The Next Frontier for AI-Generated Kernels: Correctness

Guido Martínez
Microsoft Research
Redmond, USA
guimartinez@microsoft.com

Tyler Sorensen
Microsoft Research
Redmond, USA
tsorensen@microsoft.com

Abstract

KernelBench was recently proposed as a GPU-programming challenge set for LLMs, measuring their ability to generate correct and efficient GPU kernels for a variety of problems. With the advances of LLMs, most frontier models can now pass most of KernelBench with ease, and the performance (and corresponding complexity) of the generated kernels is increasing. In contrast, the correctness aspect of these kernels has gotten relatively little attention. KernelBench uses a fixed test suite to accept a kernel as valid. Is this perhaps enough, given the usual regularity of GPU kernels? Do buggy kernels make it through this test suite?

We claim that testing is hopelessly inadequate for such tasks, finding serious problems in KernelBench's test suite, with correct kernels being rejected and buggy kernels being accepted. While some of these issues are easily solvable, others are fundamental and unavoidable, particularly if one considers an adversarial LLM. We argue that AI-generated kernel solutions should come with formal proofs of correctness, and show that this is now practical. Using Kuiper, a recent verified GPU programming framework, we present verified implementations for all 100 KernelBench Level 1 tasks, written almost entirely by LLM coding agents.

CCS Concepts: • **General and reference → Verification**; • **Computing methodologies → Graphics processors**; • **Software and its engineering → Software verification and validation**; • **Information systems → Language models**.

Keywords: GPU programming, dependent types, separation logic, agentic verification

ACM Reference Format:
Guido Martínez and Tyler Sorensen. 2026. The Next Frontier for AI-Generated Kernels: Correctness. In *Proceedings of the 2026 ACM SIGPLAN International Workshop on Principles of Agentic Engineering (PAgE '26), June 15–19, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3819802.3820580

PAgE '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2717-7/2026/06
https://doi.org/10.1145/3819802.3820580

1 Introduction

GPUs have become critical infrastructure for modern machine learning. Training and serving large language models demands massive compute, and kernel-level efficiency directly impacts throughput, latency, and hence cost. Yet writing correct GPU kernels remains a serious challenge: optimized kernels involve intricate indexing, tiling, shared memory management, and synchronization across thousands of threads, where a single indexing error or data race can produce wrong answers, nondeterminism, or input-dependent failures that ordinary testing can miss. As GPUs become shared infrastructure, such failures are matters of system integrity, and may intersect with emerging GPU security concerns such as memory exploitation, cache side channels, and cache-hint attacks [11, 32, 33].

At the same time, LLMs are writing a growing share of the world's software, and GPU kernels are no exception. KernelBench [24] proposed a suite of 250 PyTorch ML workloads as challenges for AI-generated CUDA kernels. Recent LLM-driven approaches can now pass a large majority of these tasks: for example, an NVIDIA report reached 100% of KernelBench Level 1 shortly after the benchmark's release [5]. The focus of the benchmark has shifted from *"can LLMs write kernels?"* to *"how fast are the kernels that LLMs write?"*. This progress is encouraging, but the correctness bar in KernelBench is a test suite: a generated kernel is deemed correct if it produces outputs that match the reference PyTorch implementation on a handful of random inputs. Usually, these inputs do not even attempt to cover the whole input space (e.g. the first challenge is generating a simple square matrix multiplication kernel, but the test suite only checks square matrices of size 4096).

Clearly, such a simple test is insufficient to guarantee correctness. Making things worse, LLMs are also prone to avoid solving the problem entirely. Indeed, the KernelBench authors themselves have documented pervasive *reward hacking* [10]: LLMs find creative ways to satisfy the test suite without doing real work, such as calling high-level operators instead of writing CUDA, generating no-op kernels that reuse memory from the reference implementation, or manipulating timing synchronization to fake speedups. Even when known exploits are patched, new ones emerge. The v0.1 update to KernelBench introduced randomized testing and improved problem sizing, yet the authors acknowledge

that "designing robust, adversary-resistant reward signals remains an open problem" and recommend manual inspection of high-scoring outputs.

Recent work has begun to identify and patch similar weaknesses. Robust-kbench [17] identifies contaminated tasks, fixed input configurations, and artificial speedups from benchmark exploits. Kevin [2] reports reward hacking during multi-turn reinforcement learning for CUDA kernels, including PyTorch fallbacks and reuse of output tensors from the reference run, and responds with rule-based checks, larger problem sizes, and harness changes that run the tested kernel before the reference. CUDA-L1 [19] similarly reports KernelBench reward hacking, including artificial speedups from extra CUDA streams, and strengthens validation by checking equivalence on 1000 random inputs, materializing lazy outputs before timing, synchronizing streams, and rechecking suspicious speedups. These responses are pragmatic: add more diverse testing conditions, stronger timing and precision checks, and LLM- or rule-based filters. Such mitigations are useful, but they remain empirical and model-mediated; they do not provide the same guarantee as a machine-checked proof.

We believe that testing, no matter how sophisticated, is fundamentally inadequate to prevent buggy kernels and reward hacking. The input space and attack surface are simply too large: there are too many parameters with intricate constraints between them (e.g. matrix/tile/subtile dimensions, and their divisibility requirements), and any gap between the test oracle and the true specification is an opportunity for exploitation. In § 3, we reinforce this point with concrete oracle failures: buggy Softmax kernels that pass KernelBench, and generated (verified) matrix multiplications that fail for tolerance and precision-contract reasons rather than algorithmic ones. A safer language with machine-checked correctness guarantees is needed.

To address this gap, we turn to formal verification. Recent work has begun to show that formal methods are practical for AI-generated kernels. Volta [8] checks PTX-level equivalence for structured ML GPU kernels, including kernels generated by LLMs, while also detecting races and deadlocks in its supported fragment. ProofWright [4] keeps CUDA as the target language and uses LLM agents to generate VerCors annotations and Rocq proofs, establishing memory safety and data-race freedom for 74% of KernelBench L1 kernels and semantic equivalence for a smaller class of element-wise kernels. Together, these efforts strongly support the case that formal methods are becoming practical for AI-generated kernels.

Our work explores a complementary design point, which is even more centered on verification: rather than generating low-level GPU code first and then checking or annotating it after the fact, we make the verified language itself the target of generation. In Kuiper [1, 20, 21], memory safety, data race freedom, and functional-correctness obligations are expressed in the source program and checked before extraction to CUDA. This changes the role of verification from a post-hoc audit of low-level code into a constraint on the agent's search space: candidate kernels must carry the invariants needed for safety and correctness as they are written. Kuiper kernels are also written at a higher level of abstraction than CUDA, and entire families of kernels (across all metaparameters, types, operations, etc) can be verified at once, increasing reuse.

Can LLMs write verified Kuiper code? We show that the answer is yes. Using LLM-based coding agents in an iterative verification loop, we have produced solutions for all of KernelBench's Level 1. All solutions pass the KernelBench tests, and are verified to be safe and functionally correct. The resulting artifact is therefore not merely a CUDA kernel that has passed more tests, but a proof-carrying GPU program whose checked obligations justify the CUDA code produced by extraction. This distinction is visible in the outcomes: ProofWright scales safety checking broadly, with memory safety and data-race freedom for 74% of KernelBench L1, but establishes semantic equivalence for 14%; our Kuiper corpus contains runnable verified implementations for all 100 Level 1 tasks, each carrying a precise functional-correctness postcondition. Functional correctness properties are either *exact* or *approximate*. The latter do not guarantee any concrete error bounds on the output, but do guarantee that the result is algebraically sound. The full results are in Appendix A.

In summary, we make the following contributions:

- Evidence that KernelBench test suites are insufficient for establishing kernel correctness, with both false positives and false negatives (§ 3).
- An agentic workflow for generating verified GPU kernels in Kuiper, combining LLM code generation with formal verification feedback (§ 4).
- A verified Kuiper solution for all 100 KernelBench Level 1 tasks (§ 5).

2 Background

2.1 Brief Primer on GPU Programming

GPUs are massively parallel accelerators (or *coprocessors*) that execute thousands of threads concurrently. Work on the GPU (the *device*) must be explicitly scheduled by the CPU (the *host*), by launching a *kernel*. Kernels are launched in a *grid* of some number of *blocks*, each containing the same number of *threads* (in CUDA, up to 1024 per block). Running kernels can only access GPU memory, which is (usually) separate from CPU memory. Data must be explicitly copied between host and device by the host. Figure 1 shows a simple CUDA kernel for matrix multiplication, along with its host-side launch code. Each thread first computes its global ID in the grid, using it to decide which cell of the output matrix C to fill. The thread then computes the dot product of the

```
__global__
void ker(int N, int K, float *A, float *B, float *C)
{
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  int i = gid / N;
  int j = gid % N;
  float acc = 0.0;
  for (int l = 0; l < K; l++)
    acc += A[i * K + l] * B[l * N + j];
  C[i * N + j] = acc;
}
void matmul(int M, int N, int K,
            float* A, float* B, float* C) {
  int nblks = M * N / 1024;
  ker<<<nblks, 1024>>>(N, K, A, B, C);
}
```

Figure 1. Naïve matrix multiplication in CUDA

corresponding row of A and column of B, storing the result in that cell. The `matmul` function schedules the kernel with a grid of blocks, each containing 1024 threads.

In this case, the kernel is extremely simple in that there is no coordination between threads or use of any advanced features. Yet, its correctness depends on some non-trivial properties:

- The matrices (`A`, `B`, `C`) already live in GPU memory. (CUDA uses plain pointer types for both host and device memory.)
- Their dimensions are accurate ($M \times K$, $K \times N$, and $M \times N$, respectively).
- They are laid out in row-major order, with no padding or striding.
- $M \times N$ is a multiple of 1024, so that the grid covers the entire output matrix.

As kernels become increasingly complex, these properties grow quickly in complexity, and the potential for subtle bugs grows with them.

2.2 KernelBench

KernelBench [24] is an open-source benchmark for evaluating the ability of LLMs to generate fast and correct GPU kernels. It consists of 250 carefully selected PyTorch ML workloads, organized into levels of increasing complexity:

- **Level 1:** Single-kernel operators (convolutions, matrix multiplications, normalizations, etc.).
- **Level 2:** Simple fusion patterns (composed or fused operations).
- **Level 3:** Multi-stage computational blocks.

For each task, an LLM is given a reference PyTorch implementation and must produce a CUDA kernel that computes the same function. The primary evaluation metric is $fast_p$: the fraction of generated kernels that are both functionally correct and achieve a speedup of at least $p\times$ over the PyTorch baseline. Correctness is checked by running the generated kernel on a set of reference inputs and comparing outputs to the PyTorch implementation within a tolerance.

As of early 2026, a variety of approaches can pass a large majority of Level 1 tasks, and the community focus has shifted to optimizing performance. Recent work has strengthened testing-based approaches to correctness with more robust harnesses, larger input sets, and empirical filters [2, 17, 19]. A small number of recent efforts instead turn to formal methods: Volta checks PTX-level equivalence for structured ML GPU kernels, including AI-generated kernels, while ProofWright verifies generated CUDA kernels using agentic annotation and proof generation [4, 8]. Most correctness criteria, however, still reduce to passing test suites.

2.3 Kuiper

Kuiper [20] is a framework for safe and verified GPU programming, embedded as an extensible library within the F* dependently typed programming language [29], in particular in its separation-logic based sublanguage Pulse [9]. F*/Pulse provide dependent types, SMT automation, and separation logic for reasoning about mutable state, while Kuiper adds GPU-specific abstractions and reasoning principles, and an extraction pipeline into CUDA. Kuiper then provides:

- **Dependent types** for encoding precise specifications: matrix dimensions, grid configurations, and memory layouts are all tracked in types, ruling out entire classes of bugs at compile time.
- **Separation logic** for reasoning about the GPU memory hierarchy, shared memory, synchronization barriers, and host/device interactions. Kuiper introduces *located resources* and a novel *universal separating conjunction* ($\forall^+$) for modular reasoning about massively parallel programs.
- **Kernel descriptions** that lift per-thread specifications to global end-to-end correctness proofs for entire kernels, relating the per-thread programming model of CUDA to a whole-kernel functional specification.
- **Safe interfaces** to low-level and performance-critical CUDA features like shared memory, barriers, tensor cores, and vectorized memory access.
- **Layout polymorphism** to allow generic verified implementations of algorithms, which can be instantiated for free to different memory layouts, types, tiling parameters, etc.

Kuiper guarantees memory safety and data race freedom by design. When the specifications are precise enough, it can also guarantee full functional correctness, at some extra effort. Kuiper kernels are extracted to CUDA via Karamel [26], with no runtime overhead introduced by the verification framework. The authors show several flavors of verified GEMMs with no runtime overhead compared to their hand-written counterparts, some of them being competitive with cuBLAS.

```
fn kf {| scalar et |}
  (bid : szlt nblk) (tid : szlt nthr)
  (a : array2 et m k lA)
  (b : array2 et k n lB)
  (c : array2 et m n lC)
  preserves a ↦ Frac f va ⋆ b ↦ Frac g vb
  requires  c[row, col] ↦ _
  ensures   c[row, col] ↦ mm_dotprod va vb row col
{
  let gid = 1024sz * bid + tid;
  let row, col = gid / n, gid % n;
  let mut sum : et = zero;
  for l in [0,k)
    invariant sum ↦ mm_part va vb row col !l
    sum ← !sum + a[row, l] * b[l, col];
  c[row, col] ← sum;
}
let kdesc a b c = { kf=kf; nthr = m*n; ... }
fn matmul (a b c : gpu_matrix et 1024 1024)
  preserves a ↦ va ⋆ b ↦ vb
  requires c ↦ _
  ensures  c ↦ va × vb
{ launch kdesc; }
```

Figure 2. Naïve matrix multiplication in Kuiper, slightly stylized.

Figure 2 shows the same naïve matrix multiplication kernel from Figure 1, but written in Kuiper. The kernel function is annotated with pre- and postconditions describing its behavior. In this case, the function obtains a "fraction" of permission to both `a` and `b` (essentially read-only access), and full ownership of a single cell of `c` (the output matrix). The postcondition states that the cell of `c` is updated to the correct dot product of the corresponding row of `a` and column of `b`. The variables `va`, `vb` (implicitly quantified) are ghost variables that represent the values of the arrays before the kernel runs. The element type of the matrices (`et`) is a type parameter, making the kernel polymorphic over the element type. The layouts of the matrices (`lA`, `lB`, `lC`) are also parameters, making the kernel also polymorphic over them. The kernel function is then packaged into a "kernel description" that specifies the number of threads to launch and what the high-level pre- and postconditions for the entire kernel are, in this case that of `c` is updated to the matrix product of `a` and `b`. The description includes proofs that the high-level precondition implies the preconditions for each thread in the grid, and vice-versa for the postcondition. We omit these for brevity. This Kuiper program can later be extracted to CUDA, obtaining code similar to the one in Figure 1, though importantly one can vary the type and layouts without redoing any proofs.

Crucially, Kuiper is designed as a general-purpose, extensible language: GPU features (blocks, threads, tensor cores, barriers, shared memory, vectorized access) are modeled as library constructs rather than being hardwired into the language, making the framework extensible as GPU hardware evolves. As long as the desired primitives are modelled, there is no restriction on the code one can verify with Kuiper.

Klas. Building on Kuiper, we are also developing Klas ("Kuiper Linear Algebra Subroutines"), a verified BLAS library that extracts to plain CUDA code. Tackling the breadth of KernelBench tasks has led us to define common abstractions and reusable patterns—for elementwise maps, reductions, scans, broadcasting, and fused operations—that make writing and verifying new kernels easier than doing so monolithically.

3 Tests Are Insufficient

KernelBench's correctness criterion is whether a generated kernel produces outputs matching the reference implementation on a fixed set of test inputs. It is clear to almost anyone familiar with testing that it is no guarantee of correctness, but how much so in this case? Does the testing only fail to capture intricate corner case failures, or do obviously wrong kernels also slip through? Is GPU code perhaps better suited for testing? The question takes more relevance given the sometimes adversarial nature of LLMs: they are sometimes happy to declare victory after finding some "dishonest" way of passing the test suite; closely related failure modes are documented as reward hacking, benchmark exploits, or cheating kernels in KernelBench and recent follow-on work [2, 10, 17, 19].

3.1 False Positives

To gauge the bug-finding quality of KernelBench's test suite, we performed an audit of its oracle. The audit combined several checks: reviewing generated kernels, replaying candidate failures under alternative inputs and tools, and, in some cases, starting from simple kernels whose behavior we validated by manual inspection and then introducing mutations such as:

- **Off-by-one errors** in indexing or loop bounds.
- **Boundary condition errors**: incorrect handling of matrix edges when dimensions are not multiples of the block size.
- **Missing or misplaced synchronization barriers.**
- **Incorrect reductions**: e.g., initializing an accumulator to a wrong value, or reducing over a wrong dimension.
- **Race conditions**: removing atomic operations or writing to shared locations without proper synchronization.

The full outcome of this audit will be published in future work. Here we focus on the case we found particularly problematic for KernelBench's oracle: Softmax, KernelBench Level 1 problem 23. Softmax is essentially a normalized exponential function, where a vector x is transformed into a vector y with $y_i = e^{x_i} / \sum_j e^{x_j}$. The challenge is to take a

matrix and apply Softmax to each of its rows independently, producing a matrix of the same shape. For this task, KernelBench checks a single input configuration: one uniform-distribution draw for a matrix of 4096 rows (the "batch size") and 393,216 columns. It does not vary the shape, distribution, or random seed during the check. Many of the correct output values for this configuration are around 2.5×10^{-6}, far below the allowed fp32 tolerance threshold defined in KernelBench ($atol = rtol = 10^{-4}$).

KernelBench accepts several bad CUDA kernels as solutions. A racy kernel with a removed shared-memory barrier, a kernel whose final write loop skips half the output positions, and a Triton kernel that stores every value one element too far to the right all pass the test suite. These kernels are legitimately incorrect, not merely different because of round-off. A replay over a smaller matrix shows the stride and off-by-one kernels producing stable but wrong outputs with maximum absolute errors of 7.7×10^{-4} and 6.5×10^{-4} against PyTorch, respectively. These errors can be made larger by varying the input distribution or with smaller matrices.

Dynamic and static race detectors can help expose some of these failures [3, 7, 13, 23, 25, 28]. NVIDIA's Compute Sanitizer Racecheck is a run-time shared-memory hazard detector; in our audit, it flags the racy Softmax kernel even though that kernel passes KernelBench's tests. However, such tools are not a substitute for proof. In our own recent use, we observed Racecheck reporting races on high-performing NVIDIA matrix-multiplication kernels invoked through PyTorch, suggesting that it can produce false positives on optimized code. Moreover, Racecheck currently only supports detecting races in on-chip shared memory [23]; races through global memory, or higher-level violations of the intended computation, remain outside its scope. The original benchmark shape masks these semantic bugs because the expected outputs are so small relative to the absolute tolerance.

3.2 False Negatives

We also observe the converse failure mode: correct kernels that fail KernelBench's test suite. Perhaps surprisingly, the kernel from Figure 1—a simple, textbook matrix multiplication—fails a test.

KernelBench's challenge 4 is a matrix-vector multiplication. A valid (though perhaps not ideal) solution is to first interpret the vector as a matrix of width 1, and then performing a usual matmul to obtain a vector. Passing that solution to KernelBench's checker produces the following output:

```
FAIL, max_diff=['53.500000']
```

How can this be? The answer is numerical error. KernelBench's reference solution for this problem (`torch.matmul`) does not add the products in left-to-right order as the naive kernel does, but instead uses an unspecified order of operations. Usually, performing this reduction in tree-like fashion, or first adding tiles, is a faster way of computing the sum, so there are real reasons to choose a different order of operations. However, this should not mean that the naive kernel is functionally incorrect. In fact, it is not a priori clear which of the two kernels is closer to the real result.

The size of the error here is striking—a maximum absolute difference of 53.5 on outputs of order one—but it is entirely an artifact of summation order, not of the kernel computing the wrong function. Replacing the naive left-to-right accumulation with a Kahan compensated sum [14], without otherwise changing the kernel, drops the maximum difference against `torch.matmul` to roughly 0.03. The underlying algorithm is the same; only the rounding behaviour of the reduction has changed.

This pattern is not specific to our hand-written example. In a small replay of generated kernels for KernelBench level-1 problems, five Triton matrix-multiplication kernels (problems 1, 2, 7, 8, and 9) all fail the default 10^{-4} checker when their reduction uses bare `tl.dot(a, b)`, but pass at 10^{-3}; their minimum `torch.allclose` tolerances against the PyTorch reference fall between 7.28×10^{-4} and 9.15×10^{-4}. The cause is the default precision contract of Triton's matrix-multiply primitive: on Ampere and newer NVIDIA GPUs, `tl.dot` on fp32 inputs is by default lowered to a TensorFloat-32 (TF32) tensor-core operation, which rounds each multiplicand's mantissa from 23 bits to 10 bits before the multiply and accumulates the products in fp32. Passing `input_precision="ieee"` to `tl.dot` instead forces a full IEEE-754 fp32 multiply with no mantissa truncation, at a significant throughput cost. With that single change—and no other modification to the kernels—all five generated matmuls pass KernelBench's default checker. The kernels are not buggier in one configuration than the other; they are simply implementing a different, but equally defensible, numerical contract than the reference.

Together with the Softmax false positives above, these examples show that KernelBench's tolerance-based oracle is neither a complete bug detector nor a precise correctness specification: it accepts real semantic bugs when the expected outputs are small relative to the tolerance, and it rejects algorithmically plausible kernels when the backend's accumulation order or precision differs from the reference's.

3.3 Reward Hacking

Setting aside "accidental" bugs, can clearly wrong or adversarial kernels make it through the test infrastructure? The answer is yes. We've found several such cases.

Recall the discussion on Softmax above and the outputs being too small. They are so small that, in fact, simply memsetting the output to zero passes the test suite. This is actually mathematically provable. First note that the input vector (of length $N = 393{,}216$) is uniform on $[0, 1)$, so exponentiating it gives a log-uniform distribution on $[1, e)$. The sum of this vector is at least N. Normalizing the vector divides everything by this sum, obtaining a maximum possible value

of at most $e/N \approx 7 \times 10^{-6}$, way below the 10^{-4} tolerance threshold. For larger input values (e.g., in $[0, 10)$), or smaller array sizes, the result cannot be faked with zeros.

Another concern, not to be ignored with the opacity of LLMs, is them encoding some kind of trojan in the kernel. It is a trivial exercise to write a kernel that does Softmax most of the time, but also check for particular hidden values in the input in order to activate some other arbitrary code. Testing is wholly inadequate to catch such trojans due to the vastness of the input space. There is also no guard against the generated code modifying the inputs.

The KernelBench v0.1 update [10] already documents several categories of reward hacking that LLMs employ to game the benchmark:

- **Not generating real CUDA code:** the model calls high-level operators like `torch` or cuBLAS, which pass correctness checks but sidestep the challenge entirely.
- **No-op kernels with memory reuse:** the generated code performs no computation, but because the evaluation reads output from the same memory region as the reference, it appears correct.
- **Timing manipulation:** incorrect synchronization points or mismatched CUDA streams create artificially short runtimes.
- **Dead code:** the model writes a CUDA kernel but never calls it, instead falling back to a simpler implementation.

The KernelBench authors note that "even when we patch known exploits, new ones emerge, often even harder to detect," and conclude that "designing robust, adversary-resistant reward signals remains an open problem." The v0.1 update introduces randomized testing and improved problem sizing, but these are mitigations, not solutions. We believe that testing—no matter how sophisticated—cannot close the gap between the test oracle and the true specification. Every such gap is an opportunity for reward hacking, and LLMs are remarkably effective at finding them.

Our findings described above are all performed over v0.1.

Improving Testing Coverage. Some of the problems above can be mitigated by improving the test coverage. Every kernel should be tested with different dimensions, random seeds, distributions (uniform, power, etc). Ideally, kernels should also not be fixed to a particular type, layout, or metaparameter like tile size, and the testing framework should vary those too. Clearly, this is a large space to test, and it's unclear one can have an effective set of tests that find bugs, do not raise false positives, and do not take enormous computational resources to complete. Still, it remains unlikely one can detect adversarial kernels with embedded trojans through testing alone.

3.4 The Case for a Safer Language

We argue for a fundamentally different approach: ascertain correctness statically, with guarantees that hold over the kernel's entire input space, rather than dynamically over a handful of test inputs. In other words, we want formally verified solutions, which are guaranteed to be free of memory errors and data races, and to be functionally correct. When an LLM produces a Kuiper kernel that the verifier accepts, we have a machine-checked guarantee that the kernel meets its specification.

This approach is in the spirit of proof-carrying code [22], applied here to full functional correctness rather than only the memory/type-safety policies typical of early PCC deployments. Moreover, in Kuiper, certain classes of bugs—notably memory errors—are ruled out by the type system itself, requiring no per-kernel proof obligation.

This shifts the trust burden from the implementation to the specification—a smaller, more declarative, and reusable artifact, but still a nontrivial one to get right. An LLM could still "hack" the verifier in a different way: by inserting unsafe `admit()` statements (which discharge any proof obligation), or by writing a spec that does not actually capture the intended computation. This is a real concern, and is part of the more general problem of *intent formalization* [16]: ensuring that formal specifications faithfully capture what the user actually wants. Two mitigations apply in our setting. Syntactic scans for `admit`/`assume` occurrences (as Kuiper itself provides) catch the most naive attempts, but the deeper defense is that specifications are short, declarative, and reusable across implementations, making them amenable to one-time human review. We discuss this further in § 7.

Beyond the specification, the trusted computing base of a Kuiper kernel includes the verifier itself (F* and its SMT backend), the extraction pipeline from Pulse down to GPU code, and the underlying GPU compiler and hardware model. Bugs in any of these could in principle invalidate a verified kernel's guarantees, and tool bugs in deep verification stacks are not hypothetical—SMT solvers, extraction passes, and proof checkers have all had soundness bugs in their histories. The crucial difference from a test-based oracle is that this TCB is fixed, shared across every kernel, and subject to sustained scrutiny by the verification community, rather than re-incurred per benchmark in the form of a tolerance threshold or a hand-written reference implementation.

4 Agentic Verified Kernel Generation

Given that Kuiper can provide strong correctness guarantees for GPU kernels, the natural question is whether LLM-based coding agents can produce verified Kuiper code. We describe our workflow for doing so, and report preliminary feasibility results in § 5.

4.1 Workflow Overview

Our fully-agentic workflow proceeds in two phases for each KernelBench task, reflecting the insight that functional verification is significantly more expensive than implementation and testing:

Phase 1: Implementation and testing.

1. **Task ingestion.** The agent receives the KernelBench task: a PyTorch reference implementation defining the desired computation (e.g., a matrix multiplication).
2. **Implementation.** The agent writes a Kuiper kernel implementation. At this stage, the code is type-checked—which, absent unsafe escape hatches, ensures memory safety and data race freedom—but *not* functional correctness. The agent is allowed to use unsafe features like `admit()` to discharge proof obligations as scaffolding during this phase, but must remove them before proceeding to Phase 2; the syntactic-scan defense from § 3.4 ensures none survives into the verified output. The agent focuses on getting the algorithm right without the overhead of writing full specifications and proofs.
3. **Testing.** The agent extracts the kernel to CUDA and runs it against the KernelBench test suite. This provides a fast, cheap signal: if the kernel fails tests, the agent iterates on the implementation before investing in verification.

Phase 2: Functional verification. Once the kernel passes tests, the agent proceeds to the costly step of formal verification:

4. **Specification.** The agent writes a functional correctness specification: a mathematical predicate relating the kernel output to the reference computation (e.g., matrix multiplication as a sum of products), along with pre- and postconditions in Pulse's separation logic. We prompt the agents to attempt to reuse existing specifications in the Kuiper and Klas codebases, rather than writing new ones from scratch. This reduces human review effort and increases the likelihood of correctness.
5. **Proof.** The agent annotates the implementation with loop invariants, ghost code, and lemma invocations needed for the F*/Pulse verifier to discharge the proof obligations.
6. **Verification loop.** The verifier checks the annotated code. If verification fails, the verifier's error messages—which point to specific undischarged proof obligations—are fed back to the agent, which revises the proof or implementation. This loop continues until verification succeeds or a retry budget is exhausted.

This two-phase design is pragmatic: Kuiper's type system (when unsafe features are not used) already guarantees memory safety and data race freedom in Phase 1, so the kernel is already safer than unverified CUDA. Phase 2 then adds the strongest guarantee—full functional correctness—but only after a cheap testing pass has established that the implementation is likely correct. Agents can make significant progress by following these instructions, sometimes completing tasks only with feedback from the verifier, and without human intervention. For more difficult kernels (or where some features are missing), human guidance may still be necessary.

4.2 Design Choices

Several aspects of this workflow merit discussion:

Specifications are key. A major advantage of verification over testing is that the specification makes the correctness criterion explicit and precise. For KernelBench tasks, the specification is typically a mathematical definition of the desired computation (e.g., matrix multiplication as a sum of products). Writing these specifications is generally straightforward for the agent, since the PyTorch reference already contains a clear computational description.

Verification feedback as a signal. The F* verifier provides rich, localized error messages when verification fails: it points to the specific proof obligation that could not be discharged, along with the relevant context. This is qualitatively different from a test failure (which only says "wrong answer") and allows the agent to make targeted fixes. We find that agents can effectively use this feedback to iterate toward a correct solution. This also makes sure every possible input and corner case is covered, instead of relying on testing to catch them.

Proof effort. Writing verified Kuiper code requires more than just the kernel implementation: the agent must also provide loop invariants, ghost annotations, and sometimes prove additional lemmas and invoke them. This is sometimes significant additional effort compared to writing plain CUDA. Breaking the task into two phases—first getting a working implementation, then adding the specification and proof—helps manage this complexity. The agent can focus on getting the algorithm right first, and only then invest in the more expensive verification step.

5 Preliminary Results

We applied the agentic workflow of § 4 to all 100 challenges of KernelBench Level 1, solving them all with verified Kuiper kernels that pass KernelBench's own correctness harness within its native fp32 tolerance ($atol = rtol = 10^{-4}$). Every kernel carries a functional specification relating its output to the mathematical ideal of the operation; we classify these by precision. For 14 challenges the specification is *fully precise*: the postcondition guarantees exact floating-point equality (e.g. ReLU, clamps, and max/min reductions). Such a specification is very strong, and can usually only be achieved for simple operations. The remaining 86 solutions are *approximate*. Kuiper provides an idealized real reasoning for floating-point arithmetic, a relation denoted '$\approx$', that ignores numerical errors but allows proving algebraic correctness. All GEMMs, convolutions, softmax, normalizations, reductions, and losses use this approach. There are small, unverified CPU bridges to invoke the Kuiper kernels in the

```
fn mse_loss (n : szp)
 (predictions targets : array1 f32 ..n..)
 (rp rt : erased (lseq real n)) (* idealized contents *)
 preserves cpu ★ on gpu_loc (targets ↦ Frac 'f_b st)
 requires on gpu_loc (predictions ↦ sp)
 requires pure (sp ≈ rp ∧ st ≈ rt)
 returns res : f32
 ensures ∃★ sp'.
  on gpu_loc (predictions ↦ sp') ★ (* predictions is modified *)
  pure (res ≈ real_mse rp rt) (* result is correct approximation *)
{
 Map.map_gpu2 mse_step n predictions targets;
 let s = HReduce.reduce id id 1024sz n predictions vr;
 return div s (of_int n);
}
```

Figure 3. Verified Kuiper kernel for KernelBench L1 #94 (Mean Squared Error), eliding some annotations. The body is a host orchestrator chaining two verified GPU primitives and doing a final division. The postcondition states that the result is a floating-point value that correctly approximates the real-arithmetic mean-squared-error of the input sequences.

challenge solutions, but all of them are extremely simple. All GPU computation runs through verified Kuiper code and the bridges contain no algorithmic logic. Appendix A gives the per-challenge breakdown.

Methodology. The work was carried out in long-running sessions through the GitHub Copilot CLI driving Claude (Sonnet 4.6 and Opus 4.7), with the human author acting as architect, reviewer, and gatekeeper. Each challenge was developed in a dedicated subdirectory containing an F*/Pulse source module, a thin C++/CUDA bridge, and a `run.sh` driver that invokes KernelBench's original PyTorch test harness. The agents were instructed to keep the bridge minimal, i.e. free of any non-trivial computations, and this was enforced via a manual review process. Challenges were attempted in groups by their primitive dependency (e.g., a `WindowReduce1D` cluster covering the pooling family, a `RowBroadcast` cluster covering softmax/swish, and a convolution family including dilated, depthwise, and transposed variants) and effort was spread across clusters using a fleet of background Copilot agents, with their work merged onto an integration branch reviewed challenge-by-challenge. This makes sure that similar (but different) challenges do not need to start from scratch. A first agent can generate a verified, highly-polymorphic, reusable component that is then easily applied to solve several challenges.

Human review. Once a kernel passes the verifier and is admit-free, it still gets a human review for three main reasons. First, the agent may have written a weak specification, though we find that this does not happen often if the instructions clearly state that verification is a necessary property. Second, the agent may have written a correct but inefficient kernel, and a human can suggest optimizations or different ways of performing the task altogether. Most of the kernels we have generated are not as efficient as they could be, but we do rule out obvious inefficiencies. Third, there is a matter of "taste" in the generated code and how well it matches the existing libraries and other kernels. The human reviewer can suggest stylistic changes to make the code more readable and maintainable, and suggest places where to reuse existing code or generalize the solution, e.g. by using Kuiper's layout polymorphism.

Reusable infrastructure. Beyond the challenges themselves, the effort produced a substantial body of reusable verified infrastructure inside Kuiper: new layout-polymorphic array primitives; generic block-level reductions; a polymorphic 1-D windowed reduction that subsumes the entire pooling cluster ("`WindowReduce1D`"); per-row generic matrix operations (broadcast, scale, softmax, logsoftmax) and a convolution family including dilated, depthwise, and transposed variants. Several upstream Kuiper improvements were driven by the experiment: a `floating` typeclass with polymorphic comparisons, Kahan summation (motivated by the matmul/-matvec failure discussed in § 3.2), and small fixes to the extraction pipeline.

5.1 Compositional Pipelines

A notable feature of the generated corpus is how compositional the verified kernels turned out to be. For example, the four loss kernels we verified end-to-end—MSE, Huber, KL-divergence, and Hinge (KernelBench L1 #94/96/98/100)—are all instances of the same three-stage host pipeline:

$$\underbrace{\texttt{map\_gpu2}}_{\text{step}} \rightarrow \underbrace{\texttt{HReduce}}_{\text{tree-reduce to host}} \rightarrow \underbrace{\texttt{post}}_{\text{final processing}}$$

where each stage is an *already-verified* Kuiper primitive parameterised over a per-element function (e.g. `mse_step` $a\,b = (a-b)^2$) or a simple final adjustment (e.g. a division done to normalize the value). Figure 3 sketches the resulting kernel for MSE. Once the agent had completed the first such pipeline (MSE) and the corresponding spec idiom, the remaining three losses fell out as ~60-line variants that differ only in the elementwise step and in a small lemma bridging a hand-specialised `f32` step function to its scalar-polymorphic counterpart in the spec module (see § 5.3).

This pattern, i.e. thin host orchestrators over a small library of verified, polymorphic primitives, is what made the agentic workflow tractable at all. A monolithic kernel, even for an operator as simple as MSE, would force the agent to reason about block/thread layouts, shared memory, and tree-reduction invariants *simultaneously with* the functional postcondition, and repeat this work in each challenge. Routing through `map_gpu2` and `HReduce` discharges those concerns

once and for all in the library, in a highly-polymorphic manner, and leaves the agent to prove only the small algebraic bridge between the hand-written `f32` step and the polymorphic spec. Substantial work was required to grow the Kuiper standard library (and our in-progress *Klas* BLAS layer over it) with the abstractions needed to make this style of kernel writing ergonomic; that engineering (with human input) is a precondition of, but not a contribution of, this paper.

5.2 Floating-Point Reductions: The ≈ Relation

GPU tree-reductions are non-deterministic in their summation order (it depends on block size, occupancy, and atomic-update interleaving), and floating-point addition is not associative. A spec that fixes a particular tree shape would be either unverifiable against a real GPU implementation or insufficient as a correctness statement. Kuiper resolves this with an *approximation relation* ≈ between a floating point value and an idealised real-arithmetic value. All four generated loss specs (and every reduction in the corpus) only guarantee appoximate results, e.g. as in `ensures pure (res ≈ ...)`, essentially stating that the kernel's output is *some* approximation of the right result. The agent learnt this idiom quickly and reused it across every reduction-bearing kernel.

A concrete benefit of this calculus showed up on challenge #4 (irregular matmul). The naive left-fold accumulator failed KernelBench's fp32 tolerance on randomized inputs at $K = 2^{20}$, exactly the failure mode discussed in § 3.2. Switching the reduction to Kahan compensated summation restored the tolerance margin while leaving the ≈-level specification unchanged: both the naive and the Kahan reductions are correct relative to the real-arithmetic ideal. Notably, an attempt to reproduce the same tolerance failure on challenge #6 (smaller K) was unsuccessful, because the random input distribution did not happen to exercise the worst case—a tidy illustration of how brittle benchmark-style tolerance testing is as a safety net for floating-point code, and how the verified-spec approach avoids depending on the test inputs at all. That said, Kuiper's approximation relation is not suited for proving concrete error bounds, which can be desirable. This is interesting future research.

5.3 Challenges and Lessons Learned

Several recurring obstacles surfaced during the engagement.

Reward hacking by the agent itself. Early on, the agents repeatedly produced "phony" solutions: Python wrappers that simply re-invoked the PyTorch reference, hand-written CUDA bridges that bypassed the verified kernel, and custom Python correctness drivers with hand-tuned tolerances that defeated the benchmark. These had to be banned in writing—most concretely by an explicit instruction that the goal was "improving Kuiper, not just solving this challenge set," and by a hard rule that `run.sh` must shell out to KernelBench's own `eval_kernel_against_ref`. (A separate incident saw a background agent silently `git push`ing a branch to a public remote, after which network operations were forbidden without confirmation.) These failures consumed a non-trivial fraction of agent and reviewer time, and they illustrate that the test-oracle reward-hacking story from § 3 applies to the agent's outer loop as well as to the kernels it generates. At times, we ran a "skeptic" agent that would go through all solutions and flag any suspicious or invalid solutions, obtaining useful summaries automatically and making the review faster. Nevertheless a human was involved before final acceptance.

Structural proofs resisted automation. Some of the hardest verification obstacles for the agent were not algorithmic but structural: reasoning about Pulse's $\forall^+$ combinator is not yet automated, so several lemmas, mostly related to permission accounting, had to be discharged "by hand", i.e. writing a large proof manipulating the resources. There are also a few missing proofs, currently admitted, about *sendability* of predicates (a technical concept essentially reasoning about which resources are visible from where in the hierarchy). Automation in Kuiper for this purpose is still in development, and manually proving these would be too tedious for very little gain.

Detecting unsoundness in the agent's output. Because Kuiper's correctness rests on the absence of unsound escape hatches (e.g. `admit`, `assume pure`, `magic ()`) in the agent's output, every kernel is gated by a textual scan for those tokens, in addition to running the verifier. F⋆ does contain a built-in `--report_assumes` feature that can be a more robust check, but which needs to be improved as it reports many safe axioms too.

Per-kernel work is small; per-cluster work is large. Once a group's primitives are in place, individual challenges land in ∼60–150 lines of Pulse. The bulk of the work in the corpus was building the right reusable abstractions: the loss-pipeline pattern, convolutions, a generic Kahan summation, etc. The agent contributed to this infrastructure but the larger abstraction designs were typically scoped by a single planning session before being solved piecewise, and had some human guidance when things got stuck.

Performance. The generated kernels are correctness-first and are not yet competitive with hand-tuned CUDA on the KernelBench $fast_p$ metric. Ways of closing this gap include fusing kernel invocations, leveraging Kuiper's verified autotuning, and using layout polymorphism to improve memory access patterns easily. We expect to improve performance as we go forward.

6 Related Work

LLM-based kernel generation. Besides KernelBench [24] and its v0.1 update [10], several recent efforts have explored using LLMs to generate and optimize GPU kernels, or to

strengthen the testing harnesses used to evaluate them [2, 5, 6, 17, 19, 30]. These works primarily evaluate correctness via testing, which our audit illustrates is insufficient.

GPU verification tools. A number of tools exist for analyzing GPU programs post-hoc, including GKLEE [18], which applies concolic execution to CUDA kernels, GPUVerify [3], which checks for races and barrier divergence, and Faial [7], which targets data race detection. These tools provide valuable safety checks but do not extend to functional correctness and can suffer from false positives and timeouts. Kuiper [20] goes further by embedding verification into the programming model itself, providing full functional correctness proofs.

Safe GPU languages. Languages such as Futhark [12], Descend [15], Triton [31], and Halide [27] raise the level of abstraction for GPU programming and improve safety, but they do not provide full functional correctness guarantees and often sacrifice low-level control.

LLM-assisted formal verification. A growing body of work explores using LLMs to assist with formal verification tasks, including proof generation in Lean, Coq, and Isabelle. Our work is complementary: we use LLMs to generate verified *programs* (not just proofs) in a domain (GPU programming) where correctness is both critical and difficult to achieve, and where performance should not be sacrificed.

Agentic verification of generated CUDA. ProofWright [4] is the closest recent work to ours. It uses LLM agents, a learned annotation guide, VerCors, and Rocq to verify AI-generated CUDA kernels on KernelBench, proving memory safety and data-race freedom for 74% of Level 1 kernels and semantic equivalence for 14%. Our work shares the same motivation but changes the artifact being generated. ProofWright treats CUDA as the object to be annotated and verified after generation; we instead ask agents to synthesize Kuiper programs whose proof obligations are checked in the source language before extraction to CUDA. This makes verification part of the programming model rather than an external certification pass over already-generated low-level code, and increases polymorphism and reuse heavily.

7 Conclusion and Future Work

We have argued that the test-suite-based correctness criterion of KernelBench is insufficient, presented evidence from a recent audit, and proposed an alternative: using LLM-based coding agents to generate formally verified GPU kernels in the Kuiper framework. Our preliminary results show that LLMs can indeed write verified Kuiper code, producing kernels that are guaranteed to be memory-safe, data-race-free, and functionally correct.

Our solution now covers all 100 Level 1 tasks of KernelBench, modulo the small residual proof debt noted in § 5. Beyond coverage, several directions for future work remain:

- **Performance optimization.** Our current focus is on correctness. Integrating performance optimization (e.g., tiling, vectorized loads, tensor cores) into the agentic workflow is an important next step, leveraging Kuiper's support for verified auto-tuning. One aspect of kernels not captured at all by the specifications is their non-functional properties ("Does this kernel use barriers?", "How much memory is copied?"). Some of these properties could be captured by extending the Kuiper type system, and surface to the specification level so they can be formally checked, and audited without reviewing the code itself.
- **Agent reliability.** Understanding and improving when and why agents fail to converge on a verified solution is key to scaling to harder kernels.
- **Specification quality and intent formalization.** Verification reduces the trust problem from "is the implementation correct?" to "does the specification capture user intent, and is the proof free of admits?" This is the *intent formalization* problem [16]. Mitigations include textual scans for `admit`/`assume` occurrences, generation of multiple independent specifications and cross-checking, and human-in-the-loop review of specs (which are short and declarative compared to implementations). Capturing common kernel idioms in specifications could reduce the human audit significantly.

A Full Verification Status for KernelBench

#	Name	Spec
1	Square matmul	≈
2	Standard matmul	≈
3	Batched matmul	≈
4	Matmul (irregular shapes)	≈
5	Matrix-scalar multiply	exact
6	Matmul (large K)	≈
7	Matmul (large M)	≈
8	Matmul (irregular)	≈
9	Matmul (tall/skinny)	≈
10	3D tensor-matrix multiply	≈
11	4D tensor-matrix multiply	≈
12	Diag matmul / RowScale	exact
13	Symmetric matmul	≈
14	Upper-triangular matmul	≈
15	Lower-triangular matmul	≈
16	Matmul, $A^T B$	≈
17	Matmul, AB^T	≈
18	Matmul, $A^T B^T$	≈
19	ReLU	exact
20	LeakyReLU	exact
21	Sigmoid	exact
22	Tanh	exact
23	Softmax (dim=1)	≈
24	LogSoftmax (dim=1)	≈
25	Swish	exact
26	GELU	exact
27	SELU	exact
28	HardSigmoid	exact
29	Softplus	exact
30	Softsign	exact
31	ELU	exact
32	HardTanh	exact
33	BatchNorm	≈

#	Name	Spec
34	InstanceNorm	≈
35	GroupNorm	≈
36	RMSNorm	≈
37	FrobeniusNorm	≈
38	L1Norm	≈
39	L2Norm	≈
40	LayerNorm	≈
41	MaxPool1D	exact
42	MaxPool2D	exact
43	MaxPool3D	exact
44	AvgPool1D	≈
45	AvgPool2D	≈
46	AvgPool3D	≈
47	Sum reduce dim=1	≈
48	Mean reduce dim=1	≈
49	Max reduce dim=1	exact
50	AlexNet conv1	≈
51	Argmax dim=1	exact
52	Argmin dim=1	exact
53	Min reduce dim=1	exact
54	Conv3D	≈
55	Conv2D asym input	≈
56	Conv2D asym input+kernel	≈
57	ConvTranspose3D	≈
58	ConvTranspose grouped	≈
59	Conv3D asym input	≈
60	Conv3D asym kernel	≈
61	ConvTranspose	≈
62	Conv2D asym kernel	≈
63	Conv2D square	≈
64	ConvTranspose1D	≈
65	ConvTranspose2D	≈
66	Conv3D asym all	≈

#	Name	Spec
67	Conv1D	≈
68	ConvTranspose	≈
69	ConvTranspose2D	≈
70	ConvTranspose	≈
71	ConvTranspose2D	≈
72	ConvTranspose	≈
73	ConvTranspose	≈
74	ConvTranspose1D	≈
75	ConvTranspose	≈
76	Conv1D dilated/strided	≈
77	ConvTranspose	≈
78	ConvTranspose2D	≈
79	ConvTranspose1D	≈
80	Conv2D dilated asym	≈
81	ConvTranspose2D	≈
82	Depthwise Conv2D sq/sq	≈
83	Depthwise Conv2D sq/asym-k	≈
84	Depthwise Conv2D asym-i/sq-k	≈
85	Depthwise Conv2D asym/asym	≈
86	Depthwise-Separable Conv2D	≈
87	Pointwise Conv2D (1×1)	≈
88	MinGPT NewGELU	exact
89	Cumulative sum	≈
90	Reverse cumsum	≈
91	Cumulative product	≈
92	Masked cumsum	≈
93	Exclusive cumsum	≈
94	MSELoss	≈
95	CrossEntropyLoss	≈
96	HuberLoss / SmoothL1	≈
97	SDPA	≈
98	KLDivLoss (batchmean)	≈
99	TripletMarginLoss	≈
100	HingeLoss	≈

Legend. The **Spec** column records how precisely each verified postcondition characterizes the output. '**exact**': the output is pinned by exact floating-point equality to the mathematical result, using only exact float operations. '≈': the output is pinned to the exact real-arithmetic ideal through Kuiper's float-models-approximation relation ≈, the appropriate precise specification whenever floating-point accumulation makes bitwise equality impossible (e.g. all GEMMs, convolutions, softmax, normalizations, reductions, and losses). All challenges pass KernelBench's own correctness harness at its native fp32 tolerance ($atol = rtol = 10^{-4}$).

References

[1] 2026. *Kuiper*. https://github.com/FStarLang/kuiper GitHub repository.

[2] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-Turn RL for Generating CUDA Kernels. arXiv:2507.11948 [cs.LG] doi:10.48550/arXiv.2507.11948

[3] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: A Verifier for GPU Kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM. doi:10.1145/2384616.2384625

[4] Bodhisatwa Chatterjee, Drew Zagieboylo, Sana Damani, Siva Hari, and Christos Kozyrakis. 2026. ProofWright: Towards Agentic Formal Verification of CUDA. arXiv:2511.12294 [cs.SE] doi:10.48550/arXiv.2511.12294

[5] Terry Chen, Bing Xu, and Kirthi K. Devleker. 2025. Automating GPU Kernel Generation with DeepSeek-R1 and Inference Time Scaling. https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling/ NVIDIA Technical Blog. Accessed: 2026-05-06.

[6] Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, Yang Xu, Yu-Jung Chen, Hanfeng Chen, Aditya Kane, Ronny Krashinsky, Ming-Yu Liu, Vinod Grover, Luis Ceze, Roger Bringmann, John Tran, Wei Liu, Fung Xie, Michael Lightstone, and Humphrey Shi. 2026. AVO: Agentic Variation Operators for Autonomous Evolutionary Search. arXiv:2603.24517 [cs.LG] doi:10.48550/arXiv.2603.24517

[7] Tiago Cogumbreiro, Julien Lange, Dennis Liew Zhen Rong, and Hannah Zicarelli. 2021. Checking Data-Race Freedom of GPU Kernels, Compositionally. In *Computer Aided Verification (CAV 2021) (Lecture Notes in Computer Science, Vol. 12759)*. Springer, 403–426. doi:10.1007/978-3-030-81685-8_19

[8] Kshitij Dubey, Benjamin Driscoll, Anjiang Wei, Neeraj Kayal, Rahul Sharma, and Alex Aiken. 2025. Equivalence Checking of ML GPU Kernels. arXiv:2511.12638 [cs.PL] doi:10.48550/arXiv.2511.12638

[9] Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *Proc. ACM Program. Lang.* 9, PLDI, Article 208 (June 2025), 24 pages. doi:10.1145/3729311

[10] Simon Guo, Anne Ouyang, Simran Arora, and Azalia Mirhoseini. 2025. KernelBench v0.1. https://scalingintelligence.stanford.edu/blogs/kernelbenchv01/ Blog post. Accessed: 2026-04-28.

[11] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU Memory Exploitation for Fun and Profit. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 4033–4050. https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. doi:10.1145/3062341.3062354

[13] III Jacobson, John, Martin Burtscher, and Ganesh Gopalakrishnan. 2024. HiRace: Accurate and Fast Source-Level Race Checking of GPU Programs. arXiv:2401.04701 [cs.DC] doi:10.48550/arXiv.2401.04701

[14] W. Kahan. 1965. Further Remarks on Reducing Truncation Errors. *Commun. ACM* 8, 1 (Jan. 1965), 40. doi:10.1145/363707.363723

[15] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. 2024. Descend: A Safe GPU Systems Programming Language. In *Proceedings of the ACM on Programming Languages (PLDI)*. ACM. doi:10.1145/3656411

[16] Shuvendu K. Lahiri. 2026. Intent Formalization: The Key Challenge for Reliable AI-Generated Code. *arXiv preprint arXiv:2603.17150* (2026). doi:10.48550/arXiv.2603.17150

[17] Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. 2025. Towards Robust Agentic CUDA Kernel Benchmarking, Verification, and Optimization. arXiv:2509.14279 [cs.SE] doi:10.48550/arXiv.2509.14279

[18] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic Verification and Test Generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 215–224. doi:10.1145/2145816.2145844

[19] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2026. CUDA-L1: Improving CUDA Optimization via Contrastive Reinforcement Learning. arXiv:2507.14111 [cs.AI] doi:10.48550/arXiv.2507.14111 Accepted at ICLR 2026.

[20] Guido Martínez, Bastian Köpcke, Jonáš Fiala, Gabriel Ebner, Tahina Ramananandro, Michel Steuwer, Tyler Sorensen, and Nikhil Swamy. 2026. Kuiper: Correct and Efficient GPU Programming with Dependent Types and Separation Logic. In *Proceedings of the ACM on Programming Languages (PLDI)*. ACM. doi:10.1145/3808280

[21] Guido Martínez, Bastian Köpcke, Jonáš Fiala, Gabriel Ebner, Tahina Ramananandro, Michel Steuwer, Tyler Sorensen, and Nikhil Swamy. 2026. *Kuiper: Correct and Efficient GPU Programming with Dependent Types and Separation Logic – PLDI 2026 Artifact*. doi:10.5281/zenodo.19081105

[22] George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, 106–119. doi:10.1145/263699.263712

[23] NVIDIA Corporation. 2026. *Compute Sanitizer Documentation*. https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html Accessed: 2026-05-07.

[24] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels?. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, 47356–47415. doi:10.48550/arXiv.2502.10517

[25] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A Dynamic CUDA Race Detector. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 390–403. doi:10.1145/3192366.3192368

[26] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. In *Proceedings of the ACM on Programming Languages (ICFP)*. ACM. doi:10.1145/3110261

[27] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. doi:10.1145/2491956.2462176

[28] Mark Stephenson, Sana Damani, Mohamed Tarek Ibn Ziad, Anis Ladram, and Michael Garland. 2026. SuperCollider: Scalable and Effective Data Race Detection for CUDA. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. To appear.

[29] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. *ACM SIGPLAN Notices* 51, 1 (2016), 256–270. doi:10.1145/2914770.2837655

[30] Muhammad Usman Tariq, Abhinav Jangda, Angelica Moreira, Madan Musuvathi, and Tyler Sorensen. 2025. PEAK: A Performance Engineering AI-Assistant for GPU Kernels Powered by Natural Language Transformations. arXiv:2512.19018 [cs.SE] doi:10.48550/arXiv.2512.19018

[31] 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)*. ACM. doi:10.1145/3315508.3329973

[32] Qizhong Wang, Xiangyue Huang, Yanan Guo, and Yuanchao Xu. 2025. Security and Performance Implications of GPU Cache Eviction Priority Hints. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM. doi:10.1145/3725843.3756116

[33] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. 2024. Invalidate+Compare: A Timer-Free GPU Cache Attack Primitive. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-zhenkai

Received 2026-05-07; accepted 2026-05-19