Equivalence Checking of ML GPU Kernels

BENJAMIN DRISCOLL[*✉], Stanford University, USA
KSHITIJ DUBEY[*], Microsoft Research, India
ANJIANG WEI, Stanford University, USA
NEERAJ KAYAL, Microsoft Research, India
RAHUL SHARMA[†], Google DeepMind, India
ALEX AIKEN, Stanford University, USA

With the rapid progress of deep learning and large language models (LLMs), companies spend enormous sums executing GPU kernels. These kernels have become prime targets for aggressive optimization. Recent efforts increasingly leverage LLMs to generate GPU kernels, but make no formal guarantees about the generated kernels. We present the first equivalence checker for GPU kernels and use it to formally verify the correctness of machine learning (ML) kernels optimized by hand, by LLM, and by compiler. We show that our equivalence checker is sound and, for a well-defined class of GPU kernels which includes many programs of interest, complete. Our implementation, Volta, can verify ML computations such as convolutions, matrix multiplications, and various attention mechanisms.

ACM Reference Format:
Benjamin Driscoll, Kshitij Dubey, Anjiang Wei, Neeraj Kayal, Rahul Sharma, and Alex Aiken. 2026. Equivalence Checking of ML GPU Kernels. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 338 (October 2026), 28 pages.

1 Introduction

Given two GPU kernels—a reference implementation and an optimized counterpart—our goal is to prove their extensional equivalence, i.e., that they produce identical outputs on all valid inputs, and, thereby, to verify the correctness of the optimized implementation. Equivalence checking has been extensively studied for over half a century [50] and recent works, such as Gupta et al. [28], have successfully proven the correctness of optimized CPU code. However, GPU programming introduces unique challenges arising from fine-grained synchronization among a massive number of threads and, to date, the literature proffers no equivalence checker for GPU programs.

In the past few years, companies have begun to spend enormous sums executing the GPU kernels for deep learning and large language models. These kernels have become prime targets for aggressive optimization, performed manually [11], fully automatically by agents [5, 58, 74], or even the old-fashioned way, by compilers [72]. Recent efforts increasingly leverage large language models to generate GPU kernels. For example, NVIDIA has used DeepSeek-R1,[1] Stanford researchers have used OpenAI o3 and Gemini 2.5 Pro,[2] Kevin-32B is a model trained with reinforcement learning for kernel generation,[3] Feature Search and Reinforcement combines search with reinforcement learning

[*]Both authors contributed equally to this research.
[†]This work was done while the author was at Microsoft Research.
[1]https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling
[2]https://crfm.stanford.edu/2025/05/28/fast-kernels.html
[3]https://cognition.ai/blog/kevin-32b

Authors' Contact Information: Benjamin Driscoll, Stanford University, Stanford, USA, bdrisc@cs.stanford.edu; Kshitij Dubey, Microsoft Research, Bengaluru, India, t-ksdubey@microsoft.com; Anjiang Wei, Stanford University, Stanford, USA, anjiang@cs.stanford.edu; Neeraj Kayal, Microsoft Research, Bengaluru, India, neeraka@microsoft.com; Rahul Sharma, Google DeepMind, Bengaluru, India, simplyrahul@google.com; Alex Aiken, Stanford University, Stanford, USA, aiken@cs.stanford.edu.

2026. ACM 2475-1421/2026/10-ART338
https://doi.org/

to derive kernels [14], Standard Kernel is building AI infrastructure for kernel generation,[4] and `mako` offers an interface for generating GPU kernels with different LLMs.[5] Both subject kernels to a range of empirical correctness checks, but neither currently provides formal correctness guarantees. Such guarantees offer greater confidence than empirical validation alone, which can be valuable before deployment in production setting where bugs can be costly.

GPU kernels execute thousands of threads in parallel on runtimes such as NVIDIA's CUDA [54] or AMD's ROCm [1], which can introduce subtle concurrency errors that testing alone cannot reliably detect. One major source of such errors is data races, which may appear only in extremely rare executions or may remain latent when kernels rely on implicit synchronization that happens only for a specific version of the test runtime or the test hardware but that is not guaranteed to hold in general. For example, a widely cited GPU programming tutorial [29] relies on warp-level synchronization that was removed after CUDA 9.0 [9]. Testing-based correctness checking is further complicated by floating-point arithmetic. GPU optimizations often reorder arithmetic operations, causing small numerical differences between the outputs of the reference and optimized implementations. Such variations are usually tolerated in tests, yet similar deviations can arise from genuine semantic bugs. For instance, an unsound optimization that omits a clipping operation may seem acceptable for some test inputs even though it reduces overall model accuracy. While compilers rarely introduce such mistakes, they are plausible when kernels are hand-optimized by human programmers or generated by agents.

State-of-the-art equivalence checking supports single-threaded integer programs [28] and high-level tensor operations [3, 77], but it is not obvious how to decompose the equivalence checking of GPU kernels into single-threaded sub-tasks. In an optimized kernel, there may be no one-to-one correspondence between its threads and the threads in the reference implementation. In addition, existing techniques have no support for synchronization among multiple threads.

Although equivalence checking is undecidable in general and GPUs allow programmers to write arbitrary code, efficient GPU kernels typically follow structural patterns that, incidentally, make formal reasoning feasible. We assume that the programs analyzed by our equivalence checker are *structured-CTAs*. Importantly, the number of threads, the sizes of tensors, the targets of pointers, and the values of any expressions that are branched on must be statically known. Although it is easy to construct programs that violate these assumptions and render our techniques inapplicable, we and others [38, 45, 66] observe that these assumptions very often hold in practice (Section 3.2). For example, JAX, a popular ML framework, requires tensor dimensions to be fixed[6]; the underlying XLA compiler requires tensor sizes and data flow graphs to be known a priori to perform aggressive optimizations. We discuss structured-CTAs further in Section 3.2 and formally define them in Section 4.1. Our implementation, Volta,[7] checks that its input is a structured-CTA and raises an exception if it is not. Finally, we note that, while our work focuses on kernels targeting NVIDIA GPUs due to their widespread adoption, the techniques are also applicable to other GPU runtimes.

The core of our equivalence checker is symbolic evaluation. Consider a tensor computation—a workload that dominates machine learning. We symbolically execute the reference computation under an arbitrary, fixed *schedule* to derive an expression over $\mathbb{R}$, the real numbers, that represents the output tensor as a function of the input tensors, discovering any races or deadlocks as we go. Our main technical result, *confluence*, which we formally verify in the Agda [53] proof assistant, guarantees that all symbolic evaluator schedules lead to the same expression or report a race or deadlock. The same procedure is applied to the optimized computation. Finally, we apply a

[4] https://standardkernel.com
[5] https://generate.mako.dev
[6] https://docs.jax.dev/en/latest/notebooks/Common_Gotchas_in_JAX.html#dynamic-shapes
[7] Verification Of Large Thread Arrays

procedure to decide equality of the two symbolic expressions and, if they are equal, we conclude that the optimized kernel is semantically equivalent to the reference kernel, for all schedules.

We are not aware of a proof in the literature that the symbolic identities we encounter are decidable, though there is work on related problems, e.g., [42, 48]. In particular, in Section 4.4, we prove decidability for equalities of the form $\sum_{i=1}^{m} p_i(\bar{x})e^{h_i(\bar{x})} = 0$, given multivariate polynomials $p_1, h_1, \ldots \in \mathbb{R}[\bar{x}]$. It is worth noting that optimized kernels are almost never bit-wise equivalent to the reference implementation in floating point arithmetic, since most optimizations reorder arithmetic operations when distributing work among GPU threads.

Although there is no prior equivalence checker for GPU programs, there are prior verification frameworks that check weaker properties such as data race freedom [45]. Our equivalence checker, Volta, soundly verifies data race and deadlock freedom of kernels utilizing barrier synchronization (Theorem 1) in time proportional to the number of instructions. It satisfies a *true positives property*, which guarantees that every reported data race or deadlock corresponds to a concrete schedule that triggers erroneous behavior (Theorem 2). Barrier synchronization operations include `mma.sync` and `syncwarp`, which programmers commonly employ when accelerating machine learning workloads with tensor cores [55]. We compare with data race checking tools in Section 6.4 and Section 7.

Volta checks the identities output by its symbolic executor in less than 3 minutes on all evaluated benchmarks, including convolution, matrix multiplication, and attention (Section 6). Volta operates as a black-box equivalence checker, requiring no information about how an optimized kernel was produced; it verifies correctness solely from the assembly corresponding to the optimized program itself. In particular, Volta analyzes PTX assembly, the lowest documented level of the stack for NVIDIA GPUs [57]. Operating at the PTX level enables our approach to handle various developer front-ends. For example, GPU programmers can write CUDA kernels or CUTLASS templates [19]. NVIDIA's `nvcc` compiler translates both into PTX.

In sum, we make the following contributions:

- A linear time algorithm for detecting races and deadlocks in a class of GPU kernels (structured CTAs) utilizing barrier synchronization.
- A proof of decidability for equalities of the form $\sum_{i=1}^{m} p_i(\bar{x})e^{h_i(\bar{x})} = 0$, given multivariate polynomials $p_1, h_1, \ldots \in \mathbb{R}[\bar{x}]$.
- The first equivalence checker for a practical class of GPU kernels (again, structured CTAs), Volta, that can verify the correctness of optimized convolutions, matrix multiplications, reductions, and attention mechanisms. We prove that Volta guarantees data race freedom and satisfies a true positives property, and we formalize the key, constituent lemma in Agda.

The rest of the paper is as follows. Section 2 shows, at a high-level, how Volta verifies a motivating example. Section 3 provides background on GPU architecture, synchronization, and ML GPU kernels. Section 4 describes our symbolic execution algorithm formally. Section 4.3 establishes our main technical results. Section 5 describes Volta's implementation and Section 6 evaluates it across a diverse set of GPU kernels, demonstrating its ability to verify realistic programs and detect bugs. Section 7 reviews related work and Section 8 concludes with directions for future research.

2 Motivating Example

As a motivating example, consider the *softmax* computation, which appears in the attention mechanism of large language models (LLMs). Attention is the primary performance bottleneck during LLM inference with long contexts [21]. Listing 1a presents a naive reference implementation of softmax in CUDA. In this example, we use $N = 4$, meaning that four GPU threads operate on a

```
__global__ void
softmax (float *x, float *y, int N) {
    extern __shared__ float buf[];

    int tid = threadIdx.x;
    buf[tid] = expf(x[tid]);
    __syncthreads();

    float sum = 0.0f;
    for (int i = 0; i < N; i++)
        sum += buf[i];
    y[tid] = buf[tid] / sum;
    return;
}
```

(a) Naive Softmax

```
__global__ void
online_softmax (float *x, float *y, int N) {
    int tid = threadIdx.x;

    float m = -INFINITY;  // running max
    float d = 0.0f;       // running denom

    for (int i = 0; i < N; i++) {
        float v = x[i];
        float new_m = fmaxf(m, v);
        d = d * expf(m - new_m) + expf(v - new_m);
        m = new_m;
    }
    y[tid] = expf(x[tid] - m) / d;
    return;
}
```

(b) Online Softmax (FlashAttention)

Fig. 1. Naive softmax (left) requires materializing all exponentiations in shared memory; online softmax (right, used in FlashAttention) maintains a running max and normalization factor, enabling numerically stable and memory-efficient streaming.

vector $[x_1, x_2, x_3, x_4]$ to compute $[y_1, y_2, y_3, y_4]$, where

$$y_i = e^{x_i} / \left(\sum_{j=1}^{4} e^{x_j} \right) \tag{1}$$

The parameter N is typically known at compile time. Kernels are often specialized for particular values of N, since a single CUDA kernel rarely achieves optimal performance across all possible configurations.

Listing 1a uses an array, `buf`, stored in *shared memory*, a software-managed cache accessible to all threads within a thread block. Each thread has a unique identifier, `tid`, and thread i stores e^{x_i} in `buf`. The `syncthreads` is a barrier primitive that synchronizes all the threads ensuring that all shared memory writes are visible before any thread proceeds. After synchronization, each thread i sums the elements of `buf` into the variable `sum` and writes $\frac{e^{x_i}}{\texttt{sum}}$ to y_i.

Listing 1b gives a softmax implementation used in FlashAttention [21], a major advance in the implementation of attention mechanisms. It employs an online softmax computation that is both numerically stable and avoids the use of shared memory. This formulation enables tiling optimizations that are key to FlashAttention's performance. The two softmax implementations shown in Figure 1 are not bit-wise identical due to floating point rounding differences, yet both are intended to compute the mathematical expression in Equation 1. Volta is designed to verify that they both, in fact, compute this expression.

Conceptually, our analysis expands the `softmax` kernel into a list of per-thread programs, where the i^{th} program represents the code executed by thread i.

Figure 2 illustrates the expanded program. Observe that the code executed by each thread is straight-line, containing no branches or loops. Moreover, the addresses corresponding to all memory accesses are statically known.

Note that this representation is shown only for clarity of exposition. In fact, the actual implementation (Section 5) never constructs the fully expanded program, nor does it perform other AST transformations. It operates over the original program, requiring that values are sufficiently static such that the symbolic executor can resolve all control flow.

tid=0	**tid=1**	**tid=2**	**tid=3**
buf[0] = exp(x[0]);	buf[1] = exp(x[1]);	buf[2] = exp(x[2]);	buf[3] = exp(x[3]);
__syncthreads();	__syncthreads();	__syncthreads();	__syncthreads();
sum = 0;	sum = 0;	sum = 0;	sum = 0;
sum += buf[0];	sum += buf[0];	sum += buf[0];	sum += buf[0];
sum += buf[1];	sum += buf[1];	sum += buf[1];	sum += buf[1];
sum += buf[2];	sum += buf[2];	sum += buf[2];	sum += buf[2];
sum += buf[3];	sum += buf[3];	sum += buf[3];	sum += buf[3];
y[0] = buf[0]/sum;	y[1] = buf[1]/sum;	y[2] = buf[2]/sum;	y[3] = buf[3]/sum;

Fig. 2. Naive softmax execution for N=4 threads. Each column shows the code executed by a single thread indexed by **tid**.

Given the program, Volta performs simple round-robin scheduling: it iterates over each column until a thread encounters a `syncthreads` barrier and blocks. For each thread, all shared memory locations that are read or written are recorded. When a thread is blocked, Volta proceeds to the next active thread. If the recorded accesses reveal any data races, Volta generates an error report. Once all threads reach the corresponding barrier, the recorded information is cleared, and all threads become active. If all threads become blocked, there is a deadlock and Volta also generates an error report. In Figure 2, if the `syncthreads` barrier is missing, Volta reports data races. For example, when analyzing the thread with **tid** 0, the analysis records a read from `buf[1]`. Subsequently, when analyzing the thread with **tid** 1, it reports a data race between the read of `buf[1]` in thread 0 and the write to `buf[1]` in thread 1.

Given Listing 1a, the executor produces the following, symbolic output state:[8]

$$y_{\text{naive}}[0] = \frac{e^{x_0}}{D},\ y_{\text{naive}}[1] = \frac{e^{x_1}}{D},\ y_{\text{naive}}[2] = \frac{e^{x_2}}{D},\ y_{\text{naive}}[3] = \frac{e^{x_3}}{D},\ D = ((e^{x_0} + e^{x_1}) + e^{x_2}) + e^{x_3}$$

Confluence (Lemma 1) guarantees that, since we did not detect a data race, if we had used some other schedule different from the round-robin scheduling of Volta, we would still end up with an identical output state. Likewise, since we did not encounter a deadlock, no schedule will deadlock. Repeating the same process for the online softmax in Listing 1b produces the following output:

$$y_{\text{opt}}[0] = \frac{e^{x_0 - M_3}}{D},\ y_{\text{opt}}[1] = \frac{e^{x_1 - M_3}}{D},\ y_{\text{opt}}[2] = \frac{e^{x_2 - M_3}}{D},\ y_{\text{opt}}[3] = \frac{e^{x_3 - M_3}}{D}$$

where

$$M_1 = \max(x_0, x_1),\ M_2 = \max(M_1, x_2),\ M_3 = \max(M_2, x_3)$$

and

$$D = \left(\left(e^{x_0 - M_1} + e^{x_1 - M_1}\right) e^{M_1 - M_2} + e^{x_2 - M_2}\right) e^{M_2 - M_3} + e^{x_3 - M_3}$$

Finally, we create the following decision problem,

$$\forall i \in \{0, 1, 2, 3\}.\ y_{\text{naive}}[i] = y_{\text{opt}}[i]$$

Our decision procedure (Section 4.4) successfully validates these identities, allowing Volta to conclude that the two programs in Figure 1 are equivalent. As an additional contribution, we demonstrate the decidability of such identities (Section 4.4).

[8]We add the subscript "naive" in y_{naive} to disambiguate it from the output of the optimized computation

3 Background

We begin with necessary background on GPUs. To execute a task on a GPU, the CPU host launches a grid of cooperative thread arrays (CTAs), also known as *thread-blocks*. All CTAs share access to *global memory* (HBM), typically sized in gigabytes. Different CTAs in a grid usually write to disjoint regions of global memory due to the cost of inter-CTA synchronization, which was not even possible for many GPU generations. The GPU hardware schedules each CTA on a *streaming multiprocessor* (SM); a modern GPU may contain hundreds of SMs. The resources assigned to each SM are limited and hence each CTA is tasked to compute only a *tile* of the output tensor. Each CTA consists of 128 to 1024 *threads*, depending on the hardware generation. Within a CTA, threads are grouped into *warps*, each consisting of 32 threads. Every thread maintains its own *registers* for local storage, while *shared memory* serves as a software-managed cache accessible to all threads in the same CTA. To coordinate access to shared memory, threads must use explicit *synchronization* operations. Missing synchronization leads to data races and over synchronization leads to loss of performance. The shared memory available per CTA ranges from 48 KB to 256 KB, depending on the GPU architecture. In a typical CTA, threads first load data from global memory into shared memory, perform arithmetic operations while keeping intermediate values in registers or shared memory, and finally write the results back to global memory. Writing code for efficient CTAs is challenging because programmers must reason about massive thread-level parallelism and low-level hardware details. Volta aims to statically identify incorrect or unsafe optimizations. We next describe the synchronization mechanisms available on GPUs.

3.1 Synchronization operations

NVIDIA GPUs support independent thread scheduling, meaning that in the absence of explicit synchronization, each thread executes independently. The fundamental synchronization primitive is `syncthreads`, a barrier that spans all threads within a CTA. Threads that reach the barrier earlier are suspended until all threads in the CTA arrive or exit, at which point execution resumes for all of the threads blocked at the barrier.

Another synchronization primitive is `syncwarp`, which operates on a user-specified subset of threads within a single warp. This mechanism is commonly used when programming tensor cores—CUDA intrinsics such as `wmma::mma_sync` synchronize all threads in a warp to perform matrix multiplication. Most prior verification techniques for GPU kernels support only `syncthreads` and therefore cannot reason about machine learning kernels that rely on warp-level synchronization. While `syncthreads` is deadlock free (per the CUDA documentation, it can only be placed under conditionals that evaluate uniformly across the entire CTA), `syncwarp` can introduce deadlocks.

In summary, the synchronization mechanisms we consider are barriers that block subsets of the available threads. Asynchronous producer-consumer-style primitives such as `arrive` are beyond the scope of this paper. The race checking mechanism of Volta could potentially be swapped out for a more general, if less efficient, mechanism such as a happens-before analysis in the style of Sharma et al. [66], without disrupting the rest of the system, but we leave this for future work.

3.2 Why Structured-CTAs?

In this section, we discuss, at a high level, the assumptions that Volta makes about input kernels. We first state our assumptions, then motivate them. Note that these assumptions are checked: Volta raises an exception if a kernel does not satisfy our requirements (Section 5).

To be checked for equivalence, a program must operate on a known number of CTAs and threads per CTA. Each CTA must operate over tensor tiles whose sizes are statically known. In each thread of each CTA, given the `tid` (thread ID), all branch targets and addresses used in memory accesses

must be statically known. We compare kernels CTA-to-CTA, so we do not support equivalence checking of optimizations that mix responsibilities between CTAs. In Section 4.1, we define a language of *structured-CTAs*, formally encoding these assumptions, and, in Section 4.3, prove the completeness of Volta for such programs.

Structured-CTAs reflect well-established best practices for achieving high GPU performance, as supported by empirical study. A large-scale analysis of GPGPU kernels in the wild found that most memory accesses are *control independent* and *data independent* [45] (59.5% of 2770 programs). Similarly, a study of the CUDA SDK showed that most programs exhibit *access invariance* [38], meaning that threads perform the same memory accesses for all inputs (52 out of 76). Weft [66], a race detector for scientific and cudaDMA kernels, makes all of the above assumptions—*all* 26 scientific computations studied by Weft fall into the aforementioned class.

Moreover, we are focused on the kernels that are performance bottlenecks in the training or inference of machine learning (ML) models, such as matrix multiplications, convolutions, and multi-head attention. In these workloads, the structure of the ML model is fixed: tensor entries depend on runtime inputs, but tensor sizes are statically known. Kernels are therefore specialized for particular tensor dimensions to achieve maximum performance. In fact, JAX [12], a popular ML framework, only permits tensors with fixed dimensions.

Once tensor shapes are known, loop bounds can also be determined, since the bounds in tensor operations depend only on tensor shapes. Volta can statically evaluate these loops to "flat" expressions that do not contain conditionals, effectively unrolling them. We also require that the targets of conditional branches can be statically resolved; note this restriction exploits the fact that efficient machine learning kernels do not make truly data dependent runtime decisions. Under these assumptions, the code executed by each individual thread can be represented as *straight-line*, no dynamic branches or loops. It is common for threads to branch on predicates of their thread index or of a loop index (counting up to some tensor dimension); however, such control flow still preserves the straight-line property for each thread's execution. For example, causal masking in FlashAttention [21] branches on whether an array index is above or below the diagonal, but this is a static condition given the tensor shape.

Efficient GPU kernels carefully orchestrate memory transfers by coalescing accesses to global memory and avoiding shared memory bank conflicts. As a result, GPU programmers typically define memory access patterns statically, making memory addresses independent of the tensor contents. Consequently, when a straight-line thread accesses an array element, both the base array and the exact index can be determined statically. This property allows the symbolic evaluator to record memory accesses precisely and to detect data races accurately.

Typically, ML kernels launch a grid of CTAs where the code of each CTA is identical and differs only in the instantiation of free variables `blockIdx.x`, `blockIdx.y`, and `blockIdx.z`. Each CTA is responsible for computing a fixed-size *tile* of the output tensor. Many GPU optimization focus on this CTA-level code. For instance, the CTA code can be modified to use tensor cores. When checking a reference kernel against an optimized kernel, we focus on proving equivalence between the code of a CTA from the reference kernel and the code of the corresponding CTA in the optimized kernel. Hence, correctness checking of optimizations that change the number of CTAs launched in a grid is beyond the scope of this work. Since each pair of CTAs can be checked independent of each other, following prior work [66], our evaluation simply checks the equivalence of the CTAs with `blockIdx.x` = `blockIdx.y` = `blockIdx.z` = 0 in the reference and the optimized kernels.

Which practical kernels fall outside the class of structured-CTAs? Notably, top-k/argsort kernels do, as commonly used for expert selection in mixture-of-experts models, beam search, and ranking. Their control flow is data dependent—comparison results determine swap operations—and implementations often use atomic operations for parallel reductions across CTAs.

$$
\begin{array}{lrcl}
\text{Statement} & s & ::= & r \coloneqq c \mid r \coloneqq r_1 \oplus r_2 \mid r \coloneqq r' \mid r \coloneqq *g \mid *g \coloneqq r \mid \textbf{sync}\, I \\
\text{Thread Program} & T & ::= & \textbf{return} \mid s \fatsemi T \\
\text{Program} & P & ::= & \diamond \mid P \parallel T
\end{array}
$$

Fig. 3. Syntax of $\mathcal{L}$, the language of structured-CTAs.

Finally, a sound equivalence checker operating over floating-point arithmetic would likely reject many aggressively optimized kernels; IEEE-754 is not even associative. In practice, both developers and domain-specific compilers treat tensor values as real numbers when applying optimizations. Such transformations, similar in spirit to the `-ffast-math` option in `gcc`, are unsound for floating point arithmetic. Volta follows the same convention as existing optimizers and models tensor elements as reals.

4 Algorithm

Our equivalence checker proceeds in two phases. The first phase symbolically executes a pair of programs, determining whether they are data race free and, if so, expresses their output values as terms of symbolic input tensors. If both programs are data race free, we proceed to the second phase, which verifies that, for each output element, the symbolic terms derived for both programs are equal. For example, when analyzing Listing 1a, the symbolic evaluator starts with environment $\{x[0] \coloneqq \alpha, x[1] \coloneqq \beta, x[2] \coloneqq \gamma, x[3] \coloneqq \delta\}$ for fresh symbols $\alpha, \beta, \gamma, \delta$. It determines there is no data race and outputs a symbolic state with $y[0] \coloneqq e^{\alpha}/(e^{\alpha} + e^{\beta} + e^{\gamma} + e^{\delta})$.

4.1 Formal Setting

As a setting to formalize our analysis, we develop a simple language $\mathcal{L}$ of structured-CTAs capturing the relevant aspects of PTX. The syntax of $\mathcal{L}$ programs is given in Figure 3 in terms of registers r, constants c, shared memory addresses g, and sets of thread IDs (TIDs) I. We give $\mathcal{L}$ a dynamics via a pair of small-step relations, $(\mathcal{G}, \mathcal{R}, P) \to (\mathcal{G}, \mathcal{R}, P)$ and $(\mathcal{G}, R, T) \to (\mathcal{G}, R, T)$:

$$\textsc{Schd}\quad \frac{(\mathcal{G}, \mathcal{R}(i), P(i)) \to (\mathcal{G}', R', T')}{(\mathcal{G}, \mathcal{R}, P) \to (\mathcal{G}', \mathcal{R}[i \mapsto R'], P[i \mapsto T'])}$$

$$\textsc{Sync}\quad \frac{\begin{array}{c}\forall i \notin I.\ P'(i) = P(i) \\ \forall i \in I.\ (P(i) = \textbf{return} \wedge P'(i) = \textbf{return}) \vee (\exists T_i.\ P(i) = \textbf{sync}\, I \fatsemi T_i \wedge P'(i) = T_i)\end{array}}{(\mathcal{G}, \mathcal{R}, P) \to (\mathcal{G}, \mathcal{R}, P')}$$

$$\textsc{Const}\quad \frac{}{(\mathcal{G}, R, r \coloneqq c \fatsemi T) \to (\mathcal{G}, R[r \mapsto c], T)} \qquad \textsc{BinOp}\quad \frac{}{(\mathcal{G}, R, r \coloneqq r_1 \oplus r_2 \fatsemi T) \to (\mathcal{G}, R[r \mapsto R(r_1) \oplus R(r_2)], T)}$$

$$\textsc{RdReg}\quad \frac{}{(\mathcal{G}, R, r \coloneqq r' \fatsemi T) \to (\mathcal{G}, R[r \mapsto R(r')], T)} \qquad \textsc{RdMem}\quad \frac{}{(\mathcal{G}, R, r \coloneqq *g \fatsemi T) \to (\mathcal{G}, R[r \mapsto \mathcal{G}(g)], T)}$$

$$\textsc{WrMem}\quad \frac{}{(\mathcal{G}, R, *g \coloneqq r \fatsemi T) \to (\mathcal{G}[g \mapsto R(r)], R, T)}$$

The top-level small-step relation is indexed by configurations $(\mathcal{G}, \mathcal{R}, P)$ where $\mathcal{G}$ models shared memory and is a map from addresses to values, $\mathcal{R}$ models the threads' register files and is a map from TIDs to registers to values, and P is the current program. We will consider both concrete program executions, where values are drawn from $\mathbb{R}$, and abstract program executions, where values are symbolic.

Schd allows any thread i (which is not blocked at a **sync**) to be advanced, modeling the non-deterministic execution of the machine. We use $f[x \mapsto y]$ to denote the function which takes value y at x and is f elsewhere. The second top-level rule, Sync, advances the threads blocked at a **sync** I, but only if every thread in I is at a **sync** I or **return**. When $I = \{1, \ldots, N\}$ for N the size of a thread block, **sync** models the PTX instruction `syncthreads`. When $I \subseteq \{k, \ldots, k+31\}$ with $k \bmod 32 = 1$, **sync** models the warp-level barriers, e.g., `syncwarp` with a mask or tensor core `mma.sync`. Permitting threads in I to be at a **return** captures the flexibility of PTX, which allows threads to terminate early. The program is stuck if a thread i_1 synchronizes on I_1 and another i_2 on I_2, where $I_1 \neq I_2$ and $i_1, i_2 \in I_1 \cap I_2$. Although some GPU architectures may permit specific combinations of I_1 and I_2 to proceed without deadlock, we conservatively treat all such cases as stuck. Finally, note that **sync** operations are not named: as with `syncthreads` in CUDA, whichever **sync** instances happen to align in the dynamics are matched with one another. (In fact, because control flow in a structured-CTA cannot depend on runtime values, the same instances align in every execution.)

The other small-step relation given above is a thread-level relation indexed by configurations $(\mathcal{G}, R, T)$. $\mathcal{G}$ is, again, a map from addresses to values, modeling shared memory, R is the current thread's register file, and T is the current thread's program. Const, BinOp, RdReg, RdMem, and WrMem perform the obvious updates to $\mathcal{G}$ and R. $\oplus$ models the various arithmetic operations handled by the real implementation, Volta. Note that there is no thread-level small-step rule for a thread program of the form **sync** $I \fatsemi T'$—one must use the top-level Sync rule.

Note that, for confluence (Lemma 1), it will be essential that (dynamic) control flow cannot affect whether a **sync** executes or the set of addresses read and written, which is why thread programs are straight-line sequences of statements. That said, the formalism is more permissive than it may first appear: the operators $\oplus$ are black boxes, assumed only to be pure functions of their register operands, and can therefore "hide" control flow. For example, $\oplus$ could be max implemented via a branch.

4.2 Symbolic Execution

A *program trace* is a derivation of the reflexive, transitive closure of $(\mathcal{G}, \mathcal{R}, P) \to (\mathcal{G}, \mathcal{R}, P)$, which we denote $(\mathcal{G}, \mathcal{R}, P) \to^* (\mathcal{G}, \mathcal{R}, P)$. A *data race* occurs between two memory accesses a_i and a_j when at least one is a write (a Schd of a WrMem) and there exist two traces, one in which a_i is directly followed by a_j and another in which a_j is directly followed by a_i. This is the definition used by, e.g., Brookes [13].[9] Since data races are typically bugs and it is not clear, in general, how equivalence should be defined for programs that race, we extend the dynamics of $\mathcal{L}$ to return error value $\bot$ if the program has a data race and use this extended dynamics when performing symbolic execution.

Concretely, we maintain an extra context $\mathcal{X}$ tracking: (1) for each address g and each thread i, all threads j that have not performed a **sync** with i since it last read g; (2) for each address g, the last writer i and all threads j that have not performed a **sync** with i since it wrote g. That is,

$$\mathcal{X} : \text{MemEvs} \triangleq \text{Addr} \to (\text{TID} \to \mathscr{P}(\text{TID})) \times (\text{TID} \times \mathscr{P}(\text{TID}))$$

[9]We could define a data race without requiring that a_i and a_j follow each other directly. There exist traces in which they occur in different orders and follow each other directly iff there exist traces in which they occur in different orders.

Races arise because a thread reads an address before synchronizing with its last writer, or writes an address before synchronizing with its previous readers and its last writer. We define a predicate *noRacingRd* taking a thread i and the read data of $\mathcal{X}(g)$ and determining whether i has synchronized with all other readers of g as required. Likewise, we define a predicate *noRacingWr* taking a thread i and the write data of $\mathcal{X}(g)$ and determining whether i is or has synchronized with the last writer of g as required.

$$
\begin{aligned}
&\mathit{noRacingRd} : \mathrm{TID} \to (\mathrm{TID} \to \mathscr{P}(\mathrm{TID})) \to \mathrm{Prop} \\
&\mathit{noRacingRd}(i, \mathrm{rd}) \triangleq \forall j.\ i = j \vee i \notin \mathrm{rd}(j) \\
&\mathit{noRacingWr} : \mathrm{TID} \to \mathrm{TID} \times \mathscr{P}(\mathrm{TID}) \to \mathrm{Prop} \\
&\mathit{noRacingWr}(i, (j, J)) \triangleq i = j \vee i \notin J
\end{aligned}
$$

We can now refine our thread-level small-step relation. The new relation $(\mathcal{X}, \mathcal{G}, R, T) \hookrightarrow_i (\mathcal{X}', \mathcal{G}', R, T)$ tracks the TID of the thread i being executed, and includes an $\mathcal{X}$ in its configurations. The non-trivial changes are to RdMem and WrMem. Letting $\mathbb{I}$ be the set of all TIDs:

$$
\textsc{RdMem'}\ \frac{(\mathrm{rd}, \mathrm{wr}) = \mathcal{X}(g) \qquad \mathit{noRacingWr}(i, \mathrm{wr}) \qquad x' = (\mathrm{rd}[i \mapsto \mathbb{I}], \mathrm{wr})}{(\mathcal{X}, \mathcal{G}, R, r \coloneqq *g \fatsemi T) \hookrightarrow_i (\mathcal{X}[g \mapsto x'], \mathcal{G}, R[r \mapsto \mathcal{G}(g)], T)}
$$

$$
\textsc{WrMem'}\ \frac{(\mathrm{rd}, \mathrm{wr}) = \mathcal{X}(g) \qquad \mathit{noRacingRd}(i, \mathrm{rd}) \qquad \mathit{noRacingWr}(i, \mathrm{wr}) \qquad x' = (\mathrm{rd}, (i, \mathbb{I}))}{(\mathcal{X}, \mathcal{G}, R, *g \coloneqq r \fatsemi T) \hookrightarrow_i (\mathcal{X}[g \mapsto x'], \mathcal{G}[g \mapsto R(r)], R, T)}
$$

Next, we must adjust Sync so that $\mathcal{X}$ is updated appropriately:

$$
\textsc{Sync'}\ \frac{\begin{array}{c}\forall i \notin I.\ P'(i) = P(i) \\ \forall i \in I.\ (P(i) = \textbf{return} \wedge P'(i) = \textbf{return}) \vee (\exists T_i.\ P(i) = \textbf{sync}\, I \fatsemi T_i \wedge P'(i) = T_i)\end{array}}{(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow (\mathit{syncMem}(I, \mathcal{X}), \mathcal{G}, \mathcal{R}, P')}
$$

$$
\begin{aligned}
\mathit{syncMemRd}(I, \mathrm{rd})(i) &\triangleq (\mathrm{rd}(i) \setminus I) \text{ if } i \in I \text{ else } \mathrm{rd}(i) \\
\mathit{syncMemWr}(I, (j, J)) &\triangleq (j, J \setminus I) \text{ if } j \in I \text{ else } (j, J) \\
\mathit{syncMem}(I, \mathcal{X})(g) &\triangleq (\mathit{syncMemRd}(I, \pi_1(\mathcal{X}(g))), \mathit{syncMemWr}(I, \pi_2(\mathcal{X}(g))))
\end{aligned}
$$

Finally, we would like symbolic execution to error when a race is detected rather than getting stuck:

$$
\textsc{RdMemBad}\ \frac{(\mathrm{rd}, \mathrm{wr}) = \mathcal{X}(g) \qquad \neg \mathit{noRacingWr}(i, \mathrm{wr})}{(\mathcal{X}, \mathcal{G}, R, r \coloneqq *g \fatsemi T) \hookrightarrow_i \bot}
$$

$$
\textsc{WrMemBad}\ \frac{(\mathrm{rd}, \mathrm{wr}) = \mathcal{X}(g) \qquad \neg \mathit{noRacingRd}(i, \mathrm{rd}) \vee \neg \mathit{noRacingWr}(i, \mathrm{wr})}{(\mathcal{X}, \mathcal{G}, R, *g \coloneqq r \fatsemi T) \hookrightarrow_i \bot} \qquad \textsc{SchdBad}\ \frac{(\mathcal{X}, \mathcal{G}, \mathcal{R}(i), P(i)) \hookrightarrow_i \bot}{(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow \bot}
$$

We write the reflexive, transitive closure of $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow (\mathcal{X}', \mathcal{G}', \mathcal{R}', P')$ as $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* (\mathcal{X}', \mathcal{G}', \mathcal{R}', P')$.

Note that the formalism assumes all addresses are valid, though validity could be tracked by enriching the codomain of $\mathcal{G}$ and R and adding a check in RdMem' and WrMem'.

4.3 Guarantees

We prove the soundness (Theorem 5) and completeness (Theorem 6) of our symbolic evaluation based equivalence checking. For soundness, we prove that if execution of two programs under the *checked* dynamics $\cdot \hookrightarrow \cdot$ with symbolic inputs produces equal terms, then those programs evaluate to the same expression under the usual, *unchecked* dynamics $\cdot \to \cdot$ for all inputs in $\mathbb{R}$. The central challenge lies in reasoning about the apparent non-determinism introduced by Schd'. Since symbolic execution selects an arbitrary unblocked thread of the program for execution at each step, we must show that, for all programs, all possible schedules either produce the same output $\mathcal{G}, \mathcal{R}$, raise an error $\bot$ (indicating a data race), or become stuck (indicating a deadlock). Our key lemma is the following, which we verify in the Agda [53] proof assistant:

Lemma 1 (Confluence). *If* $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* (\mathcal{X}_1, \mathcal{G}_1, \mathcal{R}_1, P_1)$ *and* $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* (\mathcal{X}_2, \mathcal{G}_2, \mathcal{R}_2, P_2)$ *then there exists* $\mathcal{X}', \mathcal{G}', \mathcal{R}', P'$ *such that* $(\mathcal{X}_1, \mathcal{G}_1, \mathcal{R}_1, P_1) \hookrightarrow^* (\mathcal{X}', \mathcal{G}', \mathcal{R}', P')$ *and* $(\mathcal{X}_2, \mathcal{G}_2, \mathcal{R}_2, P_2) \hookrightarrow^* (\mathcal{X}', \mathcal{G}', \mathcal{R}', P')$.

Proof. In our Agda development, we apply the standard technique for obtaining confluence results. Namely, we show a one-step confluence/diamond property and use it to tile the desired, many-step diamond. While $\cdot \hookrightarrow \cdot$ does not satisfy a one-step diamond property (consider the case where one of the initial steps is SchdBad), its reflexive closure does, which suffices. The proof has a large number of cases, but most boil down to commutativity lemmas such as: Sync' for I followed by Sync' for J produces the same configuration as Sync' for J followed by Sync' for I, assuming all necessary hypotheses for these Sync' steps hold. □

We will use confluence to prove soundness and completeness of equivalence checking. But first, we take a brief detour to prove data race freedom and the true positives property for $\cdot \hookrightarrow \cdot$, our checked dynamics.

We will need some notation. Let $(\mathcal{G}, \mathcal{R}, P) \downarrow (\mathcal{G}', \mathcal{R}')$ denote that $(\mathcal{G}, \mathcal{R}, P) \to^* (\mathcal{G}', \mathcal{R}', \textbf{return} \parallel \cdots \parallel \textbf{return})$, $(\mathcal{G}, \mathcal{R}, \mathcal{X}, P) \int (\mathcal{G}', \mathcal{R}')$ denote that $(\mathcal{G}, \mathcal{R}, \mathcal{X}, P) \hookrightarrow^* (\mathcal{X}', \mathcal{G}', \mathcal{R}', \textbf{return} \parallel \cdots \parallel \textbf{return})$ for some $\mathcal{X}'$, and $(\mathcal{G}, \mathcal{R}, \mathcal{X}, P) \int \bot$ denote that $(\mathcal{G}, \mathcal{R}, \mathcal{X}, P) \hookrightarrow^* \bot$. Let $\mathcal{X}_\varnothing$ denote the initial context of memory events $\lambda \_. ((\lambda \_. \varnothing), (0, \varnothing))$ (the choice of 0 here is arbitrary), and let $(\mathbb{V}, \oplus_{\mathbb{V}})$ denote the collection (free algebra) of symbolic terms. We will sometimes drop the subscript on $\oplus_{\mathbb{V}}$ where it is obvious. Finally, we define a *map* $\varphi : (A, \oplus_A) \to (B, \oplus_B)$ to be a function $\varphi : A \to B$ between the collections of program values A and B such that $\varphi(v_1 \oplus_A v_2) = \varphi(v_1) \oplus_B \varphi(v_2)$.

Now we are ready to proceed with our proofs. For every unchecked trace, there is a checked trace that reaches the same result, or detects a data race:

Lemma 2. *If* $(\mathcal{G}, \mathcal{R}, P) \to^* (\mathcal{G}', \mathcal{R}', P')$ *then, for all* $\mathcal{X}$, $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* (\mathcal{X}', \mathcal{G}', \mathcal{R}', P')$ *for some* $\mathcal{X}'$ *or* $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* \bot$

Proof. By induction on $(\mathcal{G}, \mathcal{R}, P) \to^* (\mathcal{G}', \mathcal{R}', P')$. □

Conversely, for every checked trace, there is an unchecked trace that reaches the same result.

Lemma 3. *If* $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* (\mathcal{X}', \mathcal{G}', \mathcal{R}', P')$ *then* $(\mathcal{G}, \mathcal{R}, P) \to^* (\mathcal{G}', \mathcal{R}', P')$.

Proof. By induction on $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* (\mathcal{X}', \mathcal{G}', \mathcal{R}', P')$. □

The initial values of shared memory and registers do not affect whether a data race is detected:

Lemma 4. *If* $(\mathcal{X}, \mathcal{G}_1, \mathcal{R}_1, P) \int \bot$ *then* $(\mathcal{X}, \mathcal{G}_2, \mathcal{R}_2, P) \int \bot$.

Proof. By induction we can prove that, for all $\mathcal{X}, \mathcal{G}_1, \mathcal{R}_1, P, \mathcal{X}', \mathcal{G}_1', \mathcal{R}_1', P', \mathcal{G}_2, \mathcal{R}_2$, if we have $(\mathcal{X}, \mathcal{G}_1, \mathcal{R}_1, P) \hookrightarrow^* (\mathcal{X}', \mathcal{G}_1', \mathcal{R}_1', P')$, then $(\mathcal{X}, \mathcal{G}_2, \mathcal{R}_2, P) \hookrightarrow^* (\mathcal{X}', \mathcal{G}_2', \mathcal{R}_2', P')$ for some $\mathcal{G}_2', \mathcal{R}_2'$. Hence,

the desired result follows from the fact that RdMemBad and WrMemBad depend only on the context of memory events and the program. □

The next theorem proves that if there is a data race, then the symbolic evaluation detects it:

Theorem 1 (Data Race Freedom). *If P has a data race, then $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \int \bot$.*

Proof. The fact that P has a data race implies there are two traces—that is, derivations of $(\mathcal{G}_o, \mathcal{R}_o, P) \rightarrow^* (\mathcal{G}', \mathcal{R}', P')$ for some $\mathcal{G}_o$, $\mathcal{R}_o$, $\mathcal{G}'$, $\mathcal{R}'$, P'—with a common prefix α followed by conflicting accesses a_i, a_j in one case, and, in the other, a_j, a_i. By Lemma 2 on α, either we can obtain a derivation of $(\mathcal{X}, \mathcal{G}_o, \mathcal{R}_o, P) \hookrightarrow^* (\mathcal{X}_\alpha, \mathcal{G}_\alpha, \mathcal{R}_\alpha, P_\alpha)$ for some $\mathcal{X}_\alpha$, $\mathcal{G}_\alpha$, $\mathcal{R}_\alpha$, P_α, or $(\mathcal{X}, \mathcal{G}_o, \mathcal{R}_o, P) \int \bot$ and, by Lemma 4, we are done. In the former case, without loss of generality, let a_j be a write. Either we can append a_i to our $(\mathcal{X}, \mathcal{G}_o, \mathcal{R}_o, P) \hookrightarrow^* (\mathcal{X}_\alpha, \mathcal{G}_\alpha, \mathcal{R}_\alpha, P_\alpha)$ without obtaining $\bot$, or we are done by Lemma 4. In the former case, the write a_j now induces a $\neg$*noRacingRd* if a_i is a read or a $\neg$*noRacingWr* if a_i is a write, completing the proof by Lemma 4. □

The next theorem proves that if the symbolic evaluation detects a data race, then there is actually a data race:

Theorem 2 (True Positives Property). *If $(\mathcal{X}_\varnothing, \mathcal{G}, \mathcal{R}, P) \int \bot$ then P has a data race.*

Proof. Since $(\mathcal{X}_\varnothing, \mathcal{G}, \mathcal{R}, P) \int \bot$, the last step of our derivation is a read or a write for which we have a $\neg$*noRacingRd* or $\neg$*noRacingWr* for some $i, \mathcal{X}(g)$. Call this step a_i. Without loss of generality, we consider the case where a_i is a write. There must also be a write to g by some $j \neq i$, lest, by induction, our derivation does not hit $\bot$. Say the last such write, a_j, occurs at step n of the derivation. After the last write a_j, it grows monotonically easier to read/write g. Hence, there cannot be a **sync** I at any step $m > n$ such that $i, j \in I$, since it would be possible for i to read g immediately after the **sync** and, therefore, at all following steps. Hence, there is no **sync** I at step $m > n$ on thread i such that $j \in I$ or vice-versa, since that would induce a deadlock, but the hypothesis $(\mathcal{X}_\varnothing, \mathcal{G}, \mathcal{R}, P) \int \bot$ entails that the dynamics run to completion, producing $\bot$—in particular, they do not get stuck. From this fact, we conclude that we can shift our derivation so that a_j and a_i are next to each other. After all, the only steps from other threads that cannot be shifted past a_i are those Sync' where the set includes i and the only steps from other threads that cannot be shifted past a_j (i.e., deleted from the end of the trace) are those Sync' where the set includes j. Now, we can transform the piece of the trace before a_j, a_i from a $\cdot \hookrightarrow^* \cdot$ into a $\cdot \rightarrow^* \cdot$ by Lemma 3. Finally, we can append a_j, a_i to the result and, separately, a_i, a_j to the result to obtain the desired two traces. □

The theorems above characterize executions that run to completion (note the use of $\int$); the next two theorems account for executions that get stuck. We say that a program is *deadlocked* if no step can be taken, but the program is not of the form **return** $\| \cdots \|$ **return** and, in the checked case, the machine is not in state $\bot$.

Theorem 3 (Sound Deadlock Detection). *If any $\cdot \hookrightarrow \cdot$ trace deadlocks from some initial state, then all such traces from that state deadlock. Furthermore, if any $\cdot \hookrightarrow \cdot$ trace deadlocks, then some $\cdot \rightarrow \cdot$ trace deadlocks.*

Proof. The first statement is immediate from confluence, noting that, definitionally, the machine cannot step from a deadlocked state. The second statement is immediate from Lemma 3. □

Theorem 4 (Complete Deadlock Detection). *If some $\cdot \rightarrow \cdot$ trace deadlocks, then, for all initial states, all $\cdot \hookrightarrow \cdot$ traces deadlock or reach error state $\bot$.*

Proof. That, for each initial state $\mathcal{X}$, some $\cdot \hookrightarrow \cdot$ trace deadlocks or reaches error state $\bot$ is immediate from Lemma 2. Furthermore, again by confluence, all such traces must deadlock. □

One subtlety of the above theorems is that, if the unchecked semantics both races and deadlocks, the checked semantics might detect one error or the other, though it always detects the same error once we fix the initial state, $\mathcal{X}_\varnothing$ in the implementation since we want the true positives property. We now prove a few more lemmas characterizing $\cdot \hookrightarrow \cdot$ and $\cdot \to \cdot$ to build up to a proof of soundness and completeness for our equivalence checker.

The unchecked dynamics commutes with application of a map to shared memory and the registers. Together Lemmas 3 and 5 (below) should be thought of as a "sound abstraction" result: they say that running the checked semantics on a symbolic input tells us everything there is to know about running the unchecked semantics on concrete inputs, as we can take $\varphi : \mathbb{V} \to \mathbb{R}$ in Lemma 5:

Lemma 5. *For all maps φ, if $(\mathcal{G}, \mathcal{R}, P) \to^* (\mathcal{G}', \mathcal{R}', P')$ then $(\varphi(\mathcal{G}), \varphi(\mathcal{R}), P) \to^* (\varphi(\mathcal{G}'), \varphi(\mathcal{R}'), P')$.*

Proof. By induction on $(\mathcal{G}, \mathcal{R}, P) \to^* (\mathcal{G}', \mathcal{R}', P')$. □

Lemma 6. *If $(\mathcal{G}, \mathcal{R}, P) \downarrow (\mathcal{G}_1, \mathcal{R}_1)$ and $(\mathcal{G}, \mathcal{R}, P) \downarrow (\mathcal{G}_2, \mathcal{R}_2)$, then $(\mathcal{G}_1, \mathcal{R}_1) = (\mathcal{G}_2, \mathcal{R}_2)$ or, for all $\mathcal{X}$, $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \Big\downarrow \bot$.*

Proof. By Lemma 2, either $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* \bot$ and we are done, or we have $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* (\mathcal{X}_1, \mathcal{G}_1, \mathcal{R}_1, \textbf{return} \parallel \cdots \parallel \textbf{return})$ and $(\mathcal{X}, \mathcal{G}, \mathcal{R}, P) \hookrightarrow^* (\mathcal{X}_2, \mathcal{G}_2, \mathcal{R}_2, \textbf{return} \parallel \cdots \parallel \textbf{return})$ for some $\mathcal{X}_1, \mathcal{X}_2$. But, in the latter case, confluence yields a contradiction if $(\mathcal{G}_1, \mathcal{R}_1) \neq (\mathcal{G}_2, \mathcal{R}_2)$. □

Letting *Eq* denote our procedure for deciding equality between symbolic terms, we are finally ready to state soundness and completeness of equivalence checking. Soundness says that, if *Eq* decides that the outputs of the checked dynamics on two programs are equal on appropriate symbolic inputs, then the outputs of the unchecked dynamics on the two programs are equal for all real inputs (since we can write any real inputs as symbolic inputs under a substitution φ):

Theorem 5 (Soundness of Equivalence Checking). *For all $P, Q : \mathcal{L}$, contexts $\mathcal{G}, \mathcal{R}$ over $\mathbb{V}$, and maps $\varphi : (\mathbb{V}, \oplus) \to (\mathbb{R}, \oplus)$, if $(\mathcal{X}_\varnothing, \mathcal{G}, \mathcal{R}, P) \Big\downarrow (\mathcal{G}_P, \mathcal{R}_P)$, $(\mathcal{X}_\varnothing, \mathcal{G}, \mathcal{R}, Q) \Big\downarrow (\mathcal{G}_Q, \mathcal{R}_Q)$, $Eq(\mathcal{G}_P, \mathcal{G}_Q)$, $Eq(\mathcal{R}_P, \mathcal{R}_Q)$, $(\varphi(\mathcal{G}), \varphi(\mathcal{R}), P) \downarrow (\mathcal{G}'_P, \mathcal{R}'_P)$, and $(\varphi(\mathcal{G}), \varphi(\mathcal{R}), Q) \downarrow (\mathcal{G}'_Q, \mathcal{R}'_Q)$, then $\mathcal{G}'_P = \mathcal{G}'_Q$ and $\mathcal{R}'_P = \mathcal{R}'_Q$.*

Proof. By Lemmas 3 and 5, $(\varphi(\mathcal{G}), \varphi(\mathcal{R}), P) \downarrow (\varphi(\mathcal{G}_P), \varphi(\mathcal{R}_P))$ and $(\varphi(\mathcal{G}), \varphi(\mathcal{R}), Q) \downarrow (\varphi(\mathcal{G}_Q), \varphi(\mathcal{R}_Q))$. By Lemma 6, either $\varphi(\mathcal{G}_P) = \mathcal{G}'_P$ and $\varphi(\mathcal{R}_P) = \mathcal{R}'_P$ or $(\mathcal{X}_\varnothing, \varphi(\mathcal{G}), \varphi(\mathcal{R}), P) \Big\downarrow \bot$ and thus, by Lemma 4, $(\mathcal{X}_\varnothing, \mathcal{G}, \mathcal{R}, P) \Big\downarrow \bot$, a contradiction. Hence, $\varphi(\mathcal{G}_P) = \mathcal{G}'_P$ and $\varphi(\mathcal{R}_P) = \mathcal{R}'_P$. By the same argument, $\varphi(\mathcal{G}_Q) = \mathcal{G}'_Q$ and $\varphi(\mathcal{R}_Q) = \mathcal{R}'_Q$. Hence, if *Eq* indeed decides mathematical equality of the symbolic terms (see the Section 4.4), we have $\mathcal{G}'_P = \varphi(\mathcal{G}_P) = \varphi(\mathcal{G}_Q) = \mathcal{G}'_Q$ and $\mathcal{R}'_P = \varphi(\mathcal{R}_P) = \varphi(\mathcal{R}_Q) = \mathcal{R}'_Q$, as desired. □

Conversely, completeness says that, if *Eq* decides the outputs of the checked dynamics on two programs are not equal on symbolic inputs, then the outputs of the unchecked dynamics on the two programs are not equal for some real input (namely the symbolic input under substitution φ):

Theorem 6 (Completeness of Equivalence Checking). *For all $P, Q : \mathcal{L}$ and contexts $\mathcal{G}, \mathcal{R}$ over $\mathbb{V}$, if $(\mathcal{G}, \mathcal{R}, \mathcal{X}_\varnothing, P) \Big\downarrow (\mathcal{G}_P, \mathcal{R}_P)$, $(\mathcal{G}, \mathcal{R}, \mathcal{X}_\varnothing, Q) \Big\downarrow (\mathcal{G}_Q, \mathcal{R}_Q)$, and $\neg Eq(\mathcal{G}_P, \mathcal{G}_Q)$ or $\neg Eq(\mathcal{R}_P, \mathcal{R}_Q)$, there exists a map $\varphi : (\mathbb{V}, \oplus) \to (\mathbb{R}, \oplus)$ such that, if $(\varphi(\mathcal{G}), \varphi(\mathcal{R}), P) \downarrow (\mathcal{G}'_P, \mathcal{R}'_P)$ and $(\varphi(\mathcal{G}), \varphi(\mathcal{R}), Q) \downarrow (\mathcal{G}'_Q, \mathcal{R}'_Q)$, then $\mathcal{G}'_P \neq \mathcal{G}'_Q$ or $\mathcal{R}'_P \neq \mathcal{R}'_Q$.*

Proof. Without loss of generality, $\neg Eq(\mathcal{G}_P, \mathcal{G}_Q)$; the cases are symmetric. As we shall show in Section 4.4, $\forall a, b : \mathbb{V}$ such that $a \neq b$ there exists a map $\varphi : (\mathbb{V}, \oplus) \to (\mathbb{R}, \oplus)$ such that $\varphi(a) \neq \varphi(b)$. From this fact, we can obtain φ such that $\varphi(\mathcal{G}_P) \neq \varphi(\mathcal{G}_Q)$. By the same argument that we used in the proof of Theorem 5, if $(\varphi(\mathcal{G}), \varphi(\mathcal{R}), P) \downarrow (\mathcal{G}'_P, \mathcal{R}'_P)$ and $(\varphi(\mathcal{G}), \varphi(\mathcal{R}), Q) \downarrow (\mathcal{G}'_Q, \mathcal{R}'_Q)$, then $\varphi(\mathcal{G}_P) = \mathcal{G}'_P$ and $\varphi(\mathcal{G}_Q) = \mathcal{G}'_Q$. If $\mathcal{G}'_P = \mathcal{G}'_Q$, then $\varphi(\mathcal{G}_P) = \mathcal{G}'_P = \mathcal{G}'_Q = \varphi(\mathcal{G}_Q)$, a contradiction. Hence, $\mathcal{G}'_P \neq \mathcal{G}'_Q$ as desired. □

In fact, we only care about equality (or the lack thereof) on a subset of $\mathcal{G}_P, \mathcal{G}_Q$ and $\mathcal{R}_P, \mathcal{R}_Q$, namely the designated program outputs. Fortunately, the same steps prove the desired versions of Theorems 5 and 6, though their statements become ever more unwieldy.

4.4 Decision procedure

Thus far in the formalism, we have assumed, without loss of generality, a single binary operator $\oplus$. In the decision procedure, we support expressions involving additions, multiplications, and exponentiations over reals and, in this section, we show that it is possible to decide equalities of the form $f_1(x_1, \ldots, x_n) = f_2(x_1, \ldots, x_n)$ where f_1 and f_2 involve such expressions. Although we do not give a formal account of other functions such as max, the implementation supports them. In principle, min and max could be handled by case splits without affecting decidability. In the ML kernels we verify, max and min arise only in restricted patterns, such as the max-subtraction of softmax, so we are able to sufficiently normalize them and do not split. We are not aware of a proof of decidability for our desired equalities, though there is work on related problems, such as on Tarski's exponential function problem [48]. Closest to our result is Lemma 16 of Li and Wu [42] (the basis of Mirage's verifier [75]), a univariate analogue that additionally permits ratios of polynomials in the exponents. However, its proof invokes the Lindemann–Weierstrass theorem and, consequently, applies only when the coefficients involved are algebraic numbers. Our proof is direct, avoids heavy number-theoretic machinery, and handles arbitrary real coefficients.

If there were no exponentiations, decidability would be immediate. Because $f_1 = f_2$ iff $f_1 - f_2$ is identically zero, and $f_1 - f_2$ can be rewritten as a sum of monomials and such a sum is identically zero iff the coefficients of all the monomials are zero. We show that even with exponentiations, identities $f_1 = f_2$ are decidable.

Without loss of generality, we assume that the polynomial in the exponent of terms $p(x)e^{h(x)}$ has a zero constant, i.e., $h(0) = 0$. If $h(0) \neq 0$ then we can rewrite $p(x)e^{h(x)}$ as $p'(x)e^{h'(x)}$ where $p'(x) = p(x)e^{h(0)}$ and $h'(x) = h(x) - h(0)$.

Theorem 7. *Let $m \geq 1$ be an integer. Suppose that $\mathbb{R}[x]$ polynomials $p_1(x), \ldots, p_m(x)$ and $h_1(x), \ldots, h_m(x)$ satisfy $\forall i \neq j. h_i(x) \neq h_j(x)$ and $\sum_{i=1}^{m} p_i(x)e^{h_i(x)} = 0$ then it must be that $p_1(x) = p_2(x) = \ldots = p_m(x) = 0$.*

Proof. Proof by induction on m. For the base case with $m = 1$, we have $p_1(x)e^{h_1(x)} = 0$. Since $e^{h_1(a)} > 0$ for all a, it must be that $\forall a. p_1(a) = 0$ and hence $p_1(x) = 0$.

For the induction case, assume that the result holds for $(m-1)$. Suppose WLOG that the leading coefficient of $h_m(x)$ is the largest among the leading coefficients of the $h_i(x)$'s. Then the leading coefficient of $h_i(x) - h_m(x)$ is negative for all $i < m$. This implies, $\forall i < m. \lim_{x\to\infty} e^{h_i(x)-h_m(x)} = 0$.

We are given, $p_1(x)e^{h_1(x)-h_m(x)} + p_2(x)e^{h_2(x)-h_m(x)} + \ldots + p_{m-1}(x)e^{h_{m-1}(x)-h_m(x)} + p_m(x) = 0$. Taking limits as x approaches ∞, we have:

$$\left(\sum_{i<m} \lim_{x\to\infty} p_i(x)e^{h_i(x)-h_m(x)}\right) + \lim_{x\to\infty} p_m(x) = 0$$

Therefore, $\lim_{x\to\infty} p_m(x) = 0$. But for a non-zero polynomial p, $\lim_{x\to\infty} p(x)$ is either ∞ if the leading coefficient of p is positive and $-\infty$ if negative. Hence, $p_m(x) = 0$. □

Hence, in the uni-variate case, identity checking with exponentiations reduces to identity checking without exponentiations, which is decidable as discussed earlier. The above theorem for univariates generalizes to multi-variate polynomials.

Corollary 8. *Let $m \geq 1$ be an integer. Suppose that $\mathbb{R}[\bar{x}]$ multivariate polynomials $p_1(\bar{x}), \ldots, p_m(\bar{x})$ and $h_1(\bar{x}), \ldots, h_m(\bar{x})$ satisfy $\forall i \neq j. h_i(\bar{x}) \neq h_j(\bar{x})$ and $\sum_{i=1}^{m} p_i(\bar{x}) e^{h_i(\bar{x})} = 0$ then it must be that $p_1(\bar{x}) = p_2(\bar{x}) = \ldots = p_m(\bar{x}) = 0$.*

Proof. Consider the substitution $x_1 = \alpha_1 y, x_2 = \alpha_2 y, \ldots, x_n = \alpha_n y$ where each $\alpha_i \in \mathbb{R}$ are chosen independently at random. Making this substitution, we get a univariate functional identity of the form

$$\hat{p}_1(y) e^{\hat{h}_1(y)} + \hat{p}_2(y) e^{\hat{h}_2(y)} + \ldots + \hat{p}_m(y) e^{\hat{h}_m(y)} = 0 \tag{2}$$

where for each $i \in \{1, \ldots, m\}$, $\hat{p}_i(y) = p_i(\alpha_1 y, \alpha_2 y, \ldots, \alpha_n y)$ and $\hat{h}_i(y) = h_i(\alpha_1 y, \alpha_2 y, \ldots, \alpha_n y)$.

From our convention, $\hat{h}_i(0) = h_i(0, 0, \ldots, 0) \neq 0$. For $i \neq j$, consider

$$\hat{h}_i(y) - \hat{h}_j(y) = h_i(\alpha_1 y, \ldots, \alpha_n y) - h_j(\alpha_1 y, \ldots, \alpha_n y)$$

Let the degree of $h_i(\bar{x}) - h_j(\bar{x})$ be d and the largest degree homogeneous component of $h_i(\bar{x}) - h_j(\bar{x})$ be $g_d(\bar{x}) \neq 0$. Then, $\hat{h}_i(y) - \hat{h}_j(y)$ is a polynomial of degree at most d and the coefficient of y^d in $\hat{h}_i(y) - \hat{h}_j(y)$ is $g_d(\alpha_1, \alpha_2, \ldots, \alpha_n)$. By Schwartz Zippel Lemma, $g_d(\alpha_1, \alpha_2, \ldots, \alpha_n) \neq 0$ almost surely and so almost surely (over the random choice of $\bar{\alpha} = (\alpha_1, \ldots, \alpha_n)$) we have $\forall i \neq j : \hat{h}_i(y) \neq \hat{h}_j(y)$. Applying the theorem to Equation 2, we get almost surely (over the random choice of $\bar{\alpha}$, $\hat{p}_1(y) = \ldots = \hat{p}_m(y) = 0$). But $\hat{p}_i(y) = p_i(\alpha_1 y, \ldots, \alpha_m y)$ and this must imply that $p_i(\bar{x}) = 0$ (via Schwartz Zippel Lemma applied to the largest homogeneous component of $p_i(\bar{x})$). Thus, $\forall 1 \leq i \leq m. p_i(x) = 0$. □

Note that the theorem and corollary require the exponents h_i to be polynomials. Nested exponentials, as would arise, for example, if one softmax were composed with another, are not covered by our decidability proof. Our implementation simplifies nested exponentials and ratios of functions in exponents if they arise, but it may not be complete for such expressions.

Given our interest in proving completeness of the overall equivalence checking procedure, we do not focus on the computational complexity of the decision procedure and leave it for future work. Our implementation (Section 5) is able to handle ML kernels that arise in practice (Section 6). Finally, to make good on our claim in the proof of completeness for equivalence checking that $\forall a, b : \mathbb{V}$ such that $a \neq b$ there exists a map $\varphi : (\mathbb{V}, \oplus) \to (\mathbb{R}, \oplus)$ (more precisely $(\mathbb{V}, +, \cdot, \exp) \to (\mathbb{R}, +, \cdot, \exp)$) such that $\varphi(a) \neq \varphi(b)$, we remark that a, b can be viewed as polynomials in $\mathbb{R}[\bar{x}]$ for some $\bar{x}$. By contraposition of the above theorem, recalling that each p_i can be written as a sum of monomials, if $a \neq b$ (that is, $a - b$ is not formally 0), they must be unequal on some input $\bar{r}$. Then,

$$\varphi(x) = \begin{cases} r_i & \text{if } x = x_i \text{ for some } x_i \in \bar{x} \\ x & \text{otherwise} \end{cases}$$

can be extended in the obvious way to the desired map $(\mathbb{V}, +, \cdot, \exp) \to (\mathbb{R}, +, \cdot, \exp)$.

5 Implementation

We implement our techniques in a proof-of-concept tool, Volta, written in Rust.[10] Volta operates on CTAs written in PTX ISA 9.1[11]. In addition to the PTX program, it accepts specification of parameters such as the grid dimensions and the locations and sizes of input tensors as well as the output tensors that should be compared for equivalence. Volta is comprised of two main components: a simulator that performs symbolic execution to generate verification conditions (VCs), and a decision procedure that resolves them. For a program where each thread runs at most N statements, the simulator's runtime is linear in N, whereas the decision procedure exhibits superlinear complexity in N. The rest of this section will relay the key details of symbolic execution and the decision procedure, though it does not exhaustively describe their implementations.

Before symbolic execution, the input PTX is lowered, resolving variable, parameter, and register names, as well as labels. During symbolic execution, for each shared or global memory cell that has been written, we track an expression and the value of $\mathcal{X}$ (see Section 4.2) at that cell, where $\mathcal{P}(\text{TID})$ is implemented via a fixed-width bitset. In fact, memory cells additionally track the width of the last store and require (to a first approximation) that reads are of the same width—it is not clear what it means to read half of a real valued symbolic expression. Each expression is passed around as a 4-byte index into a bump allocator holding a 16-byte representation of the expression. Constant folding, including simplification of $\min(\infty, x)$, $\min(x, \infty)$, $\max(-\infty, x)$, and $\max(x, -\infty)$, is performed eagerly as expressions are constructed, with floating point constants represented via GMP's [27] arbitrary precision rationals. Then, it is a trivial matter to resolve branches and memory accesses as they occur, should they be fully statically determined, or else to return an error.

Threads are scheduled in a round-robin fashion, with each selected thread being run until it blocks at a barrier or exits. If all threads are blocked, the scheduler checks whether any barriers can be released because all threads that are expected to reach them have done so. If not, a deadlock is reported—a strategy justified by Theorems 3 and 4. At any point, a race might be detected via $\mathcal{X}$, it might be decided that the program is not a structured-CTA because it attempts to branch or index memory using a symbolic value, or some other violation might be detected such as an out-of-bounds read. If the symbolic execution succeeds, the verification conditions (VCs) corresponding to the output tensors are given to a decision procedure. In particular, for each pair of corresponding output tensor elements p_1 and p_2, the decision procedure checks whether $p_1 - p_2$ is identically zero.

Volta currently assumes that CTAs behave uniformly and simply checks CTA 0 of the reference against CTA 0 of the optimized program. If the optimizations are intra-CTA, checking multiple CTAs is embarrassingly parallel, but it might become intractably expensive for real, multi-billion parameter models. It is worth noting that, while the number of CTAs grows, the per-CTA tensor size does not.

The decision procedure consists of a canonicalizer producing a ratio of polynomials (rational function), each polynomial consisting of a sorted list of terms along with their coefficients, each term consisting of a sorted list of factors along with their exponents, optionally multiplied by e to some (non-zero) polynomial, each factor consisting of a symbolic input, an uninterpreted function of rational functions, or a min or max of rational functions. Min and max factors store their arguments sorted and deduplicated, and nested min and, respectively, max factors are flattened. Everything is hash-consed, providing an arbitrary but stable sort order. The results of canonicalization are memoized/cached based on the generation time indices that represent expressions, but only if an expression is referenced by multiple other expressions, where reference counts are determined by a walk of the generation phase expression store (the bump allocator) before running the decision

[10] https://github.com/willtunnels/volta
[11] https://docs.nvidia.com/cuda/parallel-thread-execution/

procedure. This check significantly reduces the size of the memoized data, e.g., in the presence of accumulation chains, and the memoized data in not incrementally collected.

The content of Corollary 8 is that the above procedure is complete for the subclass of expressions $\sum_{i=1}^{m} p_i(\bar{x})e^{h_i(\bar{x})}$. It says such functions are determined by their polynomial coefficients (which are themselves determined by their monomial coefficients). Canonicalization requires distributing multiplication over addition to find the coefficient of each monomial term. This operation, in principle, may cause exponential blowup. Since machine learning workloads do not typically have computations with high multiplicative depth, this blowup does not happen in practice.

It is worth noting that Volta's implementation is currently single threaded. One could try to parallelize VC generation by simulating the CUDA parallel machine with a parallel machine. The decision procedure would be a more interesting task because of canonical form memoization, out of which the implementation gets quite a bit of mileage, as we shall see (Section 6.5). We leave parallelization to future work.

Finally, we give an inventory of the PTX operations Volta currently supports. It supports collective operations `bar.sync`, `bar.warp.sync`, `shfl.sync`, `activemask`, `ldmatrix`, `mma.sync`, and `wmma`. It can symbolically compute addition, multiplication, division, reciprocal, fused multiply-add, and exponentiation along with with max and min, float-width conversions (modeled as the identity), and floating-point saturation and ReLU clamps (modeled via min/max). Some additional operations can be performed on statically known values. We do not support square root or trigonometric functions. These operations are tricky because they place requirements on their arguments, e.g., square root requires a positive argument. If the set of operations appearing in expressions grows sufficiently complex, an SMT solver might be necessary.

6 Evaluation

There are three main ways to create efficient GPU kernels for machine learning workloads: manual optimization (Section 6.1), generation by LLMs (Section 6.2), and compilation by domain-specific compilers (Section 6.3). Since our equivalence checker operates as a black box and makes no assumptions about the optimization process, it can verify the correctness of optimized kernels produced by all three approaches. We evaluate on reductions, matrix multiplications, convolutions, and various attention mechanisms. We report the time to generate the verification conditions (VCs), which includes symbolically executing both kernels ("VC Gen"), and the time to solve them ("VC Time"), measured on a machine with a 128-core 2.25 GHz AMD EPYC 7742 processor and 995 GB of RAM. We show that Volta scales to realistic machine learning workloads and effectively identifies correctness issues when they exist. In particular, Volta detects data races that have been previously reported in open-source GitHub repositories (Section 6.4).

6.1 Human-generated Kernels

6.1.1 Reduction. "Optimizing Parallel Reduction in CUDA" by Harris [29] is a well-cited tutorial that introduces a sequence of progressively optimized reduction kernels. It begins with a baseline implementation, Red-1, and presents seven versions, each achieving higher performance than the previous one. Each CTA reduces 128 values using up to 128 threads. The first optimization, Red-2, removes divergent branching and achieves more than a 2× speedup. The next version, Red-3, mitigates shared-memory bank conflicts for another 2× improvement. Red-4, which further reduces the number of idle threads, yields an additional reported speedup of 1.78×. The subsequent optimizations rely on warp-synchronous execution, which was later deprecated by NVIDIA. Volta detects data races in these versions, Red-5, Red-6, and Red-7, and correctly rejects them, and Volta proves that the other kernels are equivalent to Red-1.

Table 1 reports the running time of Volta and other key statistics, such as the number of CTA-wide synchronization operations and PTX instructions, for the optimized kernels. Red-1 through Red-4 were checked for equivalence with Red-1; for the racy versions, we report the time for symbolic execution to detect the race. "#Block Sync" represents the total number of `bar.sync` PTX instructions executed across all threads. Finally, verifying the equivalence of a kernel against itself, here Red-1 vs. Red-1, serves as a sanity check, confirming that the reference implementation is free of synchronization errors.

Table 1. Time to symbolically execute and check verification conditions (VCs) for kernels from NVIDIA's reduction tutorial [29]. Each kernel runs as a single CTA of 128 threads; Red-1–Red-4 reduce 128 elements and were checked for equivalence with Red-1. Volta rejects the racy Red-5–Red-7 during symbolic execution, before any VCs are generated, so their VC Gen entries report the time to detect the race.

Kernel	VC Gen (s)	VC Time (s)	#Block Sync	PTX LOC
Red-1	0.0038	0.0005	1k	76
Red-2	0.0023	0.0004	1k	79
Red-3	0.0022	0.0004	1k	73
Red-4	0.0022	0.0004	896	75
Red-5	0.0005	—	—	156
Red-6	0.0003	—	—	104
Red-7	0.0005	—	—	98

6.1.2 MatMul. "How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance" by Boehm is a widely referenced tutorial in the GPU community [11] presenting SGEMM (single-precision general matrix multiplication) implementation techniques. Similar to the reduction tutorial, it presents a series of kernels, each achieving progressively higher performance: starting from 1% of the performance of NVIDIA's cuBLAS (proprietary and not open source) and reaching 95%. Each CTA computes a 64×64 tile of the output matrix using up to 512 threads. Starting from a baseline implementation, MatMul-1, the subsequent versions, MatMul-2 through MatMul-7, apply a sequence of standard optimizations, including coalesced global-memory accesses, use of shared-memory for caching, reducing bank conflicts, multiple levels of tiling (including warp tiling), vectorized memory operations, and auto-tuning to determine the optimal sub-tile sizes for intermediate computations. As with the reduction kernels, Table 2 reports the running time, the number of CTA-wide synchronization operations, and the number of PTX instructions for each optimized variant. All kernels were checked for equivalence with MatMul-1. In contrast to the reduction tutorial, all MatMul kernels pass the equivalence check without data races.

6.1.3 Flash attention. The performance of LLM inference is often bottlenecked by the attention operation. Consequently, optimizing attention has become a central topic in the performance engineering of LLMs. We start from a naive reference implementation, Attention, that uses a single thread to compute the full attention operation, i.e., $softmax(QK^T/\sqrt{d})V$. We evaluate the setup where each CTA computes an attention head with standard matrix dimensions $Q : (16, 64)$, $K : (512, 64)$, and $V : (512, 64)$. The optimized implementations we consider next use 128 threads.

FA1 applies the FlashAttention [21] tiling strategy to reduce global-to-shared memory transfers and adopts the online softmax formulation to enable streaming. FA1-TC further accelerates FA1 by leveraging tensor cores. Finally, FA2-TC also uses tensor cores and incorporates the techniques of FlashAttention-2 [20], such as deferring softmax normalization and improving work partitioning among warps. Table 3 reports the checker runtime and statistics for these optimized variants, including the number of CTA-level barriers (`bar.sync`), warp-level barriers, PTX instruction counts,

Table 2. Symbolic execution and VC checking time for kernels from matrix multiplication tutorial [11]. All kernels were checked for equivalence with MatMul-1. Each CTA computes a 64×64 tile of $C = \alpha AB + \beta C$ with M=N=K=4096; MatMul-1 and MatMul-2 use 512 threads per CTA, MatMul-3 through MatMul-5 use 128, and MatMul-6 and MatMul-7 use 256.

Kernel	VC Gen (s)	VC Time (s)	#Block Sync	PTX LOC
MatMul-1	45.4	89.4	524k	185
MatMul-2	38.5	90.2	524k	269
MatMul-3	30.4	90.5	131k	325
MatMul-4	30.7	91.7	131k	323
MatMul-5	30.6	92.8	131k	320
MatMul-6	34.7	94.0	262k	307
MatMul-7	34.5	93.5	262k	307

and the number of threads. "#Warp Sync" represents the total number of warp-level synchronization PTX instructions such as `bar.warp.sync`, `mma.sync`, etc., executed across all threads. Note that kernels in Table 1, Table 2, Attention, and FA1 do not use tensor cores. Thus, they have zero warp-level synchronization operations. All kernels were checked for equivalence with Attention.

Table 3. VC generation and checking time for attention kernels. All kernels were checked for equivalence with Attention. Each kernel computes one attention head with $Q : (16, 64)$, $K : (512, 64)$, and $V : (512, 64)$ using a single CTA with the listed number of threads.

Kernel	VC Gen (s)	VC Time (s)	#Block Sync	#Warp Sync	PTX LOC	#Threads
Attention	3.9	3.0	0	0	477	1
FA1	6.7	153.8	187k	0	1138	128
FA1-TC	8.1	154.5	801k	204k	1237	128
FA2-TC	4.1	154.0	210k	13k	2048	128

Versions of attention which feature causal masking are commonly used in auto-regressive models. We verify that the efficient "fused" implementation of causal masking, wherein we skip any blocks such that all the column indices are more than the row indices, is equivalent to a naive approach implemented via an element-wise product with a constant triangular matrix. See Table 4.

VC checking time for these benchmarks is significantly lower than the VC checking time for the benchmarks in Table 3. The symbolic expressions are smaller due to masking and time is non-linear in expression size. Moreover, the two expressions being compared are actually precisely the same—since the only difference is the method of filtering and this difference evaporates during symbolic evaluation—allowing the implementation to avoid work via the cache.

6.2 LLM-generated Kernels

In a recent blog post, Stanford researchers demonstrated how LLMs can generate optimized GPU kernels for 2D convolution [58]. We verify the equivalence of the final LLM-generated kernel after 13 rounds of optimization with a reference implementation. The applied optimizations include transforming the convolution into a matrix multiplication to leverage tensor cores, pre-computing intermediate values to eliminate redundant arithmetic, caching indices in shared-memory, employing software pipelining, reusing address computations, and applying vectorization. Overall, the final kernel achieves 179.9% of PyTorch performance. In this setup, each CTA uses 256 threads to compute a 128×128 output tile. Table 5 reports Volta's running time and other key statistics. The use of tensor cores significantly increases the code size of the optimized kernel compared to

Table 4. VC generation and checking time for causal attention kernels with a lower triangular mask. For all kernels, a fused version of the kernel was checked against a naive version of the kernel. The first element of PTX LOC is the for the fused implementation, and second element is for the naive implementation. Dimensions and launch configurations match those of Table 3.

Kernel	VC Gen (s)	VC Time (s)	#Block Sync	#Warp Sync	PTX LOC	#Threads
Causal-Attention	3.9	0.05	0	0	(483, 480)	1
Causal-FA1	9.4	0.08	187k	0	(1141, 1142)	128
Causal-FA1-TC	11.8	0.13	801k	204k	(1241, 1242)	128
Causal-FA2-TC	4.3	0.07	210k	13k	(2081, 2088)	128

the reference. The reference has no synchronization operations whereas the final version has 6.4k CTA-wide and 102k warp-level barriers.

Table 5. Symbolic execution and VC checking time for the LLM-generated 2D convolution [58], checked against a naive reference implementation. In the pairs $(-,-)$, the first element is for the reference implementation, and the second element is for the optimized implementation. The kernels compute a 2D convolution (batch 100, $3 \times 224 \times 224$ inputs, 96 11×11 filters, stride 4, padding 2) expressed as an implicit GEMM with M=96, N=302,500, K=363; each CTA uses 256 threads and computes a 128×128 output tile.

Kernel	VC Gen (s)	VC Time (s)	#Block Sync	#Warp Sync	PTX LOC	#Threads
Conv2D	32.0	14.3	(0, 6.4k)	(0, 102k)	(463, 1312)	(256, 256)

Through the course of this verification, we discovered interesting undocumented behaviors. For example, out-of-bounds reads from shared-memory arrays may not cause a crash and may also go undetected by NVIDIA's Compute Sanitizer [56]. The following code segment—which uses two warps (64 threads) to read 48 elements of the shared-memory array A into registers v—runs without raising any exceptions on the Ampere, Hopper, and Blackwell GPUs we tested on, even though it performs out-of-bounds accesses.

```
__shared__ int A[48];
...
v = A[tid]; // threads 48 to 63 read out-of-bound
if (tid < 48){
  use V
...
```

The out-of-bound reads store zeros in register v and do not crash the tests. Volta (correctly) catches them and raises exceptions. An appropriate fix is to rewrite the code, as shown below:

```
__shared__ int A[48];
...
if(tid < 48){
  v = A[tid]; // all accesses within bounds
  use V
...
```

Although the two programs behave identically on current GPUs, there is no guarantee that future GPU hardware would continue to tolerate out-of-bound reads to shared memory. The latter code segment is correct for future hardware as well and, hence, preferable. Thus, even though we focus on verifying correctness of optimizations in this paper, we believe Volta can also help improve code quality.

Next, we use Claude Code (Anthropic's LLM-based coding tool) to improve performance through an agentic setup that automatically generates and runs tests for correctness checking [2]. In Table 6, we start with a reference implementation for a matrix multiplication, MatMul-1, adapted from [11], where the CTAs output 32×32 tiles. Claude code is able to generate various kernels where GEMM-1 and GEMM-2 have shared-memory optimizations, and GEMM-3 uses tensor cores. Volta is successfully able to prove the equivalence of all three with the reference implementation.

Table 6. Symbolic execution and VC checking time for matrix multiplication kernels generated by Claude Code. All kernels were checked for equivalence with MatMul-1. Each CTA uses 512 threads to compute a 32×32 tile of $C = AB$ with M=N=K=4096.

Kernel	VC Gen (s)	VC Time (s)	#Block Sync	#Warp Sync	PTX LOC	#Threads
GEMM-1	36.4	13.2	524k	0	562	512
GEMM-2	39.9	13.1	262k	0	587	512
GEMM-3	28.4	13.1	263k	98k	771	512

6.3 Compiler-generated Kernels

TileLang [73] is a compiler that takes as input Python code that uses TileLang's DSL API and generates CUDA kernels under different configuration settings. We use it to produce multiple implementations of matrix multiplication and verify their pairwise equivalence. Any equivalence failure would indicate a correctness bug in the TileLang compiler. Table 7 shows that the running time of Volta scales gracefully with increasing tile sizes. Note that these tiles cannot grow unbounded in size as then they would exceed the resource limits that the hardware imposes on each CTA. Hence, Table 7 evaluates on some common tile sizes, where $A \times B \times C$ indicates a multiplication of a matrix with dimensions $A \times B$ with a matrix of dimensions $B \times C$. Table 7 also reports detailed statistics for each pair of programs under verification. In all cases, the optimized implementations use tensor cores whereas their reference counterparts do not. The first elements of the pairs in Table 7 refer to the statistics for the reference implementations and the second elements correspond to the optimized implementations. The optimized implementations call into CUTLASS templates, thus demonstrating Volta's abilities to handle PTX assembly arising from CUTLASS [19].

Table 7. VC generation and checking time for matrix multiplication kernels generated by TileLang [73]. In each pair $(-,-)$, first element is the reference implementation, and second element is the optimized implementation. All kernels multiply f16 matrices with M=N=K=4096; each CTA computes one output tile of the listed size using the listed number of threads.

Tile size	VC Gen (s)	VC Time (s)	#Block Sync	#Warp Sync	PTX LOC	#Threads
$32 \times 32 \times 32$	1.1	2.5	(8k, 8k)	(0, 32k)	(322, 455)	(128, 128)
$64 \times 32 \times 32$	4.5	10.9	(16k, 16k)	(0, 115k)	(531, 537)	(128, 128)
$64 \times 64 \times 32$	18.6	60.4	(32k, 32k)	(0, 393k)	(632, 842)	(128, 128)

6.4 Data Race Detection Baseline

FaialAA is the state-of-the-art data race checker for GPU programs [45]. Its canonical failure case, as described by the authors, is the following pattern:

```
1. M[tid] = tid;
2. x = M[tid];
```

Table 8. Racy FaialAA GitHub benchmarks; races are also detected by Volta before the fix, and not detected after the fix. BucketPositions, ComputeRange, and ReduceValue run as a single CTA of 64 threads; LayerNorm runs as a single warp (32 threads); GradInput runs with 32×4 threads before the fix and 32×1 after.

Kernel	Library	PTX LOC	Bug Fix Commit
BucketPositions	OpenMM	94	`9abaa587c`
ComputeRange	OpenMM	132	`65caf22b4`
ReduceValue	OpenMM	1190	`de666e305`
LayerNorm	MegatronLM	924	`4831071c9`
GradInput	MegatronLM	546	`f1295380c`

```
3. M[x] = ...;
```

FaialAA cannot track the concrete value of `x` at line 2 and, therefore, models it as non-deterministic. Subsequently, line 3 is reported as a (spurious) data race. Since we use symbolic evaluation, and `M[x]` is constant (since `tid` is constant), we don't report an error and declare the write in line 3 correctly as data race free.

FaialAA evaluates on GitHub commits that fix data races. These programs come from well-used open source repositories OpenMM[12] and NVIDIA's MegaTron[13]. Empirically, on the programs where FaialAA successfully detects data races, Volta succeeds as well, i.e., the data races are caught by the symbolic evaluation. Volta also successfully proves data race freedom for the versions of these kernels after the bug-fixing commits. These programs are shown in Table 8 for completeness.

6.5 Equivalence Checking Baseline

To check that our decision procedure yields some benefit, we compare it to the Z3 [23] SMT solver, an "off-the-shelf" solution, giving each the VC for a single element of each benchmark's output tensor. Table 9 reports the running time of Volta's decision procedure and of Z3, with and without the exponential-product axiom, $\forall x, y.\ e^x e^y = e^{x+y}$, on the benchmarks of Section 6. The Z3 times and the Volta "Single" times are for verifying the VC of a single element of the output tensor in isolation. The Volta "Median" column reports the median per-element time from the full-footprint runs of Section 6, in which Volta checks every output element.

On the 18 benchmarks whose VCs contain no exponentials (the reduction, matrix multiplication, and convolution kernels), Z3 succeeds. On the eight attention benchmarks, whose VCs contain exponentials by way of softmax, Z3 succeeds only on those VCs which are trivial after symbolic execution (exactly the form $p - p$ for some p). Adding the axiom does not help—Z3 simply runs longer before giving up. Because Volta caches canonical forms, taking advantage of the structure of our domain, an element checked after its neighbors is one to two orders of magnitude cheaper than one checked in isolation. In contrast, Z3 solves each element's VC as an independent query at unchanged cost. There is also the fixed cost Volta pays for determining expression reference counts (see Section 5), which is amortized over canonicalizing multiple elements. Hence, Volta's first solve time is significantly than Z3's, but its median solve time is significantly lower.

7 Related Work

Prior work on equivalence checking does not support GPU parallelism or synchronization [6, 15, 28, 34, 46, 51, 52, 62, 64, 65, 67, 68, 71, 75]. Alive [47] symbolically evaluates straight-line single-threaded programs over bit-vectors to generate decision problems for equivalence checking that

[12] https://github.com/openmm/openmm
[13] https://github.com/NVIDIA/Megatron-LM

Table 9. Per-element VC times using Volta's decision procedure and Z3, without and with the axiom $\forall x, y.\ e^x e^y = e^{x+y}$. "Volta First" solves the VC of one output tensor element in isolation; "Volta Median" is the median per-element time when solving the VCs of all output elements in one pass, where later elements reuse work via a cache and the cost of determining expression reference counts is amortized. For the reductions, whose output is a single element, the two Volta columns measure the same work in different runs. The axiom is relevant only when a VC contains exponentials, so it is omitted for the other benchmarks. "Unknown" indicates that Z3 returned unknown after the parenthesized time. Both Volta and Z3 version 4.8.12 were given a single thread and a 10 minute timeout, running on a 128-core 2.25 GHz AMD EPYC 7742 processor with 995 GB of RAM.

Kernel	Volta First (ms)	Volta Median (ms)	Z3 (ms)	Z3 + Axiom (ms)
Red-1	< 1	< 1	3.6	—
Red-2	< 1	< 1	3.6	—
Red-3	< 1	< 1	3.6	—
Red-4	< 1	< 1	3.6	—
MatMul-1	435	18	45	—
MatMul-2	358	17	47	—
MatMul-3	385	17	47	—
MatMul-4	383	17	49	—
MatMul-5	383	17	48	—
MatMul-6	385	18	50	—
MatMul-7	388	18	47	—
Conv2D	313	< 1	5.1	—
GEMM-1	165	10	50	—
GEMM-2	163	10	49	—
GEMM-3	162	10	47	—
$32 \times 32 \times 32$	16	1.9	9.3	—
$64 \times 32 \times 32$	46	4.4	17	—
$64 \times 64 \times 32$	217	12	45	—
Attention	141	1.3	395	388
FA1	6385	90	Unknown (3.0 s)	Unknown (319 s)
FA1-TC	6481	88	Unknown (3.0 s)	Unknown (319 s)
FA2-TC	6382	89	Unknown (3.0 s)	Unknown (317 s)
Causal-Attention	11	< 1	3.7	3.6
Causal-FA1	33	< 1	3.8	3.8
Causal-FA1-TC	98	< 1	3.8	3.8
Causal-FA2-TC	20	< 1	3.9	3.8

can be handed over to SMT solvers. Volta can be seen as generalizing this mechanism to many threads running in parallel that synchronize with each other to perform computations over the reals. Treating floating point operations as uninterpreted functions renders standard GPU optimizations unsound [4, 18], while fully modeling IEEE-754 semantics is known to cause equivalence checking failures for such optimizations [37]. Mirage [75] operates at a higher level than Volta, directly on a language of multi-linear operators, division, and exponentiation. Ultimately, the final verification conditions look quite similar to Volta's, but with the allowance of rational functions. Mirage tackles these conditions via the identity-testing procedure of Li and Wu [42], which, unlike our algorithm, lacks provable soundness (see Section 4.4).

Several existing approaches bound the worst-case numerical error in floating point implementations [22, 35, 36, 59, 70]. However, the worst-case errors when arithmetic operations are reordered in tensor computations (as in GPU optimization) are huge for contrived inputs and do not provide

useful insights. For ensuring memory safety in floating point programs, enforcing non-interference between floating point and non-floating point values has proven useful [26].

During its operation, the equivalence checker also establishes well-synchronization, guaranteeing both deadlock freedom and the absence of data races. There is a rich body of work that focuses exclusively on data race detection in GPU programs. Traditional data race analyses consider only *where* memory accesses occur, ignoring *what* data is being accessed, whereas an equivalence checker must reason about both. FaialAA [45] represents the state of the art in sound data race freedom checking, but, like most prior work, supports only basic synchronization primitives such as `syncthreads` and does not support `syncwarp`. A notable exception is Weft [66], which addresses producer-consumer synchronization that lies outside the scope of our work. Volta achieves higher precision than FaialAA because its symbolic evaluation aggressively propagates constant values; for example, it does not report the spurious data race identified by FaialAA in its Alarm 4, Figure 1 [45] (Section 6.4). However, FaialAA provides sound (though incomplete) analysis for GPU kernels beyond structured-CTAs, including kernels with non-trivial control flow and statically unknown array indices, which Volta rejects outright. The primary advantages of Volta's race checker are its simplicity and efficiency: it runs online during symbolic execution, maintaining only the small additional context $\mathcal{X}$ of Section 4.2, and adds time linear in the number of executed instructions. Other approaches either suffer from exponential blowup [41, 44], fail to detect certain data races [25, 30, 31, 43, 60, 61, 76, 78, 79], or exhibit high false positive rates [10, 33]. Several analyses rely on reductions [7, 16, 17, 39, 40, 45] that are sound only when the synchronization primitive is a barrier across all threads. In these methods, the kernel analysis can be partitioned into *barrier intervals*, defined as code segments between two consecutive executions of `syncthreads`. However, in the presence of `syncwarp` and tensor core operations, which synchronize only subsets of threads, the barrier interval assumption no longer holds. Finally, *test amplification* [38] can be seen as a special case of Volta's data race checking, where all threads run identical commands and synchronize exclusively with barriers across all threads.

Lifting systems seek to translate code in low-level, typically general purpose, languages into code in higher-level, typically domain specific, languages so that code can be more easily maintained and better re-optimized, retargeted, or parallelized by compilers. C2TACO [49] lifts C to TACO, as its name suggests, STNG [32] lifts Fortran to a stencil DSL and then translates to Halide, Tenspiler [63] lifts C, C++, or Python to a tensor IR and then translates to various backends, such as PyTorch, and LLMLift [8] lifts C, C++, or Python to a custom Python DSL and then translates to a similar set of backends. C2TACO does not provide formal guarantees about its output, instead relying on random testing. However, the latter three systems do verify equivalence of the lifted code and the original code. The fact that these approaches rely on recovering a high-level IR complicates the kinds of VCs they must generate—all of them make non-trivial use of general purpose SMT solvers—and the process of generating the VCs itself, which is done, e.g., in the case of STNG, by counterexample-guided inductive synthesis [69]. Moreover, these systems lift sequential code—C, C++, Fortran, or Python loop nests—whereas a GPU kernel comprises hundreds or thousands of concurrent threads. Applying lifting in our setting would additionally require modeling shared memory, barrier and warp-level synchronization, and the attendant possibility of data races and deadlocks. Volta does not synthesize a new high-level semantic summary and its equivalence-checking pipeline does not rely on an SMT solver. Instead, Volta operates directly on low-level PTX, taking advantage of known tensor sizes and thread IDs to further simplify the space of expressions it must support. It interprets kernels to obtain symbolic expressions for each output tensor element and checks the immediately implied VCs (expression equalities) using a specialized normalization procedure. Soundness and completeness are guaranteed by virtue of the semantics given in this paper and the Agda-verified confluence theorem. A further advantage of working with PTX is

that it is currently the lowest-level public-facing language for NVIDIA GPUs and, therefore, the ultimate target of open-source compilation pipelines.

8 Conclusion and future work

We have presented Volta, the first equivalence checker for GPU ML kernels. We formally prove that Volta is sound and complete for structured-CTAs. Volta scales to realistic ML benchmarks. In particular, our work is the first formal verification tool which handles programs that use GPU tensor cores. We prove the decidability of identities with expressions over real algebra and exponentiations, a result that may be of independent interest. This paper focuses on verification of programs that synchronize with blocking operations. We leave asynchronous CUDA intrinsics such as `pipeline` and `tma` for future work.

Data-Availability Statement

The kernels we used in our benchmarks and the Agda code that proves Lemma 1 are available via Zenodo [24], and were submitted for artifact evaluation. Our implementation is available at https://github.com/willtunnels/volta for reproducibility.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract HR0011-25-2-0039, as part of the MOCHA program. We would like to thank the following for people for their helpful discussions: Jubi Taneja, Madanlal Musuvathi, Aditya Nori, Ashish Panwar, Aseem Rastogi, Tyler Sorensen, Nikhil Swamy, Srikar Veluvali, Hari VK, Michael Bauer, Manolis Papadakis, and Sean Treichler.

References

[1] AMD. 2025. What is ROCm? https://rocm.docs.amd.com/en/docs-6.4.2/what-is-rocm.html
[2] Anthropic. [n. d.]. *Claude Code*. Agentic coding assistant built on the Claude model. https://www.anthropic.com/claude-code
[3] Jai Arora, Sirui Lu, Devansh Jain, Tianfan Xu, Farzin Houshmand, Phitchaya Mangpo Phothilimthana, Mohsen Lesani, Praveen Narayanan, Karthik Srinivasa Murthy, Rastislav Bodík, Amit Sabne, and Charith Mendis. 2025. TensorRight: Automated Verification of Tensor Graph Rewrites. *Proc. ACM Program. Lang.* 9, POPL (2025).
[4] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. 2022. SMT-Based Translation Validation for Machine Learning Compiler. In *CAV*.
[5] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-Turn RL for Generating CUDA Kernels. In *Proceedings of the Efficient Systems for Foundation Models (ES-FoMo-III) Workshop, ICML 2025*.
[6] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2024. Program Equivalence in the Erlang Actor Model. *Computers* 13, 11 (2024).
[7] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *OOPSLA*, Gary T. Leavens and Matthew B. Dwyer (Eds.).
[8] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit Seshia, and Alvin Cheung. 2024. Verified Code Transpilation with LLMs. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 41394–41424. doi:10.52202/079017-1310
[9] NVIDIA Developer Blog. 2017. Implicit Warp-Synchronous Programming Is Unsafe. NVIDIA Developer Blog. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives
[10] Stefan Blom, Marieke Huisman, and Matej Mihelcic. 2014. Specification and verification of GPGPU programs. *Sci. Comput. Program.* 95 (2014).
[11] Simon Boehm. 2022. How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog. https://siboehm.com/articles/22/CUDA-MMM
[12] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax

[13] Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theoretical Computer Science* 375, 1 (2007), 227–270. Festschrift for John C. Reynolds's 70th birthday. doi:10.1016/j.tcs.2006.12.034
[14] Wentao Chen, Jiace Zhu, Qi Fan, YeHan Ma, and An Zou. 2025. CUDA-LLM: LLMs Can Write Efficient CUDA Kernels. *arXiv preprint arXiv:2506.09092* (2025).
[15] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *PLDI*.
[16] Tiago Cogumbreiro, Julien Lange, Dennis Liew, and Hannah Zicarelli. 2024. Memory access protocols: certified data-race freedom for GPU kernels. *Formal Methods Syst. Des.* 63, 1 (2024).
[17] Tiago Cogumbreiro, Julien Lange, Dennis Liew Zhen Rong, and Hannah Zicarelli. 2021. Checking Data-Race Freedom of GPU Kernels, Compositionally. In *CAV*, Alexandra Silva and K. Rustan M. Leino (Eds.).
[18] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic crosschecking of floating-point and SIMD code. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*.
[19] NVIDIA Corporation. 2025. CUTLASS: CUDA Templates and Python DSLs for High-Performance Linear Algebra. Available from https://github.com/NVIDIA/cutlass.
[20] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *arXiv preprint* arXiv:2307.08691 (2023). https://arxiv.org/abs/2307.08691
[21] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*.
[22] Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *POPL*.
[23] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. 4 pages.
[24] Benjamin Driscoll, Kshitij Dubey, Anjiang Wei, Neeraj Kayal, Rahul Sharma, and Alex Aiken. 2026. *Equivalence Checking of ML GPU Kernels*. doi:10.5281/zenodo.21529230
[25] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. BARRACUDA: binary-level analysis of runtime RAces in CUDA programs. In *PLDI*, Albert Cohen and Martin T. Vechev (Eds.).
[26] Patrice Godefroid and Johannes Kinder. 2010. Proving memory safety of floating-point computations by combining static and dynamic program analysis. In *ISSTA*.
[27] Torbjörn Granlund and the GMP development team. [n. d.]. *GNU MP: The GNU Multiple Precision Arithmetic Library*. https://gmplib.org/
[28] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. 2020. Counterexample-guided correlation algorithm for translation validation. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 221 (2020).
[29] Mark Harris. 2013. Optimizing Parallel Reduction in CUDA. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
[30] Anup Holey, Vineeth Mekkat, and Antonia Zhai. 2013. HAccRG: Hardware-Accelerated Data Race Detection in GPUs. In *Proceedings of the 2013 42nd International Conference on Parallel Processing*.
[31] Aditya K. Kamath, Alvin A. George, and Arkaprava Basu. 2020. ScoRD: A Scoped Race Detector for GPUs. In *ISCA*.
[32] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *SIGPLAN Not.* 51, 6 (June 2016), 711–726. doi:10.1145/2980983.2908117
[33] Kensuke Kojima, Akifumi Imanishi, and Atsushi Igarashi. 2018. Automated Verification of Functional Correctness of Race-Free GPU Programs. *J. Autom. Reason.* 60, 3 (2018).
[34] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *CAV*.
[35] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying bit-manipulations of floating-point. In *PLDI*, Chandra Krintz and Emery D. Berger (Eds.).
[36] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2018. On automatically proving the correctness of math.h implementations. *Proc. ACM Program. Lang.* 2, POPL (2018).
[37] Miriam Leeser, Saoni Mukherjee, Jaideep Ramachandran, and Thomas Wahl. 2014. Make it real: Effective floating-point reasoning via exact arithmetic. In *DATE*.
[38] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU kernels by test amplification. In *PLDI*.
[39] Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *FSE*, Gruia-Catalin Roman and André van der Hoek (Eds.).
[40] Guodong Li and Ganesh Gopalakrishnan. 2012. Parameterized Verification of GPU Kernel Programs. In *IPDPS*.
[41] Guodong Li, Peng Li, Geoffrey Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: concolic verification and test generation for GPUs. In *PPOPP*, J. Ramanujam and P. Sadayappan (Eds.).
[42] Jiatu Li and Mengdi Wu. 2026. Identity Testing for Circuits with Exponentiation Gates. In *ITCS*.
[43] Pengcheng Li, Xiaoyu Hu, Dong Chen, Jacob Brock, Hao Luo, Eddy Z. Zhang, and Chen Ding. 2017. LD: Low-Overhead GPU Race Detection Without Access Monitoring. *ACM Trans. Archit. Code Optim.* 14, 1 (2017).

[44] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2014. Practical Symbolic Race Checking of GPU Programs. In *SC*, Trish Damkroger and Jack J. Dongarra (Eds.).
[45] Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2024. Sound and Partially-Complete Static Analysis of Data-Races in GPU Programs. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024).
[46] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI*.
[47] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *PLDI*, David Grove and Stephen M. Blackburn (Eds.). ACM, 22–32.
[48] Angus Macintyre and Alex Wilkie. 1996. On the Decidability of the Real Exponential Field. In *Kreiseliana*. A K Peters, 441–467.
[49] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. 2023. C2TACO: Lifting Tensor Code to TACO. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Cascais, Portugal) *(GPCE 2023)*. Association for Computing Machinery, New York, NY, USA, 42–56. doi:10.1145/3624007.3624053
[50] John McCarthy and James Painter. 1967. Correctness of a Compiler for Arithmetic Expressions. In *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics, Vol. 19)*. 33–41.
[51] Dragana Milovančević and Viktor Kuncak. 2023. Proving and Disproving Equivalence of Functional Programming Assignments. *Proc. ACM Program. Lang.* 7, PLDI (2023).
[52] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *PLDI*. 83–95.
[53] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation* (Savannah, GA, USA) *(TLDI '09)*. Association for Computing Machinery, New York, NY, USA, 1–2. doi:10.1145/1481861.1481862
[54] NVIDIA. 2024. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/
[55] NVIDIA Corporation. 2017. *NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU*. Technical Report. NVIDIA. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[56] NVIDIA Corporation. 2025. NVIDIA Compute Sanitizer. https://developer.nvidia.com/compute-sanitizer. CUDA correctness and memory checking tool.
[57] NVIDIA Corporation. 2025. *PTX: Parallel Thread Execution ISA*. NVIDIA. Version 9.0. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
[58] Anne Ouyang, Azalia Mirhoseini, and Percy Liang. 2025. Surprisingly Fast AI-Generated Kernels We Didn't Mean to Publish (Yet). https://crfm.stanford.edu/2025/05/28/fast-kernels.html
[59] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.* 50, 6 (2015).
[60] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: a dynamic CUDA race detector. In *PLDI*, Jeffrey S. Foster and Dan Grossman (Eds.).
[61] Phillipe A. Pereira, Higo F. Albuquerque, Hendrio Marques, Isabela da Silva, Celso B. Carvalho, Lucas C. Cordeiro, Vanessa Santos, and Ricardo Ferreira. 2016. Verifying CUDA programs using SMT-based context-bounded model checking. In *Symposium on Applied Computing*, Sascha Ossowski (Ed.). 1648–1653.
[62] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *TACAS*.
[63] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. 2024. Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:28. doi:10.4230/LIPIcs.ECOOP.2024.32
[64] David A. Ramos and Dawson Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *CAV*.
[65] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *ASPLOS*.
[66] Rahul Sharma, Michael Bauer, and Alex Aiken. 2015. Verification of Producer-Consumer Synchronization in GPU Programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[67] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-Driven Equivalence Checking. In *OOPSLA*.
[68] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. 2015. Conditionally Correct Superoptimization. In *OOPSLA*.
[69] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *SIGARCH Comput. Archit. News* 34, 5 (Oct. 2006), 404–415. doi:10.1145/1168919.1168907
[70] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2019. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1 (2019).

[71] Ofer Strichman and Benny Godlin. 2005. *Regression Verification - A Practical Way to Verify Programs.*

[72] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19)*.

[73] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. 2025. TileLang: A Composable Tiled Programming Model for AI Systems. *arXiv preprint* arXiv:2504.17577 (2025). https://arxiv.org/abs/2504.17577

[74] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. 2025. Astra: A Multi-Agent System for GPU Kernel Performance Optimization. *arXiv preprint arXiv:2509.07506* (2025).

[75] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2025. Mirage: a multi-level superoptimizer for tensor programs. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation* (Boston, MA, USA) *(OSDI '25)*. USENIX Association, USA, Article 13, 18 pages.

[76] Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2020. Simulee: detecting CUDA synchronization bugs via memory-access modeling. In *ICSE*, Gregg Rothermel and Doo-Hwan Bae (Eds.).

[77] Jiaqi Yin, Zhan Song, Nicolas Bohm Agostini, Antonino Tumeo, and Cunxi Yu. 2025. HEC: Equivalence Verification Checking for Code Transformation via Equality Saturation. *arXiv* (2025). https://arxiv.org/abs/2506.02290

[78] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: a low-overhead mechanism for detecting data races in GPU programs. In *PPOPP*, Calin Cascaval and Pen-Chung Yew (Eds.).

[79] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2014. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *IEEE Trans. Parallel Distributed Syst.* 25, 1 (2014).