{\emptyset; \mathsf{t} \vdash (\mathsf{for}\ \mathsf{x} \in 0..\mathsf{tid}\ \{\mathsf{A}[0]\}) = (\mathsf{for}\ \mathsf{x} \in 0..\mathsf{W}-1\ \{\mathsf{tick}(\mathsf{W}-\mathsf{x})\})}\ \text{FOR}$$

The loop analysis of Equation (6) states that the vector-range $0..\mathsf{tid}$ has the same number of iterations ($\approx$) as the scalar-range $0..\mathsf{W}-1$, and that the vector-range is thread divergent (D), under an empty environment $\emptyset$ and constraint t.

$$\emptyset; \mathsf{t} \vdash_{=} 0..\mathsf{tid} \approx 0..\mathsf{W}-1 : \mathsf{D} \tag{6}$$

The metric analysis of Equation (7) establishes that the number of active threads (metric M_{E}) equals (=) the scalar expression $\mathsf{W}-\mathsf{x}$. This example demonstrates our type system's expressiveness in capturing exact static bounds for thread-divergent metrics. Observe that even though the loop range $0..\mathsf{tid}$ is typed as divergent (D), which is a sound approximation of the execution behavior, the typing does not prevent the metric analysis from deriving an exact cost relation (=). Divergent types only add more proof effort, as uniform ranges are simpler to reason about. The typing environment declares a *relational* range of values that can be assigned to x, that is, in the vectorized context x is picked from $0..\mathsf{tid}$ and in the scalar context x is picked from $0..(\mathsf{W}-1)$.

$$\emptyset, \mathsf{x} : \mathsf{D}_{(0..\mathsf{tid}, 0..\mathsf{W}-1)}; \mathsf{t} \vdash \mathsf{M}_{\mathsf{E}}(0) = \mathsf{W}-\mathsf{x} \tag{7}$$

Example 4.2. The following derivation shows the relational cost analysis stated in Equation (2), from Example 2.2. Observe how the thread-divergent conditional pushes a constraint $\mathsf{tid} \% 2 = 0$

which then affects the metric analysis.

$$\dfrac{(8)\ \emptyset; \mathsf{t} \vdash_{\leq} 0..i \approx 0..i : \mathsf{U} \qquad \dfrac{\dfrac{(9)\ \emptyset, \mathsf{x} \colon \mathsf{U}_{0..i,0..i}; \mathsf{t} \land \mathsf{tid} \mathbin{\%} 2 = 0 \vdash \mathsf{ua}(\mathsf{tid} \times i + \mathsf{x}) \leq \lfloor \mathsf{W}/2 \rfloor}{\emptyset, \mathsf{x} \colon \mathsf{U}_{0..i,0..i}; \mathsf{t} \land \mathsf{tid} \mathbin{\%} 2 = 0 \vdash \mathsf{A}[\mathsf{tid} \times i + \mathsf{x}] \leq \mathsf{tick}(\lfloor \mathsf{W}/2 \rfloor)}\ \textsc{acc} \qquad \emptyset \vdash \mathsf{tid} \mathbin{\%} 2 = 0 \colon \mathsf{D}}{\emptyset, \mathsf{x} \colon \mathsf{U}_{0..i,0..i}; \mathsf{t} \vdash \mathbf{if}\ (\mathsf{tid} \mathbin{\%} 2 = 0)\ \{\mathsf{A}[\mathsf{tid} \times i + \mathsf{x}]\} \leq \mathsf{tick}(\lfloor \mathsf{W}/2 \rfloor)}\ \textsc{if-d}}{\emptyset; \mathsf{t} \vdash \mathbf{for}\ \mathsf{x} \in 0..i\ \{\mathbf{if}\ (\mathsf{tid} \mathbin{\%} 2 = 0)\ \{\mathsf{A}[\mathsf{tid} \times i + \mathsf{x}]\}\} \leq \mathbf{for}\ \mathsf{x} \in 0..i\ \{\mathsf{tick}(\lfloor \mathsf{W}/2 \rfloor)\}}\ \textsc{for}$$

The loop analysis of Equation (8) states that the vector-range $0..i$ has the same number of iterations ($\approx$) as the scalar-range $0..i$, and that the vector-range is thread uniform (U), under an empty environment $\emptyset$ and constraint t.

$$\emptyset; \mathsf{t} \vdash_{\leq} 0..i \approx 0..i \colon \mathsf{U} \tag{8}$$

The metric analysis of Equation (9) establishes that the number of uncoalesced accesses (metric ua) is bounded from above ($\leq$) by the scalar expression $\lfloor \mathsf{W}/2 \rfloor$.

$$\emptyset, \mathsf{x} \colon \mathsf{U}_{0..i,0..i}; \mathsf{t} \land \mathsf{tid} \mathbin{\%} 2 = 0 \vdash \mathsf{ua}(\mathsf{tid} \times i + \mathsf{x}) \leq \lfloor \mathsf{W}/2 \rfloor \tag{9}$$

4.3 Closed Soundness

We present the main soundness result of this paper. Soundness establishes that static cost relations, $\emptyset; \mathsf{t} \vdash p \sim s$, derived by our type system correspond to actual dynamic cost relations, $(\mathsf{t}^{\mathsf{W}}, \emptyset, \emptyset) \models p \sim s$, observed during execution. We first present a simplified version for closed terms (programs without free variables), see Corollary 4.3 below. The empty environments ensure that protocols and statements are closed, while the t constraint means all threads are active (denoted by t^{W} dynamically). The full proof with open terms is developed in Section 5.4.

Corollary 4.3 (Soundness for closed terms). *Let M be a metric and $\sim$ a cost relation. If $\emptyset; \mathsf{t} \vdash p \sim s$ then $(\mathsf{t}^{\mathsf{W}}, \emptyset, \emptyset) \models p \sim s$.*

5 Meta-Theory and Soundness

This section describes the meta-theory to establish Corollary 4.3 as well as its generalized counterpart. Readers interested primarily in the cost analysis approach and practical applications may skip the technical development that follows. We prove that our static type system accurately captures dynamic GPU execution costs. Recall that all definitions and proofs in this article have been formalized and machine-checked in Rocq. The goal of this section is twofold: first, we establish the soundness of our type system; second, we demonstrate the expressiveness and practical applications illustrated by our running examples (Examples 2.1 and 2.2) through formal proofs of their metric and loop analyses. Our approach to the metric and loop analysis judgments departs from the usual way of defining type systems: to maintain generality and support arbitrary metrics and cost relations, we define these judgments in terms of dynamic relational-cost rather than through syntactic rules. We accomplish these goals through compatible environments (Section 5.1), followed by three main developments. First, we define loop analysis, establish a key soundness lemma for loops, and prove the loop analyses (Section 5.2). Second, we formally define metric analysis and prove the soundness of the metric analyses (Section 5.3). Third, we present our main contribution: our general soundness theorem that ensures static cost derivations correspond to dynamic execution costs (Section 5.4).

5.1 Compatible Environments

Our goal with soundness is to state that a static relational cost ($\Gamma; \Phi \vdash p \sim s$) implies a dynamic relational cost ($E \models p \sim s$). To be able to state such a result, we need an invariant, which relates a typing environment $\Gamma; \Phi$ with a dynamic environment E.

Nonempty ranges $\boxed{E \models r \Downarrow b}\ \boxed{E \models r \downarrow b}$

$$\frac{\langle n \le m, \sigma\rangle \Downarrow C \qquad C \neq \mathsf{f}^{\mathsf{W}}}{(B,\sigma,S) \models n..m \Downarrow \mathsf{t}} \qquad \frac{\langle n \le m, \sigma\rangle \Downarrow \mathsf{f}^{\mathsf{W}}}{(B,\sigma,S) \models n..m \Downarrow \mathsf{f}} \qquad \frac{\langle n \le m, S\rangle \downarrow b}{(B,\sigma,S) \models n..m \downarrow b}$$

Uniform expressions $\boxed{E \models n\,\mathsf{unif}}\ \boxed{E \models c\,\mathsf{unif}}\ \boxed{E \models r\,\mathsf{unif}}$

$$\frac{x \in \mathrm{fv}(n) \implies S(x)^{\mathsf{W}} = \sigma(x)}{(B,\sigma,S) \models n\,\mathsf{unif}} \qquad \frac{x \in \mathrm{fv}(c) \implies S(x)^{\mathsf{W}} = \sigma(x)}{(B,\sigma,S) \models c\,\mathsf{unif}} \qquad \frac{E \models n\,\mathsf{unif} \qquad E \models m\,\mathsf{unif}}{E \models n..m\,\mathsf{unif}}$$

Environment compatibility $\boxed{\Gamma;\Phi \bowtie E}$

$$\textsc{empty}\ \frac{}{\emptyset; \mathsf{t} \bowtie (\mathsf{t}^{\mathsf{W}}, \emptyset, \emptyset)} \qquad \textsc{cnd}\ \frac{\Gamma;\Phi \bowtie (B,\sigma,S) \qquad \langle c,\sigma\rangle \Downarrow C}{\Gamma;\Phi \wedge c \bowtie (B \wedge C, \sigma, S)}$$

$$\textsc{var-1}\ \frac{\begin{array}{c}\Gamma;\Phi \bowtie (B,\sigma,S) \qquad x \notin \mathrm{dom}\,\Gamma \qquad (\tau = \mathsf{U} \implies (B,\sigma,S) \models r\,\mathsf{unif} \wedge r' = r) \\ \langle j + r, \sigma\rangle \Downarrow (I, C) \qquad C \neq \mathsf{f}^{\mathsf{W}} \qquad \langle j + r', S\rangle \downarrow (i, \mathsf{t})\end{array}}{\Gamma, x \colon \tau_{(r,r')}; \Phi \bowtie (B \wedge C, \sigma[x \mapsto I], S[x \mapsto i])}$$

$$\textsc{var-2}\ \frac{\Gamma;\Phi \bowtie (B,\sigma,S) \qquad x \notin \mathrm{dom}\,\Gamma \qquad \langle r, \sigma\rangle \Downarrow (I, C) \qquad \langle r', S\rangle \downarrow (i, b) \qquad (b = \mathsf{f} \vee C = \mathsf{f}^{\mathsf{W}})}{\Gamma, x \colon \tau_{(r,r')}; \Phi \bowtie (B \wedge C, \sigma[x \mapsto I], S[x \mapsto i])}$$

Iteration space judgment $\boxed{E \models r\ \mathcal{I}\ r'}$

$$\textsc{step}\ \frac{E \models (1 + r)\ \mathcal{I}\ (1 + r') \qquad E \models r \Downarrow \mathsf{t} \qquad E \models r' \downarrow \mathsf{t}}{E \models r\ \mathcal{I}\ r'}$$

$$\textsc{eq}\ \frac{E \models r \Downarrow \mathsf{f} \qquad E \models r' \downarrow \mathsf{f}}{E \models r \approx r'} \qquad \textsc{le}\ \frac{E \models r \Downarrow b \qquad E \models r' \downarrow \mathsf{f}}{E \models r \le r'} \qquad \textsc{ge}\ \frac{E \models r \Downarrow \mathsf{f} \qquad E \models r' \downarrow b}{E \models r \ge r'}$$

Cost relation and iteration type compatibility $\boxed{\sim \colon \mathcal{I}}$

$$\frac{}{\sim \colon \approx} \qquad \frac{\forall i \colon 0 \sim i}{\sim \colon \le} \qquad \frac{\forall i \colon i \sim 0}{\sim \colon \ge}$$

Fig. 5. Soundness judgments.

Judgment $\Gamma;\Phi \bowtie E$ (Figure 5) states that typing environment $\Gamma;\Phi$ is compatible with a dynamic environment E. Rule EMPTY states that the empty typing context is compatible with the empty runtime environment. The constraint t states that all threads are initially active, which is captured dynamically by t^{W}. The remaining rules extend compatible environments $\Gamma;\Phi \bowtie (B,\sigma,S)$ with either a new constraint (Rule CND), or a new variable (Rules VAR-1 and VAR-2). Rule CND adds a static constraint c and dynamic constraint C when $\langle c,\sigma\rangle \Downarrow C$ to a compatible environment $\Gamma;\Phi \bowtie (B,\sigma,S)$, which mirrors the Rule IF-T of protocol evaluation (Figure 2). The expression c (dynamically C) represents a test of a conditional that restricts the set of active threads Φ (dynamically B).

Rules VAR-1 and VAR-2 allow us to extend compatible environments with a binding for variable x. This amounts to adding type τ for x in the static environment, and adding corresponding values (scalar i in the scalar environment S and vector I in the vector environment σ) in the dynamic environment. The added values are picked from their respective ranges. Rule VAR-1 is used when both ranges r and r' are nonempty. Rule VAR-2 is used when at least one range is empty.

To explain these rules in detail, we first need to define (dynamic) uniformity. We introduce uniformity for expressions in Figure 5. Judgment $E \models n\,\mathsf{unif}$ states that expression n is uniform under environment E. We say that an expression n is uniform when every free variable x therein has the "same" value in vector σ and scalar S environments, that is, $S(x)^{\mathsf{W}} = \sigma(x)$. Function $\mathrm{fv}(\cdot)$ is the standard free variables function for protocols p, statements s, and expressions (r, c, and n). Judgment $E \models c\,\mathsf{unif}$ states that expression c is uniform under environment E, and judgment $E \models r\,\mathsf{unif}$ states the same for range expressions.

Rule VAR-1 states that the value being assigned to a loop variable x is picked in lockstep from the same j-th iteration picked from the vector r and scalar r' ranges. The rule also requires range r to be uniform ($E \models r\,\mathsf{unif}$) when the assigned type is uniform $\tau = \mathsf{U}$. Rule VAR-2 states that if either of the ranges have terminated, $b = \mathsf{f} \vee C = \mathsf{f}^{\mathsf{W}}$, then the first value of each range (vector I and scalar i) are assigned to variable x in their respective environment (vector $\sigma[x \mapsto I]$ and scalar $S[x \mapsto i]$).

Having established the rules for compatible environments, we can now examine their implications. We observe that compatible environments yield strong guarantees about the possible values of each variable and how such values relate. For instance, given type information $\Gamma(x) = \tau_{(r,r')}$ if we know that both ranges are nonempty (*i.e.*, $\langle r, S\rangle \downarrow \mathsf{t}$ and $\langle r', \sigma\rangle \Downarrow \mathsf{t}$), then there must exist an iteration i such that: the vector-value of x (*i.e.*, $\sigma(x) = I$) results from iteration i (*i.e.*, $\langle i + r, \sigma\rangle \Downarrow (I, B')$), and range r is nonempty at that point ($B \neq \mathsf{f}^{\mathsf{W}}$); the scalar-value of x (*i.e.*, $S(x) = j$) results from iteration i (*i.e.*, $\langle i + r', S\rangle \downarrow (j, \mathsf{t})$), and range r' is nonempty at that point.

We formalize this relation in Lemma 5.1 below.

LEMMA 5.1. *Let* $\Gamma; \Phi \bowtie (B, \sigma, S)$. *If* $\Gamma(x) = \tau_{(r,r')}$, $\langle r, S\rangle \downarrow \mathsf{t}$, $\langle r', \sigma\rangle \Downarrow \mathsf{t}$, *then there exists* i *such that* $\langle i + r, \sigma\rangle \Downarrow (I, B')$ *and* $B' \neq \mathsf{f}^{\mathsf{W}}$ *and* $\sigma(x) = I$; *and,* $\langle i + r', S\rangle \downarrow (j, \mathsf{t})$ *and* $S(x) = j$.

PROOF. The proof follows by induction on the structure of $\Gamma; \Phi \bowtie E$ and the various cases proceed with usual weakening and strengthening lemmas, all are included in the Rocq mechanization. □

We introduce an auxiliary result that obtains an expression encoding the vector of active threads.

LEMMA 5.2 (ACTIVE THREAD CONDITION). *Let function* $act(\Gamma)$ *be defined inductively as* $act(\emptyset) = \mathsf{t}$ *and* $act(\Gamma, x\colon \tau_{(n..m,r')}) = act(\Gamma) \wedge n \leq x \leq m$. *If* $\Gamma; \Phi \bowtie (B, \sigma, S)$, *then* $\langle act(\Gamma) \wedge \Phi, S\rangle \downarrow B$.

PROOF. The proof follows by induction on the structure of $\Gamma; \Phi \bowtie (B, \sigma, S)$. □

5.2 Loop Analysis

In this section, we present three main developments. First, we define the loop analysis judgment directly in terms of dynamic relational behavior between iteration spaces. Second, we establish our main result: a key soundness lemma for loops. Third, we prove the loop analyses for our running examples (Examples 4.1 and 4.2).

Relational evaluation of ranges. Judgment $E \models r \mathcal{I} r'$ (defined in Figure 5) states that the number of iterations of vector-r are related with the number of iterations of scalar r' according to iteration type $\mathcal{I}$. The judgment "reduces" both ranges one step at a time (Rule STEP), by incrementing the lower bound, until one of the ranges terminates, depending on the loop type's constraints. When the iteration type is $\approx$ then both ranges must terminate together (Rule EQ). When the iteration type is $\preceq$ then if the scalar-range terminates, then vector-range must terminate (Rule LE). When the iteration type is $\succeq$ then if the vector-range terminates, then scalar-range must terminate (Rule GE).

Loop analysis. We can now define the loop analysis judgment formally. The judgment holds when five conditions are satisfied: (1) the cost relation $\sim$ and iteration type $\mathcal{I}$ are compatible ($\sim\colon \mathcal{I}$), (2) the vector-range r is well typed, (3) the scalar range r' is well-formed, (4) for uniform ranges, both ranges must be syntactically equal, and (5) the ranges are dynamically related under all compatible environments. Well-formed terms α (*e.g.*, a statement s) are defined as $X \vdash \alpha \overset{\text{def}}{=} X \subseteq \text{fv}(\alpha)$. Judgment $\Gamma; \Phi \vdash_{\sim} r\ \mathcal{I}\ r' : \tau$ requires that for any compatible environment E (where $\Gamma; \Phi \bowtie E$), the dynamic relation $E \models r\ \mathcal{I}\ r'$ holds between the vector and scalar iteration spaces.

Definition 5.3 (Loop analysis). Judgment $\Gamma; \Phi \vdash_{\sim} r\ \mathcal{I}\ r' : \tau$ is defined as follows.

$$\frac{\sim\colon \mathcal{I} \qquad \Gamma \vdash r : \tau \qquad \text{dom}\,\Gamma \vdash r' \qquad \tau = \mathsf{U} \implies r = r' \qquad \forall E\colon \Gamma; \Phi \bowtie E \implies E \models r\ \mathcal{I}\ r'}{\Gamma; \Phi \vdash_{\sim} r\ \mathcal{I}\ r' : \tau}$$

Soundness for loops. Our soundness result for loops establishes loop-level soundness by proving the soundness of each individual iteration, *i.e.*, by comparing iteration i of loop r with iteration i of range r'. To achieve an expressive system that can relate loops of different lengths, we introduce iteration types $\mathcal{I}$ and a notion of compatibility that restricts which cost relations can be combined in loops of different iteration types.

Cost relation compatibility. Judgment $\sim\colon \mathcal{I}$, introduced in Figure 5, governs the requirements on a cost relation $\sim$ to compare loops of an iteration type $\mathcal{I}$. To understand the rules, consider comparing loops $\text{for}\ x \in r\ \{p\} \sim \text{for}\ x \in r'\ \{s\}$ where we have an iteration type $\Gamma; \Phi \vdash_{\sim} r\ \mathcal{I}\ r' : \tau$. Equal type $\approx$ imposes no restrictions since it guarantees the same number of iterations for both ranges, and any $\sim$ is closed under addition. At-most ($\preceq$) requires $0 \sim i$ for any i since the first loop may terminate (a cost of 0) before the second (a cost of i). At-least ($\succeq$) requires $i \sim 0$ for any i since the second loop may terminate (a cost of 0) before the first (a cost of i). Relations $=, \leq, \geq$ are compatible with $\approx$; $\leq$ with $\preceq$; and $\geq$ with $\succeq$.

Nonempty ranges. We introduce two judgments (in Figure 5) to state whether a range has elements, one for vector expressions and one for scalar expressions. Judgment $E \models r \Downarrow b$ (resp. $E \models r \downarrow b$) states whether range r has at least one element ($b = \text{t}$) using the vector (resp. scalar) semantics under environment E.

The following lemma states that as long as relation $\sim$ and type $\mathcal{I}$ are compatible, $\sim\colon \mathcal{I}$, then if we can prove that for every iteration the loop bodies are related $p \sim s$ then the loops are related, *i.e.*, $\text{for}\ x \in r\ \{p\} \sim \text{for}\ x \in r'\ \{s\}$. For each iteration, we are given that: vector I is the vector-value of variable x and the constraint on active threads C are both given by the vector-evaluation of iteration i, that is, $\langle i{+}r, \sigma\rangle \Downarrow (I, C)$; j is the scalar-value of variable x; and, both ranges are nonempty.

LEMMA 5.4. *If* $\sim$ *is a cost relation,* $\sim\colon \mathcal{I}$, $\text{dom}\,\sigma \vdash \text{for}\ x \in r\ \{p\}$, $\text{dom}\,S \vdash \text{for}\ x \in r'\ \{s\}$, $E \models r\ \mathcal{I}\ r'$, *and* $\forall i, I, C, j$

$$\langle i + r, \sigma\rangle \Downarrow (I, C) \land C \neq \mathsf{f}^{\mathsf{W}} \land \langle i + r', S\rangle \downarrow (j, \mathsf{t}) \implies (B \land C, \sigma[x \mapsto I], S[x \mapsto j]) \models p \sim s$$

then $(B, \sigma, S) \models \text{for}\ x \in r\ \{p\} \sim \text{for}\ x \in r'\ \{s\}$.

Proof of loop analysis of Example 4.1. Our goal is to prove the derivation of Equation (6) ($\emptyset; \mathsf{t} \vdash_{=} 0..\text{tid} \approx 0..\mathsf{W} - 1 : \mathsf{D}$). This proof demonstrates our analysis's ability to handle thread-divergent loop bounds where different threads execute different numbers of iterations. The key insight is showing that despite the divergent execution, the total iteration count remains predictable and matches the sequential case. We prove a slightly stronger result following.

Lemma 5.5. *If* $\sim : \mathcal{I}$, *then* $\Gamma; \Phi \vdash_{\sim} (i..\mathsf{tid})\ \mathcal{I}\ (i..(\mathsf{W}-1)) : \mathsf{D}$ *for any* i.

Proof. The proof strategy is similar to that of Lemma 5.6. The interesting case is showing that $\max(I_{\mathbb{N}} - i^{\mathsf{W}}) = \mathsf{W} - 1 - i$, which correspond to the number of iterations for vector and scalar ranges, respectively. □

Proof of loop analysis of Example 4.2. Our goal is to prove the derivation of Equation (8) ($\emptyset; \mathsf{t} \vdash_{\leq} 0..\mathsf{w} \approx 0..\mathsf{w} : \mathsf{U}$). This proof establishes the correctness of our analysis for uniform loops where all threads execute the same iteration space. The key contribution is showing that uniformity guarantees identical iteration counts between vector and scalar executions, enabling precise cost analysis. We prove a stronger result, if the vector-range r is uniform and $r = r'$, then the following loop analysis holds.

Lemma 5.6 (Uniform-loop analysis). *If* $\Gamma \vdash r : \mathsf{U}$ *and* $\sim : \mathcal{I}$, *then* $\Gamma; \Phi \vdash_{\sim} r\ \mathcal{I}\ r : \mathsf{U}$.

Proof. We give a proof intuition, as this proof requires introducing multiple new judgments that distract from the focus of the paper. First, we define a judgment that yields the number of iterations of a range. Next, we show that if the vector r and scalar r' ranges have the same number of iterations under an environment E, then $\Gamma; \Phi \vdash_{\sim} r\ \mathcal{I}\ r'$. Then, from $\Gamma \vdash r : \mathsf{U}$ we get $E \models r\,\mathsf{unif}$. Finally, we show that if $E \models r\,\mathsf{unif}$, then r has the same length under vector and scalar evaluation. □

5.3 Metric Analysis

In this section, we define the metric analysis judgment and prove the metric analyses for our running examples (Examples 4.1 and 4.2). To make our type system support arbitrary metrics and cost relations, we depart from the usual way of defining type systems for this judgment. Rather than resorting to syntactic rules, we define the metric analysis in terms of the dynamic relational-cost. Judgment $\Gamma; \Phi \vdash M(n) \sim m$ requires that for any compatible environment E (where $\Gamma; \Phi \bowtie E$), the dynamic cost relation $E \models \mathsf{A}[n] \sim \mathsf{tick}(m)$ holds.

Definition 5.7 (Metric analysis). A metric analysis is parametric on metric M and on relation $\sim$.

$$\Gamma; \Phi \vdash M(n) \sim m \overset{\text{def}}{=} \forall E \colon \Gamma; \Phi \bowtie E \implies E \models \mathsf{A}[n] \sim \mathsf{tick}(m)$$

The compatibility constraint is what makes metric analysis proofs possible. For instance, we cannot show $E \models \mathsf{A}[0] = \mathsf{W} - \mathsf{x}$ for an arbitrary E, as we have no guarantee that variable x respects its type range $0..\mathsf{W}-1$ and no information about which threads are active.

Verification of running examples. We prove that the metric analyses from Examples 4.1 and 4.2 hold under compatible environments. The proof of the former example showcases our analysis's expressiveness in capturing thread-divergent costs. We can statically predict runtime costs that vary dynamically as threads diverge during execution. Moreover, the following proof demonstrates how compatibility enables us to establish the exact equality.

Lemma 5.8. *Proposition* $\emptyset, \mathsf{x} \colon \mathsf{D}_{(0..\mathsf{tid}, 0..\mathsf{W}-1)}; \mathsf{t} \vdash \mathsf{M}_{\mathsf{E}}(0) = \mathsf{W} - \mathsf{x}$, *from Equation (7), holds.*

Proof. It is enough to show that given $\emptyset, \mathsf{x} \colon \mathsf{D}_{(0..\mathsf{tid}, 0..\mathsf{W}-1)}; \mathsf{t} \vdash E$ and $\langle \mathsf{W} - \mathsf{x}, S \rangle \downarrow i$ where $E = (B, \sigma, S)$, then we must show $\mathsf{count}(B, \mathsf{t}) = i$. Let $S(\mathsf{x}) = i_{\mathsf{x}}$ and $\sigma(\mathsf{x}) = I_{\mathsf{x}}$. From $\langle \mathsf{W} - \mathsf{x}, S \rangle \downarrow i$ we can get that $\mathsf{W} - i_{\mathsf{x}} = i$. We can do a case analysis test whether the value of x (*i.e.*, i_{x}) falls within the loop range $0..\mathsf{W}-1$, given that we have $\langle \mathsf{x} < \mathsf{W}, S \rangle \downarrow b$ for some b and we can do a case analysis on b.

Case $i_\mathsf{x} < \mathsf{W}$. We first assume that $I_\mathsf{x} = {i_\mathsf{x}}^\mathsf{W}$ to conclude our case. Afterwards, we prove $I_\mathsf{x} = {i_\mathsf{x}}^\mathsf{W}$. We have that $B = 0^\mathsf{W} \leq {i_\mathsf{x}}^\mathsf{W} \leq I_\mathbb{N}$ (from Lemma 5.2), which can be simplified to $B = {i_\mathsf{x}}^\mathsf{W} \leq I_\mathbb{N}$. Our goal becomes $\mathrm{count}({i_\mathsf{x}}^\mathsf{W} \leq I_\mathbb{N}, \mathsf{t}) = \mathsf{W} - i_\mathsf{x}$, which can be obtained by proving a more general result: $\forall l, u, x \colon l \leq u \implies \mathrm{count}((x + l)^{u+1} \leq (x + l, \ldots, x + l + u), \mathsf{t}) = (u + 1) - l$. We are left with showing that $I_\mathsf{x} = {i_\mathsf{x}}^\mathsf{W}$. Firstly, we use Lemma 5.1 to obtain $\exists j$ such that $\langle (0..\mathsf{tid}) + j, \sigma \rangle \Downarrow (I_\mathsf{x}, B')$ and $\langle (0..\mathsf{W} - 1) + j, S \rangle \downarrow (i_\mathsf{x}, \mathsf{t})$. We can now use the latter to conclude that $j = i_\mathsf{x}$. By replacing $j = i_\mathsf{x}$ in $\langle (0..\mathsf{tid}) + j, \sigma \rangle \Downarrow (I_\mathsf{x}, B')$ and inverting the latter, we conclude our remaining goal.

Case $i_\mathsf{x} \geq \mathsf{W}$. We first show that we can conclude this case given $B = \mathsf{f}^\mathsf{W}$, and then we show how to obtain $B = \mathsf{f}^\mathsf{W}$. We have that $\mathrm{count}(B, \mathsf{t}) = i \iff \mathrm{count}(\mathsf{f}^\mathsf{W}, \mathsf{t}) = \mathsf{W} - i_\mathsf{x} \iff 0 = \mathsf{W} - i_\mathsf{x}$. Thus, we are left showing that $0 = \mathsf{W} - i_\mathsf{x}$, which follows from $i_\mathsf{x} \geq \mathsf{W}$. We are left with showing that $B = \mathsf{f}^\mathsf{W}$. By doing a case analysis on $\emptyset, \mathsf{x} \colon \mathsf{D}_{(0..\mathsf{tid}, 0..\mathsf{W}-1)}; \mathsf{t} \vdash E$ understand the structure of B we get two cases to consider. Case VAR-1 (range is not empty), then we can obtain $i_\mathsf{x} < \mathsf{W}$ and reach a contradiction. Case VAR-2 (range is empty) yields assumptions $B = B_1 \wedge B_2$, $\langle \mathsf{x} \leq \mathsf{W}, \sigma \rangle \Downarrow B_1$, and (A_1) $B_1 = \mathsf{f}^\mathsf{W} \vee b = \mathsf{f}$. From Lemma 5.5 can conclude that $B_1 = \mathsf{f}^\mathsf{W} \wedge b = \mathsf{f}$ and thus $B = \mathsf{f}^\mathsf{W}$. □

The proof of Equation (9) demonstrates two key technical contributions of our approach. First, our metric analysis can express an exact cost of a computation running under a thread-divergent conditional like $\mathsf{tid} \mathbin{\%} 2 = 0$, which is a novel contribution. Second, we establish upper bounds for a complex metric (uncoalesced accesses) by leveraging the exact cost of a simpler metric (active threads). To prove that the metric analysis of this example holds, we begin by defining the uncoalesced accesses metric formally, then establish the upper bound relationship.

We can define an uncoalesced accesses metric that counts the number of distinct memory transactions (cache lines) required to satisfy the indices in I for active threads B of a warp.

$$\mathrm{ua}(B, I) = |\{\lfloor I(j)/\mathsf{C} \rfloor \mid 0 \leq j < \mathsf{W} \wedge B(j) = \mathsf{t}\}|$$

Parameter C represents the number of array elements that can be fetched in a single memory transaction and depends on the type of the array being accessed. Henceforth, and to help with comparing with related work, we set the parameter to match the choice of [45], *i.e.*, $\mathsf{C} = 4$. The relevant points to keep in mind for our analysis are that the uncoalesced accesses metric depends on which threads are active (B) and which indices are being accessed (I). Next, we show that the total number of active threads M_E is an upper bound for uncoalesced accesses.

LEMMA 5.9. *If* $\Gamma; \Phi \vdash M_\mathsf{E}(n) \leq m$, *then* $\Gamma; \Phi \vdash \mathrm{ua}(n) \leq m$.

Finally, we can prove the metric analysis of Equation (9).

LEMMA 5.10. $\emptyset, \mathsf{i} \colon \mathsf{U}_{0..i, 0..i}; \mathsf{t} \wedge \mathsf{tid} \mathbin{\%} 2 = 0 \vdash \mathrm{ua}(\mathsf{tid} \times i + \mathsf{i}) \leq \lfloor \mathsf{W}/2 \rfloor$

PROOF. By using Lemma 5.9 we simply need to show $\emptyset, \mathsf{i} \colon \mathsf{U}_{0..i, 0..i}; \mathsf{t} \wedge \mathsf{tid} \mathbin{\%} 2 = 0 \vdash M_\mathsf{E}(\mathsf{tid} \times i + \mathsf{i}) \leq \lfloor \mathsf{W}/2 \rfloor$. The rest of the proof follows a similar structure of the proof of Lemma 5.8. It is enough to show that given $\emptyset, \mathsf{i} \colon \mathsf{U}_{0..i, 0..i}; \mathsf{t} \wedge \mathsf{tid} \mathbin{\%} 2 = 0 \vdash E$ where $E = (B, \sigma, S)$, then we must show $\mathrm{count}(B, I_\mathbb{N} \mathbin{\%} 2^\mathsf{W} = 0^\mathsf{W}) = \lfloor \mathsf{W}/2 \rfloor$, which follows by induction on W. □

5.4 Soundness

In this section we prove our main result, Corollary 4.3. To this end, we introduce a more general result on soundness that applies to any compatible environment E (where $\Gamma; \Phi \vdash E$). If a protocol p and a statement s are related by $\sim$ statically ($\Gamma; \Phi \vdash p \sim s$), then the comparison holds dynamically ($E \models p \sim s$). We first present a well-formedness lemma, then our general soundness theorem with its proof by induction on the typing derivation. Finally, we restate Corollary 4.3 with its proof as a direct corollary of the general result.

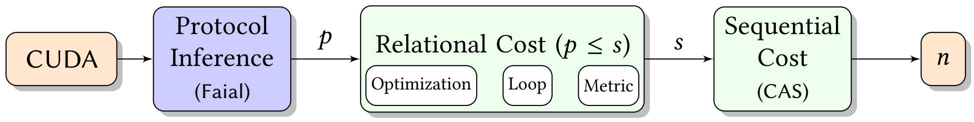

Fig. 6. Overview of the data flow of Pico.

Lemma 5.11. *Let* $E = (B, \sigma, S)$. *If* $\Gamma; \Phi \vdash E$ *and* $\Gamma; \Phi \vdash p \sim s$, *then* $\operatorname{dom} \sigma \vdash p$ *and* $\operatorname{dom} S \vdash s$.

Theorem 5.12 (Soundness). *If* $\Gamma; \Phi \vdash p \sim s$, *then for any* E *such that* $\Gamma; \Phi \vdash E$, *then* $E \models p \sim s$.

Proof. Let $E = (B, \sigma, S)$. The proof follows by induction on the derivation of $\Gamma; \Phi \vdash p \sim s$.

Case skip. Since $\langle \mathsf{skip}, B, \sigma \rangle \Downarrow_M 0$, $\langle \mathsf{tick}(0), S \rangle \downarrow 0$, and $0 \sim 0$ (from Definition 3.3), then we can conclude $E \models \mathsf{skip} \sim \mathsf{tick}(0)$.

Case acc. We get $\Gamma; \Phi \vdash M(n) \sim s$ and must show $E \models \mathsf{A}[n]s$, which follows from Definition 5.7.

Case seq. We have induction hypotheses $E \vdash p_1 \leq s_1$ and $E \vdash p_2 \leq s_2$. Our goal is to show $E \vdash p_1; p_2 \leq s_1; s_2$. From $E \models p_1 \sim s_1$ we get $\langle p_1, B, \sigma \rangle \Downarrow_M i_1$ and $\langle s_1, S \rangle \downarrow j_1$ where the two costs are in the relation $i_1 \sim j_1$. From $E \models p_2 \sim s_2$ we get $\langle p_2, B, \sigma \rangle \Downarrow_M i_2$ and $\langle s_2, S \rangle \downarrow j_2$ where the two costs are in the relation $i_2 \sim j_2$. To conclude, we must show $(i_1 + i_2) \sim (j_1 + j_2)$, which holds since $\sim$ is closed under addition.

Case if-u. We have $\Gamma \vdash c\colon \mathsf{U}$, our induction hypothesis is $E \models p \sim s$, and we must show $E \models \mathsf{if}\ (c)\ \{p\} \sim \mathsf{if}\ (c)\ \{s\}$. We can get that $\langle c, \sigma \rangle \Downarrow b^{\mathsf{W}}$ and $\langle c, S \rangle \downarrow b$ from $\Gamma \vdash c\colon \mathsf{U}$. If $b = \mathsf{t}$ then we can conclude by using our induction hypothesis. Otherwise, $b = \mathsf{t}$ and we can conclude since $0 \sim 0$.

Case if-d. We have $\Gamma \vdash c\colon \mathsf{U}$, our induction hypothesis is $\forall E'\colon \Gamma; \Phi \wedge c \vdash E' \implies E' \models p \sim s$, and we must show $E \models \mathsf{if}\ (c)\ \{p\} \sim s$. We have that there exists a vector B such that $\langle c, \sigma \rangle \Downarrow C$, thus we can obtain $(B \wedge C, \sigma, S) \models p \sim s$ from our induction hypothesis and Rule cnd. The remainder of the case uses the rules for if-t or if-f of Figure 2, by doing a case analysis on $C = \mathsf{f}^{\mathsf{W}}$.

Case for. We start by applying Lemma 5.4, Lemma 5.11, and our induction hypothesis. We get $\Gamma, x\colon \tau; \Phi \vdash (B \wedge C, \sigma[x \mapsto I], S[x \mapsto j])$, which follows from using Rule for-2.

□

Corollary 4.3 (Soundness for closed terms). *Let* M *be a metric and* $\sim$ *a cost relation. If* $\emptyset; \mathsf{t} \vdash p \sim s$ *then* $(\mathsf{t}^{\mathsf{W}}, \emptyset, \emptyset) \models p \sim s$.

Proof. This follows directly from the general soundness theorem above with the empty compatible environment $\emptyset; \mathsf{t} \bowtie (\mathsf{t}^{\mathsf{W}}, \emptyset, \emptyset)$. □

6 Pico: **Cost Analysis for GPU Kernels**

Pico implements resource cost analysis to estimate the performance costs of bank conflicts on shared memory and uncoalesced accesses on global memory in GPU kernels. The data flow of our tool is depicted in Figure 6. Pico uses Faial [19] to convert a CUDA kernel into a memory access protocol p (Section 6.1), which then goes through two main transformations to yield a symbolic cost n: (Section 6.2) *relational cost generation* that converts a protocol p into a sequential statement s such that $p \leq s$, and (Section 6.3) *sequential cost analysis* that takes a sequential statement s and computes an upper-bound cost n, using a computer algebra system (CAS), or an off-the-shelf cost analysis tool. We conclude by detailing the implementation of our metric analyses in Section 6.4 and our exactness check for precision analysis in Section 6.5. *Notably, our translation to CAS does not add approximations to the analysis, these can only result from the relational-cost generation step.*

6.1 Translating CUDA into Memory Access Protocols

Pico uses Faial to translate CUDA kernels into Memory Access Protocols by parsing the source code with libclang [36] and extracting memory accesses with their surrounding control-flow from the AST. Accurate cost analysis for our target metrics (uncoalesced accesses and bank conflicts) depends on preserving this control-flow in the protocol.

The cost analysis becomes inexact (*i.e.*, the cost relation becomes $\leq$) when program variables are over-approximated, yielding upper bounds instead of exact costs. The translation introduces over-approximations when: variable updates cannot be captured (*e.g.*, unsupported mutation patterns or external function calls), loop bounds are not found (*e.g.*, unsupported `while`-loops), or indices are computed from array values. For instance, an array read in a conditional such as `if(A[x]>0)` is over-approximated as `if($A1 > 0)`, where `$A1` is an *unconstrained* variable that weakens the conditional. Similarly, a read used as an index in `B[A[x]]` is over-approximated as `B[$A2]`, weakening the metric analysis. Liew *et al.* [43] discuss sources and impact of over-approximation in memory access protocols. In Section 6.5, we develop an exactness analysis to measure how frequently over-approximated variables impact conditionals and loops, with results in Section 7.

Faial supports CUDA features essential for memory cost analysis: barrier synchronization, atomics, memory spaces (global and shared memory), and multi-dimensional thread identifiers. Supported C++ features include templates, array aliasing, bitwise operations, inter-procedural calls, `for`/`do`/`while`-loops, and kernel parameters. Unsupported features include memory fences, warp-level communication primitives, and the thread-cooperative runtime. We direct the interested reader to [19, §6] and [43, §5.1] for a detailed discussion of supported features.

6.2 Relational-Cost Generation

Generating a statement s from a protocol p such that $p \leq s$ consists of three stages: optimization, loop analysis, and metric analysis. This is done by traversing the protocol p recursively and using the rules from Figure 4 to generate s.

Optimization. Pico first prepares protocols for analysis through array filtering and constant folding to increase precision and simplify other stages of the cost analysis. This optimization stage takes a protocol as input and outputs an optimized protocol. Arrays are filtered by their memory type (shared memory, global memory, etc.) to select relevant memory locations — shared memory arrays for bank conflict analysis, global memory arrays for uncoalesced access analysis. Then, Pico performs constant folding and propagation when there are known parameters, such as block dimensions, grid dimensions, and kernel parameters. Constant folding eliminates conditionals and unrolls loops while increasing precision in the metric analysis by replacing variables with their values whenever possible.

Loop analysis. When traversing the protocol to generate a program, our loop analysis typechecks each vector-range r and matches on the resulting type. **Thread-uniform ranges.** If the vector-range r is thread-uniform (type $\mathtt{U}$), then our algorithm proceeds with Lemma 5.6, which states $\Gamma; \Phi \vdash_{\sim} r \approx r : \mathtt{U}$. That is, our algorithm takes a uniform loop r, and outputs the same range r as scalar. **Thread-divergent ranges.** If the vector-range $r = n..m$ is thread-divergent (type $\mathtt{D}$), then our algorithm replaces tid in r by $\max(m-n)$, maximizing the iteration space. For example, Lemma 5.5 shows that $\Gamma; \Phi \vdash_{\sim} (i..\mathsf{tid})\ \mathcal{I}\ (i..(\mathsf{W}-1)) : \mathtt{D}$, so maximizing $\mathsf{tid} - i$ yields $r' = i..(\mathsf{W}-1)$. In more detail, our algorithm outputs a scalar range r' such that $\Gamma; \Phi \vdash_{\sim} n..m \approx r' : \mathtt{D}$ when all free variables are thread-uniform, *i.e.*, $\forall x \in \mathrm{fv}(n..m) \implies \Gamma(x) = \mathtt{U}$. The procedure is to replace tid with the following optimization problem (handled by an off-the-shelf SMT solver): what value of tid

maximizes the number of iteration $m - n$, formally:

$$\max_{\text{tid}}(m - n)$$

The constraint $\forall x \in \text{fv}(n..m) \implies \Gamma(x) = \mathsf{U}$ is necessary, otherwise the optimization may generate potentially impossible valuations in the output range.

Metric analysis. When traversing the protocol to generate a program, our metric analysis processes each memory access to generate cost annotations. Since memory access protocols support multi-dimensional array accesses, and metric analyses are defined in terms of hardware-level accesses, Pico translates each memory access from source-code-level to hardware-level as follows. First, by flattening multi-dimensional array accesses, then by applying data-type multipliers to match how GPU hardware addresses memory. For instance, when an array `A` is declared as `int A[512][32]`, a source-code access `A[i][j]` becomes a hardware access `A[(i*32 + j) * sizeof(int)]`. Pico then invokes a metric analysis to generate a cost, as per Definition 5.7.

6.3 Sequential Cost Analysis

This stage takes a sequential statement s and computes its cost n. Pico provides multiple backends: (*i*) our bespoke solution that relies on a Computer-Algebra-System (CAS) based solver, and (*ii*) bridges from a resource cost statement into off-the-shelf sequential cost analyses. Currently Pico supports Absynth [47], CoFloCo [25], and KoAT [26].

Loop ranges in memory access protocols are normalized, which greatly simplifies our cost analysis. Protocols unify for-loops, while-loops, and do-loops into a single representation. Loop ranges further distinguish the stride in the following ways: direction (increasing, decreasing), type (additive, multiplicative), and value (the amount added or multiplied per iteration).

The separation of direction from type, combined with Resource Calculus having no side effects (no early returns, no mutation), simplifies the problem of handling different loop ranges in cost analysis. Instead of handling each loop operator separately (*e.g.*, addition, subtraction, etc.) the direction can be ignored, reducing the problem to two patterns: additive and multiplicative ranges.

The CAS backend translates a statement s into a symbolic algebraic expression that is then simplified symbolically with Maxima [40]. Using a resource with $\mathsf{tick}(n)$ becomes expression n, loops become summations, and sequencing becomes addition. There is a mismatch between loop ranges (which can have arbitrary step values) and mathematical summations (which iterate over every element in the range). To bridge that gap, we apply loop index normalization [41], a standard compiler technique, to transform each range into an equivalent range with unit stride[1] (additive type, value of 1), as follows. For additive steps, we replace the loop variable with a multiplication: a loop `for(x=0; x<n; x+=k) S` becomes `for(y=0;y<n/k; y++)S[x` $\mapsto$ `y*k]`. For multiplicative steps, we replace the loop variable with exponentiation: a loop `for(x=1; x<n; x*=k)S` becomes `for(y=`$\log_k$`(1); y<`$\log_k$`(n); y++)S[`$x \mapsto k^y$`]`. Maxima applies symbolic simplification to the resulting expression with tactics `logcontract`, `simpsum`, and `ratsimp`.

The Absynth backend provides a straightforward translation of Resource Calculus statements into Absynth's imperative language, since Absynth can directly express all Resource Calculus constructs. The only minor transformation needed is converting for-loops into while-loops.

Discussion. In our experience, the CAS backend can handle more inputs than Absynth. Overall memory utilization is about slightly higher with Maxima (around 6–10%). Absynth exhibits a greater range in execution time and Maxima had a more stable execution time, as we will see in our evaluation (Table 1). In terms of inputs, Maxima supports the various mathematical expressions

[1]We remind that in the formalism of this paper loop ranges have a stride of 1 too.

```
// test-1
for(i=0;i<(ub>>1);i++)
  g[tid] = i;
```

```
// test-2
for(i=0;i<ub;i+=st)
  g[tid] = i;
```

```
// test-3
for(i=lb;i<ub;i*=2)
  g[tid] = i;
```

```
// test-4
for(i=0;i<ub;i++)
  for(j=0;j<ub;j++)
    for(k=0;k<ub;k++)
      g[tid] = i;
```

Fig. 7. Unsupported features in Absynth.

being generated, so more challenging inputs just yield symbolic expressions that Maxima may not be able to simplify. Absynth cannot analyze programs with certain numeric expressions and control-flow patterns, which we document in Figure 7. We identified the following list of unsupported features: bitwise operators—commonly used in GPU programs (`test-1`); variables cannot appear in the step increment (`test-2`); loops that use multiplication or division in the step statement cannot have variables in the initializer (`test-3`); three or more nested loops (`test-4`). We note that every limitation of Absynth is also shared by RaCUDA.

6.4 Metric Analyses Implementation

Uncoalesced memory accesses. This metric counts the number of distinct memory transactions (cache lines) required when threads access memory indices that cannot be combined into a single transaction. We restate the uncoalesced metric for a global parameter C:

$$\mathrm{ua}(B, I) = |\{\lfloor I(j)/\mathrm{C} \rfloor \mid 0 \le j < \mathrm{W} \land B(j) = \mathrm{t}\}|$$

This metric analysis handles five cases in order, given an index n of type τ and let i_W be result cost of the metric analysis of M_E (enabled threads). **Case 1:** When $\tau = \mathtt{U}$ the cost is 1. **Case 2:** When n is of the form $m \times \mathrm{tid}$ and m is of type $\mathtt{D}$, then cost is $\max(m, i_\mathrm{W})$. **Case 3:** When $\tau = \mathtt{D}$, the expression n is simplified by removing memory-aligned sub-expressions (multiples of word boundaries like 32 bytes) and then simulated (explained after). **Case 4:** When n is of the form $n_1 + n_2$ where the type of n_1 is divergent ($\mathtt{D}$), n_2 is uniform ($\mathtt{U}$), and n_2 contains a variable, then n is potentially misaligned. In which case, we obtain the cost n_1' of n_1 (eliding n_2) and return $\max(n_1' + 1, i_\mathrm{W})$. **Case 5:** Otherwise, return i_W (the worst case cost).

Bank conflicts. This metric counts the maximum number of threads accessing the same memory bank simultaneously, representing the extra transactions required to resolve conflicts when multiple threads compete for the same bank. First, we define the bank conflicts metric. Let bank(i, I, B) give the set of threads accessing bank i. The bank conflicts metric bc(B, I) returns the maximum number of banks and subtracts 1, so that the metric only accounts for the number of extra transactions performed (*i.e.*, the number of conflicts).

$$\mathrm{bank}(i, I, B) = \{I(j) \mid 0 \le j < \mathrm{W} \land I(j) \bmod \mathrm{B} = i \land B(j) = \mathrm{t}\}$$

$$\mathrm{bc}(B, I) = \max\{|\mathrm{bank}(0, I, B)|, \ldots, |\mathrm{bank}(\mathrm{B} - 1, I, B)|\} - 1$$

Our metric analysis handles four cases in order, given an index n of type τ and let i_W be result cost of the metric analysis of M_E (enabled threads). **Case 1:** When $\tau = \mathtt{U}$ the cost is 0. **Case 2:** When $n = n_1 + n_2$ and the type of n_2 is $\mathtt{U}$, then return cost of recursively analyzing n_1. Since simulation is only possible for closed expressions, this step eliminates as many program variables as possible, by removing constant offsets. The simplification is possible because the cost of $n_1 + n_2$ equals the cost of n_1 when n_2 is uniform, as the bank conflicts metric depends only on the number of unique banks accessed. For instance, the accesses `A[threadIdx.x + 7]` and `A[threadIdx.x]` have identical conflict patterns with banks just relabeled. Formally, for any bank i, the identity

$\mathrm{bank}(i, I + j^{\mathsf{W}}, B) = \mathrm{bank}((i+j) \bmod \mathsf{B}, I + j^{\mathsf{W}}, B)$ holds. **Case 3:** The cost is the result of simulating the metric (see following paragraph). **Case 4:** Otherwise, return $i_{\mathsf{W}} - 1$ (the worst case cost).

Metric analysis by simulation. Simulation works by evaluating the active thread condition to determine which threads are active B, evaluating the index expression to get the memory access pattern I, and then applying the cost metric $M(B, I)$ to compute the actual cost, formally to obtain $\Gamma; \Phi \vdash M(n) \leq m$ for any metric M. We can evaluate the condition of active threads, *i.e.*, $\langle \Phi, \emptyset \rangle \downarrow B$ (from Lemma 5.2), and evaluate the index $\langle n, \emptyset \rangle \downarrow I$ and then return the cost $m = M(B, I)$.

6.5 Exactness Check

To help users determine whether computed cost bounds are exact or affected by analysis over-approximations, we introduce an *exactness check* that analyzes the precision of the resulting cost. Our soundness theorem (Theorem 5.12) shows that the cost relation $\sim$ between protocol and program depends directly on what the loop and metric analyses can prove. In our experiments, we observed that the loop analysis of Pico can always generate a scalar range with the same iteration length ($\approx$) as the vector range. Given that type $\approx$ imposes no constraints on the cost relation (*c.f.*, judgment $\sim\colon \mathcal{I}$), in practice the precision of Pico solely depends on the precision of the metric analyses being used.

Our exactness check detects whether the active thread condition (*c.f.*, Lemma 5.2) can be used by the metric analysis or if the condition needs to be strengthened, thereby introducing an approximation (*i.e.*, approximation means more threads, so a potentially-higher cost). We say that a typing environment Γ is inexact, when there is any thread-divergent binding ($\mathtt{D}$). For instance, $\emptyset, \mathtt{x}\colon \mathtt{D}$ is inexact. We say that a constraint environment Φ is inexact, when there exists a thread divergent condition c that contains a free variable of type $\mathtt{D}$. For instance, $\mathtt{x} > 10$ is inexact when $\mathtt{x}$ is thread-divergent. For instance, the environment $\emptyset, \mathtt{x}\colon \mathtt{D}; \mathtt{t}$ from Equation (7) is inexact, and the environment $\emptyset, \mathtt{i}\colon \mathtt{U}; \mathtt{tid} \mathbin{\%} 2 = 0$ from Equation (9) is exact.

Simulation metric analysis requires closed expressions to evaluate costs, but our environment may contain program variables that prevent simulation from proceeding. A naive approach would assume all threads are active and yield sound, yet higher costs. Instead, our implementation uses the exactness check to selectively discard only the inexact constraints, reducing the approximation error. For bank conflicts and uncoalesced access metrics, this selective constraint removal is sound because dropping constraints can only increase the estimated number of active threads, providing a safe upper bound without introducing unsoundness. For instance, given environment $\emptyset, \mathtt{x}\colon \mathtt{D}, \mathtt{y}\colon \mathtt{U}; \mathtt{tid} \mathbin{\%} 2 \land \mathtt{x} > 10$, our algorithm retains the exact constraint $\mathtt{tid} \mathbin{\%} 2$ while discarding the inexact constraint $\mathtt{x} > 10$, yielding $\emptyset, \mathtt{y}\colon \mathtt{U}; \mathtt{tid} \mathbin{\%} 2$.

7 Evaluation

We consider two key questions in evaluating the effectiveness of the cost analysis:

§7.1 How does Pico compare to the state of the art, RaCUDA?
§7.2 How frequently does control flow affect the accuracy of Pico?

We answer each question with a different experiment. First, we reuse the evaluation conducted in [45] to compare RaCUDA and Pico for performance and correctness in analyzing uncoalesced accesses and bank conflicts. Second, we evaluate our model's effectiveness by finding the ratio of precisely analyzable structured loops and boolean expressions in a well-known dataset.

Experimental setup. The experimental tests ran on a machine with 8 GB of memory and an AMD Ryzen 7 5800X 8-core processor. Due to the many configurations to compare, we wrote an automated testing program to ensure that tools were fairly executed and compared.

7.1 How Does Pico Compare to the State of the Art, RaCUDA?

We compare Pico against RaCUDA by evaluating both supported CUDA features and the precision of predicted bounds. We first discuss expressiveness differences, then present a quantitative comparison reproducing the experiment from [45].

Expressiveness differences. Pico is the first tool to introduce support for thread-divergence (discussed in Section 6.2); RaCUDA simply rejects any program with thread-divergence in loops and ignores any thread-divergence appearing in conditionals. Pico also supports multiplicative and bitwise loop steps, whereas RaCUDA treats multiplicative loops as additive loops, *e.g.*, treating range `(i=1;i<512;i*=2)` as `(i=1;i<512;i++)`. Pico supports C++ features including templates, inter-procedural analysis, and array aliasing through Faial, which RaCUDA does not support. RaCUDA has additional limitations with variables in loop step increments and analyzing multiple nested loops (see Section 6.3).

Quantitative comparison. We use the Absynth backend for Pico so both tools use the same cost calculation algorithm. While the cost calculation is shared, each tool generates its own program representation for Absynth, isolating differences in how GPU-specific features are handled. The experiment is parameterized by two metrics. The first metric runs each tool to predict the bounds of uncoalesced accesses. The second metric runs each tool to predict the bounds of bank conflicts. A lower predicted bound is better, unless the analysis is unsound, in which case we give a justification. We manually validate every predicted bound. The dataset consists of 15 kernels hand-crafted by the authors of [45]. We omit kernel mandelbrot from the experiment in [45], as the program is not included in the paper's artifact. In the bank conflicts experiment, we omit any kernel that does not declare shared memory, because the bank conflicts only occur in shared memory. The dataset includes example kernels for matrix multiplication, vector addition, histogram, reduction operations, and addition on subarrays. Kernel vectorAdd is an easy to analyze and verify test of addition over several arrays with a loop to show matrixMul, matrixMulBad, and matrixMulTranspose each feature matrix multiplication which requires a great deal of array accesses across different dimensions. The reduce family of kernels combine values by addition over a memory into shared memory. Kernel histogram bins then combines values in both global and shared memory. The addsub family of kernels adds values from portions of arrays with analyzable strides patterns. Table 1 summarizes the predicted bounds of both experiments. As an example, the first row reports on the kernel matrixMul which was analyzed using the uncoalesced access metric and a cost formula dependent on *wA* and a runtime of less than half a second. Table 1 reports on both RaCUDA and Pico.

Discussion. Pico produces the best results in every run, outperforming RaCUDA in 10 runs (1.7× better). Notably, RaCUDA predicts 4 incorrect upper bounds and fails to run 2 kernels. *Sources of over-approximation:* Because our analysis takes into consideration some thread-divergence expressions, Pico correctly predicts the cost of broadcasts with a cost of 1 (tag `C`), whereas RaCUDA identifies a cost of 2. *Incorrect bounds in RaCUDA:* In 3 kernels (tagged with `M`), RaCUDA incorrectly assumes that the array access is always aligned and does not span a memory bank boundary. For instance, given index `wB * threadIdx.y + threadIdx.x`, RaCUDA predicts a cost of 4 and yet when `wb` = 100 and `threadIdx.y` = 1 the program yields a cost of 5. In 1 kernel (tagged with `R`) RaCUDA also predicts an incorrect upper bound because for expression `data%256` it reports a bank-conflict of 7, when it should be 31, given that `data` could be distinct for each thread. *Failed bounds:* RaCUDA aborts the analysis when an access appears within a loop that decrements its loop-variable with a division, say, `x /= 2`.

Table 1. Predicted bound for uncoalesced accesses and bank conflicts for RaCUDA and Pico. The first three columns list: Kernel for the program name, LOC for the kernel's line count, and M for the metric name (U for uncoalesced, and B for bank conflicts). Then, for each tool, we have 3 columns. The first column summarizes the outcome with a symbol, where ✓ denotes lowest predicted bound, M denotes **incorrect bound** due to memory alignment, D denotes over-approximation due to thread divergence, C denotes over-approximation due to broadcast, N denotes over-approximation due to warp-uniform data, E denotes error, and R denotes **incorrect bound** due to arithmetic. The second column is the bound. The third column is the tool's execution time in seconds. The last row tallies the kernels that exhibit the lowest predicted bound for each tool.

Kernel	LOC	M		RaCUDA			Pico	
matrixMul	127	U	M	0.31 wA + 14.0	0.2	✓	0.31 wA + 15.0	0.3
matrixMulBad	120	U	M	15 wA + 465	9.6	✓	16.0 wA + 496.0	0.5
matrixMulTranspose	126	U	M	0.31 wA + 14.0	0.2	✓	0.31 wA + 15.0	0.3
vectorAdd	42	U	✓	12	0.0	✓	12	0.3
reduce0	55	U	✓	5	0.1	✓	5	0.3
reduce1	55	U	✓	5	0.1	✓	5	0.3
reduce2	53	U	✓	5	0.1	✓	5	0.3
reduce2a	54	U	✓	5	0.1	✓	5	0.4
reduce3	59	U	✓	9	0.1	✓	9	0.2
reduce3a	60	U	✓	9	0.1	✓	9	0.3
histogram256	57	U	✓	0.08 d + 49.9	1.2	✓	0.08 d + 49.9	0.3
addSub0	10	U	D	132 w	0.3	✓	66w	0.5
addSub1	7	U	C	132 w	0.0	✓	130w	0.4
addSub2	7	U	✓	14 h + 14	0.0	✓	14 h + 14	0.3
addSub3	9	U	✓	10 h + 14	0.0	✓	10 h + 14	0.3
matrixMul	127	B	N	31 wA + 961	0.1	✓	0	0.3
matrixMulTranspose	126	B	N	31.97 wA + 991.1	0.2	✓	0.97 wA + 30.0	0.3
reduce0	55	B	✓	0	0.0	✓	0	0.2
reduce1	55	B	✓	23715	0.0	✓	23715	0.3
reduce2	53	B	E	n/a	1.4	✓	0	0.2
reduce2a	54	B	✓	0	0.1	✓	0	0.2
reduce3	59	B	E	n/a	1.5	✓	0	0.3
reduce3a	60	B	✓	0	0.1	✓	0	0.4
histogram256	57	B	R	0.88 d + 55.0	1.4	✓	3.88 d + 244.0	0.3
addSub3	9	B	✓	0	0.0	✓	0	0.2
Total:			15			**25**		

7.2 How Frequently Does Control Flow Affect the Accuracy of Pico?

To verify the accuracy of our analysis in a broader context, we are interested in knowing the ratio of loops and conditionals our exactness check deems exact (Section 6.5); Pico is dependent on this precision for tight bounds. In this experiment, we use the CAS backend, and we target a well-studied data set of 226 kernels. We report the statistics on the frequency of loops and conditionals that can be exactly analyzed per metric. Our Absynth backend was unable to analyze the whole dataset and was therefore excluded (see Section 6.3).

Data selection. The CAV'14 [4] dataset is a benchmark suite of 226 CUDA kernels from 4 benchmark suites: NVIDIA GPU Computing SDK v2.0 (8 kernels), NVIDIA GPU Computing SDK v5.0 (165 kernels), Microsoft C++ AMP Sample Projects (20 kernels), gpgpu-sim benchmarks [3] (33 kernels). The CAV'14 dataset includes 1812 array accesses, 471 structured loops, and 670 conditional

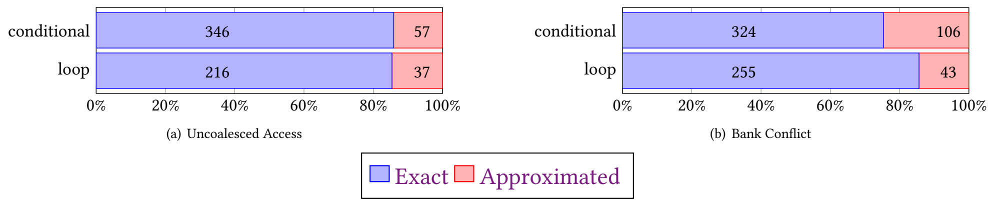

Fig. 8. Ratio of exact vs. approximated analysis by metric in CAV'14 dataset.

expressions. We count each binary boolean operation in the boolean expression of a conditional separately. Figure 8 summarizes the statistics which includes the number of loops and boolean sub-expressions across all files of the dataset that can be analyzed exactly. We separate the results per metric, since either metric focuses on different types of memory and performance issues, and are consequently, not directly comparable.

Discussion. At least 75.3% of conditionals and at least 85.4% loops can be precisely captured across metrics, which highlights the overall accuracy of Pico. We find that, in uncoalesced accesses, 346 out of 403 (85.9%) conditionals are exact, and 216 out of 253 (85.4%) loops are exact. With regard to bank conflicts, we find that 324 out of 430 (75.3%) conditionals are exact, and 255 out of 298 (85.6%) loops are exact.

8 Related Work

Performance analysis of GPU kernels. PUG was the first static performance analysis for GPU kernels (bank conflicts and uncoalesced accesses) using SMT-encoding [37, 38]. GKLEE uses concolic execution to detect bank conflicts [39]. GPUDrano detects uncoalesced accesses by abstract interpretation [1, 2]. Horga *et al.* use symbolic execution and genetic algorithms to find bank conflicts in cryptographic algorithms [30, 31]. These analyses can *detect* performance bottlenecks but cannot quantify their impact. RaCUDA proposed the first cost analysis for GPU kernels [45, 46], supporting warp-parallel metrics (divergent warps, bank conflicts, uncoalesced accesses) and task-parallel metrics (work and span, *c.f.*, [11, 12]). RaCUDA uses automated amortized resource analysis (AARA) [28, 29] that only captures upper bounds, while our relational cost analysis produces exact bounds, lower bounds, and upper bounds. A key technical distinction is loop handling: AARA requires invariants (being Hoare-logic based), while ours does not. We suspect invariant requirements complicate handling thread-divergent loops in RaCUDA, which are currently unsupported.

Cost analysis. RelCost was the first to introduce relational cost analysis [16]. RelCost establishes a precise bound on the difference between execution of two programs of the same language. ARel extends RelCost with array datatypes [49]. An important distinction in our work, is that our relational judgment takes two distinct computational models (one parallel/vectorial and one scalar), whereas the relational judgment in RelCost/ARel takes two terms of the *same* computational model. Several approaches already exist to solve cost analysis of sequential programs. The approaches listed here are non-relational cost models. KoAT analyzes the complexity of integer programs providing upper runtime and size bounds for parts of programs through control-flow graph refinement [26, 44]. CoFloCo also provides a resource analysis compositionally through control-flow refinement [25]. Absynth estimates costs using an automatic amortized resource analysis based on weakest preconditions in the analyzed program [47].

Static analysis of GPU kernels. Faial uses memory access protocols to handle data-races in CUDA [19, 20, 42, 43]. GPUVerify checks for data-races and barrier divergence in CUDA and OpenCL kernels [4–6, 8, 9, 21]. Ferrel *et al.* formalize CUDA assembly semantics in Coq [24]. VeriCUDA [34, 35] focuses on reasoning about the functional correctness using Hoare-logic. VerCors [14] uses separation logic to prove the functional correctness and data-race freedom.

9 Conclusion and Future Work

We proposed the first static cost analysis that can produce exact cost for warp-parallelism, with thread divergence support. Our analysis is parametric on resource metrics M and supports key GPU performance bottlenecks (bank conflicts, and uncoalesced accesses). Our technique uses relational-cost analysis, translating GPU kernels into sequential programs. Since the symbolic cost of generated programs can always be precisely derived and simplified with a CAS solver (*c.f.*, Section 6.3), we isolate approximations to the translation process (*i.e.*, the relational analysis). Corollary 4.3 establishes soundness for any cost relation $\sim$, including exact cost ($=$), over-approximations ($\leq$), and under-approximations ($\geq$). We provide mechanized proofs of all results in Rocq.

Our theory was implemented in Pico, as the first sound and exact cost analysis tool for GPU kernels with thread divergence support. We introduce a novel loop analysis that relies on an SMT solver to handle thread-divergent loops, as well as multiple heuristics to quantify uncoalesced accesses and bank conflicts. Pico features several alternatives to derive a symbolic cost, our CAS-based solution, and translators for Absynth, CoFloCo, and KoAT. We evaluate Pico in two experiments: we show Pico's improvements over the state of the art, and show the accuracy of our cost analysis. Our first experiment reproduces a benchmark from [45]: Pico achieves the lowest bound in all 25 kernels, outperforming RaCUDA in 10 kernels (1.7× better). Our second experiment assesses whether loops and conditionals can be handled accurately by Pico, considering a well-studied dataset [4] of 226 kernels. We found that at least 75.3% of conditionals can be precisely captured, and at least 85.4% loops can be precisely captured, which highlights the overall accuracy of Pico.

Future work is to enhance both correctness and usefulness of cost analyses. We will formally prove the correctness of metrics we have implemented, and explore trade-offs between accuracy and performance. Our framework currently reports aggregate costs for entire kernels or memory accesses, but providing per-component breakdowns could help users identify specific performance bottlenecks caused by over-approximation or high variability. We plan to study alternative reporting strategies for better static understanding of performance bottlenecks.

Acknowledgments

We thank the anonymous reviewers for steadfast attention and numerous helpful remarks. This material is based upon work supported by the National Science Foundation under Grant No. 2204986. We thank Francis Alcos, Ayden Diel, Samyak Gangwal, Austin Guiney, Ramsey Harrison, and Paul Maynard for their contributions to Faial and Pico.

References

[1] Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. 2017. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs. In *Proceedings of CAV (LNCS, Vol. 10426)*. Springer, Berlin, Heidelberg, 507–525. doi:10.1007/978-3-319-63387-9_25

[2] Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. 2021. Static detection of uncoalesced accesses in GPU programs. *Formal Methods in System Design* (2021). doi:10.1007/s10703-021-00362-8

[3] Ali Bakhoda, George Yuan, Wilson Fung, Henry Wong, and Tor Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. *ISPASS 2009 - International Symposium on Performance Analysis of Systems and Software*, 163 – 174. doi:10.1109/ISPASS.2009.4919648

[4] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. 2014. Engineering a Static Verification Tool for GPU Kernels. In *Proceedings of CAV*, Vol. 8559. Springer, Berlin, Heidelberg, 226–242. doi:10.1007/978-3-319-08867-9_15

[5] Ethel Bardsley, Alastair F. Donaldson, and John Wickerson. 2014. KernelInterceptor: Automating GPU Kernel Verification by Intercepting Kernels and their Parameters. In *Proceedings of IWOCL*. ACM, New York, NY, USA, Article 7, 5 pages. doi:10.1145/2664666.2664673

[6] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of FMCO*. Springer, Berlin, Heidelberg, 364–387. doi:10.1007/11804192_17

[7] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of ICS*. ACM, New York, NY, USA, 225–234. doi:10.1145/1375527.1375562

[8] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. 2015. The Design and Implementation of a Verification Technique for GPU Kernels. *Transactions on Programming Languages and Systems* 37, 3, Article 10 (2015), 49 pages. doi:10.1145/2743017

[9] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a Verifier for GPU Kernels. In *Proceedings of OOPSLA*. ACM, New York, NY, USA, 113–132. doi:10.1145/2384616.2384625

[10] Piotr Białas and Adam Strzelecki. 2016. Benchmarking the Cost of Thread Divergence in CUDA. In *Proceedings of PPAM*. Springer, 570–579. doi:10.1007/978-3-319-32149-3_53

[11] Guy Blelloch and John Greiner. 1995. Parallelism in sequential functional languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 226–237. doi:10.1145/224164.224210

[12] Guy E. Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming* (Philadelphia, Pennsylvania, USA) *(ICFP '96)*. Association for Computing Machinery, New York, NY, USA, 213–225. doi:10.1145/232627.232650

[13] Gregory Blike, Tiago Cogumbreiro, Julien Lange, Hannah Zicarelli, and Udaya Sathiyamoorthy. 2025. A Modular Static Cost Analysis for GPU Warp-Level Parallelism - Artifact POPL2026. (11 2025). doi:10.6084/m9.figshare.30689102.v1

[14] Stefan Blom, Marieke Huisman, and Matej Mihelčić. 2014. Specification and Verification of GPGPU Programs. *Science of Computer Programming* 95, P3 (2014), 376–388. doi:10.1016/j.scico.2014.03.013

[15] Michael Boyer, Kevin Skadron, and Westley Weimer. 2008. Automated dynamic analysis of CUDA programs. In *Proceedings of STMCS*, Vol. 33. https://www.cs.virginia.edu/~skadron/Papers/stmcs08.pdf

[16] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *Proceedings of POPL* (Paris, France). ACM, New York, NY, USA, 316–329. doi:10.1145/3009837.3009858

[17] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. 2011. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of SC*. ACM, New York, NY, USA, Article 13, 11 pages. doi:10.1145/2063384.2063401

[18] John Cheng, Max Grossman, and Ty McKercher. 2014. *Professional CUDA C Programming*. Wiley. https://books.google.com/books?id=q3DvBQAAQBAJ

[19] Tiago Cogumbreiro, Julien Lange, Dennis Liew, and Hannah Zicarelli. 2024. Memory access protocols: certified data-race freedom for GPU kernels. *Formal Methods in System Design* 63, 1 (2024), 134–171. doi:10.1007/s10703-023-00415-0

[20] Tiago Cogumbreiro, Julien Lange, Dennis Liew Zhen Rong, and Zicarelli. 2021. Checking Data-Race Freedom of GPU Kernels, Compositionally. In *Proceedings of CAV (LNCS, Vol. 12759)*. ACM, New York, NY, USA, 403–426. doi:10.1007/978-3-030-81685-8_19

[21] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 2013. Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels. In *Proceedings of ESOP*. Springer, Berlin, Heidelberg, 270–289. doi:10.1007/978-3-642-37036-6_16

[22] Shane Cook. 2013. Memory Handling with CUDA. In *CUDA Programming*, Shane Cook (Ed.). Morgan Kaufmann, 107–202. doi:10.1016/B978-0-12-415933-4.00006-5

[23] Lorenzo Dematté and Davide Prandi. 2010. GPU computing for systems biology. *Briefings in Bioinformatics* 11, 3 (03 2010), 323–333. doi:10.1093/bib/bbq006

[24] Benjamin Ferrell, Jun Duan, and Kevin W. Hamlen. 2019. CUDA au Coq: A Framework for Machine-validating GPU Assembly Programs. In *Proceedings of DATE*. IEEE, Piscataway, NJ, USA, 474–479. doi:10.23919/DATE.2019.8715160

[25] Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Proceedings of APLAS*. Springer, Cham, 275–295. doi:10.1007/978-3-319-12736-1_15

[26] Jürgen Giesl, Nils Lommen, Marcel Hark, and Fabian Meyer. 2022. *Improving Automatic Complexity Analysis of Integer Programs*. Springer, Cham, 193–228. doi:10.1007/978-3-031-08166-8_10

[27] Chunyang Gou and Georgi N. Gaydadjiev. 2012. Addressing GPU On-Chip Shared Memory Bank Conflicts Using Elastic Pipeline. *International Journal of Parallel Programming* 41 (2012), 400–429. doi:10.1007/s10766-012-0201-1

[28] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems* 34, 3, Article 14 (2012), 62 pages. doi:10.1145/2362389.2362393

[29] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL*. ACM, New York, NY, USA, 185–197. doi:10.1145/604131.604148

[30] Adrian Horga, Sudipta Chattopadhyay, Petru Eles, and Zebo Peng. 2020. Genetic algorithm based estimation of non-functional properties for GPGPU programs. *Journal of Systems Architecture* 103 (2020), 101697. doi:10.1016/j.sysarc.2019.101697

[31] Adrian Horga, Ahmed Rezine, Sudipta Chattopadhyay, Petru Eles, and Zebo Peng. 2022. Symbolic identification of shared memory based bank conflicts for GPUs. *Journal of Systems Architecture* 127 (2022), 102518. doi:10.1016/j.sysarc.2022.102518

[32] Takahiro Inoue, Hiroki Tokura, Koji Nakano, and Yasuaki Ito. 2020. Efficient Triangular Matrix Vector Multiplication on the GPU. In *Proceedings of PPAM'19*. Springer, Berlin, Heidelberg, 493–504. doi:10.1007/978-3-030-43229-4_42

[33] Artem Klashtorny, Zhuanhao Wu, Anirudh Mohan Kaushik, and Hiren Patel. 2023. Predictable GPU Wavefront Splitting for Safety-Critical Systems. *ACM Transactions on Embedded Computing Systems* 22, 5s, Article 107 (2023), 25 pages. doi:10.1145/3609102

[34] Kensuke Kojima and Atsushi Igarashi. 2013. A Hoare Logic for SIMT Programs. In *Proceedings of APLAS*, Vol. 8301. Springer, Berlin, Heidelberg, 58–73. doi:10.1007/978-3-319-03542-0_5

[35] Kensuke Kojima and Atsushi Igarashi. 2017. A Hoare Logic for GPU Kernels. *Transactions on Computational Logic* 18, 1, Article 3 (2017), 43 pages. doi:10.1145/3001834

[36] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO*. IEEE, Piscataway, NJ, USA, 75–88. doi:10.1109/CGO.2004.1281665

[37] Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of FSE*. ACM, New York, NY, USA, 187–196. doi:10.1145/1882291.1882320

[38] Guodong Li and Ganesh Gopalakrishnan. 2012. Parameterized Verification of GPU Kernel Programs. In *Proceedings of IPDPSW*. IEEE, Piscataway, NJ, USA, 2450–2459. doi:10.1109/IPDPSW.2012.302

[39] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic Verification and Test Generation for GPUs. In *Proceedings of PPoPP*, Vol. 47. ACM, New York, NY, USA, 215–224. doi:10.1145/2370036.2145844

[40] Jinhu Li and Jeffrey S. Racine. 2008. Maxima: An open source computer algebra system. *Journal of Applied Econometrics* 23, 4 (2008), 515–523. doi:10.1002/jae.1007

[41] Wei Li and Keshav Pingali. 1992. Access normalization: loop restructuring for NUMA compilers. In *Proceedings of ASPLOS*. ACM, New York, NY, USA, 285–295. doi:10.1145/143365.143541

[42] Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2022. Provable GPU Data-Races in Static Race Detection. In *Proceedings of PLACES (EPTCS, Vol. 356)*. 36–45. doi:10.4204/EPTCS.356.4

[43] Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2024. Sound and Partially-Complete Static Analysis of Data-Races in GPU Programs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2, Article 357 (2024), 28 pages. doi:10.1145/3689797

[44] Nils Lommen, Fabian Meyer, and Jürgen Giesl. 2022. Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops. In *Proceedings of IJCAR*. Springer, Berlin, Heidelberg, 734–754. doi:10.1007/978-3-031-10769-6_43

[45] Stefan K. Muller and Jan Hoffmann. 2021. Modeling and Analyzing Evaluation Cost of CUDA Kernels. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 25 (2021), 31 pages. doi:10.1145/3434306

[46] Stefan K. Muller and Jan Hoffmann. 2024. Modeling and Analyzing Evaluation Cost of CUDA Kernels. *ACM Transactions on Parallel Computing* 11, 1, Article 5 (2024), 53 pages. doi:10.1145/3639403

[47] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of PLDI*. ACM, New York, NY, USA, 496–512. doi:10.1145/3192366.3192394

[48] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. 2019. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review* 52, 1 (2019), 77–124. doi:10.1007/s10462-018-09679-z

[49] Weihao Qu, Marco Gaboardi, and Deepak Garg. 2019. Relational cost analysis for functional-imperative programs. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 92 (2019), 29 pages. doi:10.1145/3341696

[50] Weihao Qu, Marco Gaboardi, and Deepak Garg. 2021. Relational cost analysis in a functional-imperative setting. *Journal of Functional Programming* 31 (2021), e27. doi:10.1017/S0956796821000071

[51] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2017. Monadic refinements for relational cost analysis. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 36 (2017), 32 pages. doi:10.1145/3158124

[52] Julian Rosemann, Simon Moll, and Sebastian Hack. 2021. An Abstract Interpretation for SPMD Divergence on Reducible Control Flow Graphs. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 31 (2021), 31 pages. doi:10.1145/3434312

[53] Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. 2014. Divergence Analysis. *Transactions on Programming Languages and Systems* 35, 4, Article 13 (2014), 36 pages. doi:10.1145/2523815

[54] Lin Shi, Wen Liu, Heye Zhang, Yongming Xie, and Defeng Wang. 2012. A survey of GPU-based medical image computing techniques. *Quantitative imaging in medicine and surgery* 2 (2012), 188–206. Issue 3. doi:10.3978/j.issn.2223-4292.2012.08.02

[55] John E. Stone, David J. Hardy, Ivan S. Ufimtsev, and Klaus Schulten. 2010. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling* 29, 2 (2010), 116–125. doi:10.1016/j.jmgm.2010.06.010

[56] UMass Boston Software Verification Laboratory. 2024. Faial. https://gitlab.com/umb-svl/faial/. Accessed: 2025-11-24.

[57] UMass Boston Software Verification Laboratory. 2024. A Modular Static Cost Analysis for GPU Warp-Level Parallelism (Proofs). https://gitlab.com/umb-svl/faial-cost-rocq. Accessed: 2025-11-24.

[58] Pei Xue, Tao Li, Han Dong, Chunbo Liu, Wenjing Ma, and Songwen Pei. 2016. GB-RC4: Effective brute force attacks on RC4 algorithm using GPU. In *Proceedings of IGCC*. IEEE, Piscataway, NJ, USA, 1–6. doi:10.1109/IGCC.2016.7892594

[59] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-Fly Elimination of Dynamic Irregularities for GPU Computing. In *Proceedings of ASPLOS*. ACM, New York, NY, USA, 369–380. doi:10.1145/1950365.1950408

[60] Yao Zhang, Jonathan Cohen, and John D. Owens. 2010. Fast Tridiagonal Solvers on the GPU. In *Proceedings of PPoPP*. ACM, New York, NY, USA, 127–136. doi:10.1145/1693453.1693472