Project Proposal

cLean : Verifiable CPU Kernel Programming in Lean 4

Riyaz Ahuja

URL

The project webpage, as well as all code and documentation, will be hosted at: https://github.com/riyazahuja/cLean

Summary

We are building a domain-specific language embedded in Lean 4 for writing CUDA-style GPU kernels, efficiently executing them on-device, and formally verifying their correctness and safety. Namely, our system includes a full language and transpiler for efficiently and easily writing GPU kernels integrated with the Lean 4 programming language and proof assistant, as well as a formal semantics and verification framework for proving safety properties about these kernels. Moreover, we implement an execution engine for running these kernels on both GPUs and on a CPU simulator - extending prior work on verified kernel programming (e.g. GPUVerify) by integrating with a higher-order fully-featured programming language and interactive theorem prover to not only speed up, but also provide stronger developer tooling for safe GPU programming.

Background

GPU kernels are critical for modern high-performance computing, and as these kernels run at large scale and often manipulate shared memory across thousands of threads, safety (e.g., absence of data races, correct synchronization, etc) and correctness (the kernel actually implements the intended algorithm) are essential. Race conditions or incorrect thread interaction can silently corrupt results at scale, and debugging these behaviors on a GPU is notoriously difficult. Thus, a framework for writing GPU kernels that can be formally and statically verified before execution has enormous practical value.

Prior work in GPU kernel verification includes tools such as GPUVerify and GKLEE. GPUVerify in particular pioneered the “two-thread abstraction,” reducing a parallel GPU kernel to a sequential program whose properties can be checked by an SMT solver. This technique successfully proves race-freedom and barrier-safety for many small kernels, and follow-up work enhanced invariant generation to increase automation. However, these systems show clear limitations: as kernels become more complex, especially when they include loops with non-trivial inductive structure, irregular memory accesses, or deeper arithmetic reasoning, the underlying SMT solver often fails to discharge the verification conditions. This highlights a fundamental limitation of purely SMT-based approaches to verification: they lack the expressive power and interactive proof capabilities needed to reason about more sophisticated GPU programs. Moreover, these prior systems focus almost exclusively on safety; they do not attempt to verify functional correctness of the kernel relative to a high-level specification.

Lean 4, a modern interactive theorem prover designed around a small trusted kernel and a high-performance native compiler, offers the capabilities that SMT-only tools lack. Lean’s metaprogramming framework allows us to define custom DSLs, manipulate syntactic objects, and generate code programmatically. Its proof system provides powerful interactive tactics (`simp`, `auto`, `lean-smt`, etc.) that combine automation of proof search in higher-order logic and dependent type theories with human interactivity. Lean also supports reasoning about inductive invariants,

algebraic specifications, Hoare-style pre/post-conditions, and correctness statements that SMT solvers alone cannot robustly handle. This combination makes Lean an excellent foundation for GPU kernel verification in a way that allows the complex underlying parallelism to be abstracted away from the user, as well as enabling easy integration of GPU acceleration with computer algebra in formalized mathematics.

Our approach is to implement a verified GPU DSL extending GPUVerify's approach directly inside Lean 4, giving it a fully formal semantics. Using Lean's metaprogramming, we will then transpile this DSL to CUDA C++, allowing for live execution via Lean's `Infoview`. Additionally, in the transpilation process, our system will also produce a corresponding Lean expression representing the kernel's semantics, allowing us to apply Lean's tactic framework to automatically or interactively prove properties of the kernel, namely, including race-freedom, barrier-safety, and optionally - full functional correctness relative to a user-specified contract. Loop invariants, pre-/post-conditions, and algebraic correctness proofs can all be carried out inside Lean, enabling verification far beyond what SMT-only systems can achieve.

With this system, we then will proceed to not only verify small benchmark kernels, but as a stretch goal, also apply it to the acceleration of a critical compute-intensive task inside Lean itself: an efficient Gröbner-basis-based ideal-membership algorithm. As Lean is primarily used by mathematicians for formalization, there is a large demand for efficient algebraic reasoning systems, and the current tactics for testing ideal membership in commutative algebra and algebraic geometry are primarily done by CPU-bound code called via API from external CAS systems. Namely, this algorithm is highly parallelizable, as each monomial reduction step can be run concurrently on device:

1) **IdealMembership(f, I = ⟨g_1, ..., g_s⟩)**:
   a) Compute Gröbner basis G for ideal I: `G ← GrobnerBasis(g₁, ..., gₛ)`
   b) Reduce f modulo G using multivariate division: `r ← MultivariateDivision(f, G)`
   c) If $r = 0$, return true; else return false

2) **GröbnerBasis(g_1, ..., g_s)**:
   a) Initialize `G ← {g₁, ..., gₛ}`
   b) Initialize `P ← {all pairs (gᵢ, gⱼ) where i < j}`
   c) While P ≠ ∅:
      a) Select pair `(p, q) ← SelectPair(P)` (parallelizable across pairs)
      b) `P ← P \ {(p, q)}`
      c) Compute S-polynomial and reduce: `S ← SPolynomial(p, q)` (GPU parallel)
      d) `r ← MultivariateDivision(S, G)`
      e) If `r ≠ 0`:
         - Add new pairs with `r` to queue: `P ← P ∪ {(r, g) | g ∈ G}`
         - `G ← G ∪ {r}`
   d) Return G

3) **MultivariateDivision(f, G = {g_1, ..., g_s})**:
   a) Initialize `r ← 0, h ← f`
   b) While `h ≠ 0`:
      a) `LT_h ← LeadingTerm(h)`
      b) `reduced ← False`
      c) For i from 1 to s (parallel check: which g_i divides LT(h)):
         - If `LT(gᵢ) | LT_h`:
           ▸ `q ← LT_h / LT(gᵢ)`

        - `h ← h - q · g_i`
        - `reduced ← True`
        - Break
    d) If ¬reduced:
        - `r ← r + LT_h`
        - `h ← h - LT_h`
c) Return `r`

This suggests that not only can cLean provide an avenue for efficient and safe implementation of this algorithm in Lean, but also improve on existing systems by offloading this highly data-parallel computation to the GPU.

The Challenge

This problem is challenging primarily because of the multiple layers of complexity involved in the wide scope. First, the design and implementation of a DSL for GPU kernels that is both ergonomic and efficient is non-trivial, especially when embedded in a host language like Lean. Second, the actual metaprogramming and compiler construction to transpile this DSL to CUDA C++ while maintaining a formal semantics is a complex task. Additionally, the verification of GPU kernels, especially for functional correctness, requires sophisticated reasoning about parallelism, memory models, and algorithmic properties that go beyond traditional verification techniques.

Namely, integrating all these components into a cohesive system that is both usable and powerful represents a large engineering and research challenge, as we must ensure that our DSL must map semantically to GPU execution *and* to a proof environment. Then , we must also ensure that the verification framework is robust enough to handle real-world kernels, which often involve complex control flow, memory access patterns, and arithmetic reasoning. Additionally, we need to ensure that the transpiled code is efficient and can run on real GPU hardware efficiently, including any and all low-level optimizations.

Moreover, with our stretch goal of implementing an efficient Gröbner-basis-based ideal-membership algorithm, we face the additional challenge of not only implementing this complex system for authoring GPU kernels, but then properly parallelizing and optimizing a complex algorithm with intricate data dependencies and algebraic structures. Namely, the F_4 and F_6 algorithms (efficient variants of the aforementioned algorithm) involve irregular data structures (polynomials) that are not natively data-local, and although there is high arithmetic intensity during the matrix-reduction sections, each such reduction may depend on prior reductions, so we will need to be careful to ensure that any branching or synchronization does not lead to divergence or performance degradation on the GPU.

Resources

We will not be using any starter code for our project. Additionally, our project will be running on the GHC machines as needed for GPU execution since they already have CUDA-capable GPUs and the necessary software installed. We will be referring to the following paper for guidance on GPU kernel verification:

- Betts, A., Chong, N., Donaldson, A. F., Qadeer, S., & Thomson, P. (2012). GPUVerify: a verifier for GPU kernels. Proceedings of OOPSLA, pages 113–132. https://doi.org/10.1145/2384616.2384625

The following book for guidance on Lean 4 metaprogramming and tactics:

- Boucher, W., & The Lean Prover Community. (n.d.). Metaprogramming in Lean 4. Retrieved from https://leanprover-community.github.io/lean4-metaprogramming-book/

And the following paper for guidance on GPU-accelerated Gröbner basis algorithms:

- Lesnoff, D. (2022). Efficient Gröbner bases computation on GPU [Internship report]. LIP6 – Sorbonne Université & CNRS. Retrieved from https://www.lip6.fr/Dimitri.Lesnoff/pdf/rapportM2_polsys.pdf

Goals and Deliverables

Plan to Achieve

The core and minimal deliverables for this project are:

1) DSL Simulator
   - Provide a Lean 4-embedded DSL for GPU kernels supporting threadIdx, blockIdx, global and shared memory, loops and branching.
   - Build a deterministic CPU simulator for kernels written in this DSL, enabling off-device testing and trace equivalence checking.
2) Transpiler to GPU
   - Implement a transpiler that takes DSL kernels → CUDA C++ (or CUDA C) code, including kernel launch wrappers and memory management.
   - Demonstrate equivalence: run the simulator and the generated GPU code on simple kernels (e.g., vector add, reduction) and show matching output.
   - Measure compile time for our system (DSL/transpiler) and compare rough baseline compile time of hand-written CUDA and Python/Triton (for similar kernels).
3) Verification Framework
   - Build proof infrastructure inside Lean 4 to reason about safety properties (no data races, no barrier divergence) of DSL kernels.
   - Benchmark proof coverage: pick a small kernel benchmark set (e.g., from GPUVerify suite) and measure how many kernels we can successfully verify vs. how many prior tools could verify.
4) Quantitative Evaluation
   - Performance comparison: As previously mentioned, for selected kernels in our testing dataset, measure execution time on (a) hand-written CUDA C++, (b) Python/Triton version (if available), (c) our DSL/transpiler version. We will report speed-up factors, compile time overhead, generated code overhead and hope to show that our system is competitive with hand-written CUDA while providing additional formal verification guarantees.
   - Proof comparison: report number of kernels verified, time to verify (or approximate) inside Lean, any proof automation we built (e.g., custom tactics).
   - (For the poster/demo: show graphs of compile time vs kernel type, execution time across different systems, proof coverage percentages, and live examples of the workflow (DSL code → Lean proof → GPU run))

Hope to Achieve

If work goes more quickly (stretch goals):

- Apply the system to the Gröbner basis ideal-membership workload: implement kernel, transpile it, run on GPU, measure speed-up vs CPU and other baselines, and verify correctness in Lean (even if partially).
- Achieve proof coverage exceeding GPUVerify's published coverage for the benchmark suite (for safety and, if possible, correctness). For example, where GPUVerify verified 231/253 programs under candidate-based invariants.
- Extend to prove functional correctness properties for a small set of kernels: e.g., proving that the result equals the known sequential computation.

Platform Choice

- GPU Platform: We choose NVIDIA's CUDA platform because it provides a mature, high-performance GPU programming model that we have already covered and learned about at length in 15418. We will specifically using a NVIDIA GeForce NTX 2080 GPU on the GHC cluster machines for testing and running our transpiled code, as they have the necessary hardware and software stack (CUDA toolkit, drivers) pre-installed.
- Lean 4: We select Lean 4 as our host language and proof assistant because it offers powerful meta-programming and DSL-definition facilities (e.g., one can build syntax categories and internal AST in Lean) in its functional and dependently typed framework. It also supports writing proofs and linking them to program semantics.

We moreover leverage these platforms as for our core goal of verifying GPU kernels, Lean 4′s proof system provides the necessary expressiveness and interactivity that SMT-based tools lack, while CUDA provides the necessary low-level control and performance for GPU execution. Moreover, for an algorithm like Gröbner basis computation, which is highly parallelizable but also requires intricate reasoning about algebraic structures, the combination of Lean 4 for verification and CUDA for execution is particularly well-suited, whereas applying other parallelization platforms (i.e. OpenML, etc.) would not allow for the same level of integration between the algorithm in Lean, and the parallelized form, as well as the efficient execution of the matrix reduction steps.

Schedule

Week	**Dates**	**Planned Tasks**
Week 1	Nov 17 – Nov 23	Finalize DSL syntax and semantics and connect to GPU backend execution with transpiler. Connect transpiler to Lean metaprogramming framework for proving safety conditions
Week 2	Nov 24 – Nov 30	Optimize DSL framework for proving and write novel tactics for proving race-freedom and barrier-safety. Clean up transpiler backend and speed it up.
Milestone Report	(Dec 1)	Submit milestone report showing working basic examples of DSL to transpiler to GPU code, as well as safety proofs for simple kernels. Run evaluations on GPUVerify dataset and report coverage and compile time metrics.
Week 3	Dec 1 – Dec 7	Continue polishing and bugfixing as needed, and if time, upgrade the DSL to handle functional correctness specifications and Gröbner basis ideal-membership kernel (first in regular C++, then in Lean). Generally finalize the system, make examples, finalize benchmarks, and write report.