# Milestone Report

## cLean: Verifiable GPU Kernel Programming in Lean 4

Riyaz Ahuja
Fall 2025

## 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, dependently typed programming language and interactive theorem prover to not only speed up, but also provide stronger developer tooling for safe and interactive GPU programming.

## Progress Summary

Since the proposal, we have implemented the core end-to-end pipeline for cLean and validated it on several example kernels. At a high level, the system now supports:

- A Lean-embedded GPU DSL
- Macros for kernelArgs and device_kernel that elaborate a Lean do-block into:
- A Lean KernelM function for CPU simulation.
- A DeviceKernel object in our DeviceIR capturing arguments, statements, and expressions.
- Support for standard GPU concepts: threadIdx, blockIdx, globalIdx, global arrays, (basic) shared memory, conditionals, and loops.
- CPU Simulation & Runtime
- A deterministic CPU simulator in Lean (KernelM + KernelState) that iterates over grid/block/thread indices and executes the kernel body for each logical thread.
- Support for global and shared memory, scalar arguments, and simple barrier simulation (single-threaded semantics; the verification layer is responsible for race/barrier reasoning).
- CUDA Code Generation & GPU Execution
- A DeviceIR ⟶ CUDA C++ backend that:
  - Maps DType, DExpr, and DStmt to C++/CUDA syntax.
  - Emits complete compilable CUDA translation units.
- A C++ gpu_launcher host that:
  - Reads JSON from stdin (arguments and arrays).
  - Loads PTX via the CUDA Driver API.
  - Launches kernels with specified grid/block dimensions.
  - Copies results back to host and prints JSON.
- Lean glue code (ProcessLauncher) that:
  - Computes a hash of IR for kernel caching.

    - Invokes nvcc, runs gpu_launcher, and parses results back into Lean types.
    - Safety Verification Infrastructure (GPUVerify-Style)
- A verification IR / kernel spec derived from DeviceIR:
    - Representation of address/access patterns, barriers, and control flow.
    - A KernelSpec structure for abstracting kernel behavior.
- Formal predicates for:
    - RaceFree (no conflicting accesses between different threads).
    - BarrierUniform (no barrier divergence).
    - KernelSafe combining both.
- A VC generator that:
    - Uses a two-thread abstraction (threads t1, t2).
    - Extracts read/write address patterns and generates race-freedom conditions.
    - Initial automation tactics (e.g., solve_race_freedom) that discharge proof obligations for simple kernels using simp, arithmetic reasoning, etc.

Moreover, we have many example kernels & end-to-end tests. Namely, cLean contains working examples for:
- Vector operations (e.g., SAXPY-style kernels).
- Prefix sum (exclusive scan)
- Basic matmul; blocked matmul; shared memory matmul
- Basic shared-memory and synchronization-aware kernels

These can be:
- Simulated on CPU in Lean.
- Compiled & run on GPU through the PTX + launcher path.
- Proved safe for a subset of kernels using our safety framework.

In other words, the core "write ⟶ simulate ⟶ compile ⟶ run ⟶ verify" loop is implemented and exercised on numerous non-trivial examples. The system is already usable for writing and testing small kernels with safety proofs inside Lean.

## Status vs Original Goals

Original Core Deliverables (from proposal)

1) **DSL + Simulator:** A Lean-embedded DSL supporting threads, blocks, global/shared memory, loops, and branching; this should be connected to a CPU simulator for deterministic testing and trace comparison.

   **Status:**
   - DSL and macros are implemented and integrated with Lean's elaborator.
   - CPU simulator exists and is used heavily in examples.
   - This deliverable is effectively complete, aside from ongoing polish (robustness, better error messages).

2) **Transpiler to GPU + Equivalence Demos:** Implement a DeviceIR ⟶ CUDA C++ transpiler. Then run the same kernel via CPU simulator and GPU backend and demonstrate matching outputs. Also roughly compare with hand-written CUDA.

   **Status:**
   - Transpiler and CUDA codegen are implemented and working; kernels compile and run via gpu_launcher.
   - We have sanity-check tests where CPU and GPU outputs match on small kernels.

- System-level compile-time measurement and structured comparison vs hand-written CUDA/Triton are not yet done; we plan to do this in the evaluation phase.

3) **Verification Framework (Safety)**: Develop Lean-internal proof infrastructure to reason about no data races / no barrier divergence. Benchmark coverage on a small kernel set vs GPUVerify.

   **Status:**
   - Core safety semantics, VC generation, and GPUVerify-style reasoning infrastructure are implemented.
   - Tactics can prove safety automatically for some simple kernels in our examples (e.g., straightforward elementwise kernels and basic shared-memory patterns).
   - A full, apples-to-apples comparison on the entire GPUVerify benchmark suite is turning out to be too heavy for our timeline (format mismatch, porting overhead, and nontrivial invariant engineering). We are therefore updating this goal (see §3).

4) **Quantitative Evaluation**: Implement performance comparisons across:
   - Hand-written CUDA.
   - Python/Triton.
   - cLean DSL + transpiler.

   Also log proof metrics: number of kernels verified, verification time, proof automation.

   **Status:**
   - Infrastructure is in place to run kernels and time them, but we have not yet run a systematic performance study.
   - We have preliminary success in automatically proving safety for small kernels but no systematic "coverage" metrics yet.
   - This remains a key focus of the remaining schedule.

Stretch Goals
- Gröbner-basis ideal-membership kernel ($F_4$/$F_5$/$F_6$-style), GPU-accelerated.
- Functional correctness proofs for kernels (not just safety).
- Beating GPUVerify coverage on published benchmarks.

Status:
- Gröbner basis kernel: not yet implemented; still in design stage (data layout, batching strategy, and tile structure).
- Functional correctness:
- We have partial infrastructure (semantic IR, idea of generateFunctionalSpec) but not a complete, polished pipeline or user-facing tactics.
- GPUVerify coverage: we are deferring a large-scale, direct benchmark replication and instead focusing on a curated test suite (see below).

## Updated Goals for Final Poster / Demo

Given what we've learned so far and the time remaining, a realistic set of goals for the final poster/demo is as follows:

**Core Goals:**

1) Robust End-to-End System Demo
   - Show the full pipeline on several representative kernels:
   - DSL in Lean ⟶ CPU simulation ⟶ safety proof ⟶ CUDA generation ⟶ GPU execution.
   - Include at least:
   - A simple map-style kernel (e.g., SAXPY).

   - A shared-memory kernel (e.g., tiled matmul or prefix sum).
2) Safety Verification on a Curated Kernel Suite
   - Construct a custom small benchmark suite of kernels that mix:
   - Simple but nontrivial loops and index arithmetic.
   - Shared memory access patterns.
   - Barriers and synchronization.
   - Show that cLean can automatically verify safety (race freedom and barrier uniformity) for kernels that GPUVerify either:
     - Fails to verify, or
     - Verifies but only with heavier invariant engineering / less flexibility.
     - Measure verification time in Lean vs GPUVerify (where applicable).
3) Performance Evaluation
   - For a subset of kernels, compare runtime of:
     - Hand-written CUDA baseline.
     - cLean-generated CUDA/PTX run via gpu_launcher.
     - CPU simulation in Lean (as a baseline for overhead).
   - Show that GPU performance of cLean-generated code is within a reasonable constant factor of hand-written CUDA and a significant speedup over CPU simulation.
   - Also show that Lean overhead (kernel generation + proof) is acceptably low (relative to the benefits of verification).
4) Functional Correctness Prototype
   - Implement and demonstrate at least one full kernel-level functional correctness proof:
   - Example: a map-style or reduction kernel where we can state and prove that the final output matches a pure mathematical specification (e.g., $\text{result}[i] = \alpha \cdot x[i] + y[i]$ or a sequential reduction).
   - Even if this is for a small kernel, the goal is to show the end-to-end story: Lean mathematical spec $\longrightarrow$ kernel implementation $\longrightarrow$ proof that they coincide.
   - Proof UX Improvements
   - Make proofs more "visually easy" and ergonomic for users:
   - Introduce convenience tactics/macros like prove_kernel_safety saxpyKernel and prove_kernel_correct saxpyKernel spec.
   - Provide clearer error messages and better structuring of generated VCs.

**Stretch Goals**
- Implement a GPU kernel for (part of) an $F_4$/$F_6$-style Gröbner basis pipeline (e.g., batched dense linear algebra or reduction of polynomials represented as sparse/dense arrays).
- Run evaluations on dataset of performance of cLean vs simulation vs CPU vs CUDA.
  - Run evaluations on dataset of verification coverage of cLean vs GPUVerify.
- Implement broader functional correctness and extend functional correctness proofs to multiple kernels or more involved algorithms (e.g., prefix sum correctness with respect to a list fold).

We are reverting our previous goal of a full and direct comparison on the entire GPUVerify benchmark suite, as porting and verifying all of those kernels to our DSL turned out to be a larger engineering task than anticipated. Instead, we'll focus on a smaller curated suite that still highlights cases where Lean-based reasoning has an advantage.

**Planned Poster / Demo Content**

At the poster session, we plan to show:
- Live Demo (if logistics allow):
  - Edit a kernel in Lean, re-generate code, and:

- Run it on the CPU simulator and show the output.
  - Run it on the GPU via gpu_launcher and show matching output and timings.
  - Trigger a safety proof (and at least one functional correctness proof) in Lean and show the proof script / automation.
- Graphs and Tables:
  - Performance evaluations
  - Verification coverage evaluations
  - Approximate proof / verification times.
- Side-by-side code / IR visualizations: Lean DSL code, generated DeviceIR, and emitted CUDA snippet.
- A diagram of the full cLean pipeline: Lean DSL ⟶ DeviceIR ⟶ Verification IR / VCs ⟶ CUDA codegen ⟶ PTX ⟶ GPU launcher.

## Preliminary Results

We do not yet have polished quantitative graphs and large evaluations, but we do have a robust set of qualitative results, which can be all found in https://github.com/riyazahuja/cLean. Namely, we have worked end-to-end examples of the full pipeline on several kernels, such as SAXPY, prefix sum, and matmul. For example, we show on SAXPY, we may easily write the following kernel natively in Lean:

```
kernelArgs saxpyArgs(N: Nat, alpha: Float)
  global[x y r: Array Float]

device_kernel saxpyKernel : KernelM saxpyArgs Unit := do
  let args ← getArgs
  let N := args.N
  let alpha := args.alpha
  let x : GlobalArray Float := ⟨args.x⟩
  let y : GlobalArray Float := ⟨args.y⟩
  let r : GlobalArray Float := ⟨args.r⟩

  let i ← globalIdxX
  if i < N then do
    let xi ← x.get i
    let yi ← y.get i
    r.set i (alpha * xi + yi)
```

Which transpiles first to the following DeviceIR:

```
def Saxpy.saxpyKernelIR : Kernel :=
{ name := "saxpyKernel", params := [{ name := "N", ty := DType.int }, { name :=
"alpha", ty := DType.float }],
  locals := [],
  globalArrays :=
    [{ name := "x", ty := DType.float.array, space := MemorySpace.global },
      { name := "y", ty := DType.float.array, space := MemorySpace.global },
      { name := "r", ty := DType.float.array, space := MemorySpace.global }],
  sharedArrays := [],
  body :=
    (DStmt.assign "i"
          (DExpr.binop BinOp.add (DExpr.binop BinOp.mul (DExpr.blockIdx Dim.x)
(DExpr.blockDim Dim.x))
            (DExpr.threadIdx Dim.x))).seq
      ((DStmt.ite (DExpr.binop BinOp.lt (DExpr.var "i") (DExpr.var "N"))
            ((DStmt.assign "xi" ((DExpr.var "x").index (DExpr.var "i"))).seq
              ((DStmt.assign "yi" ((DExpr.var "y").index (DExpr.var "i"))).seq
                ((DStmt.store (DExpr.var "r") (DExpr.var "i")
                      (DExpr.binop BinOp.add (DExpr.binop BinOp.mul (DExpr.var
"alpha") (DExpr.var "xi"))
                        (DExpr.var "yi"))).seq
                  DStmt.skip)))
            DStmt.skip).seq
        DStmt.skip) }
```

Which then generates the following compiled cuda code:

```
extern "C" __global__ void saxpyKernel(int N, float alpha, float* x, float* y, float*
r) {
  int i = ((blockIdx.x * blockDim.x) + threadIdx.x);
  if ((i < N)) {
    float xi = x[i];
    float yi = y[i];
    r[i] = ((alpha * xi) + yi);
  }
}
```

Which can be easily executed on device from Lean via the launcher:

```

    (α : Float)
    (x y : Array Float) : IO (Array Float) := do

  let scalarParams := #[Float.ofNat n, α]
  let arrays := [
    (`x, x),
    (`y, y),
    (`r, Array.replicate n 0.0)
  ]

  let response ← runKernelGPU saxpyKernelIR saxpyArgsResponse
    ⟨(n + 511) / 512, 1, 1⟩         -- grid
    ⟨512, 1, 1⟩                     -- block
    scalarParams
    arrays
  return response.r
```

Moreover, we can prove safety of this kernel automatically in Lean, with the following (autogenerated) proof:

```
def saxpySpec (config grid: Dim3): KernelSpec :=
  deviceIRToKernelSpec saxpyKernelIR config grid

theorem saxpy_safe : ∀ (config grid : Dim3), KernelSafe (saxpySpec config grid) := by
  intro config grid
  unfold KernelSafe
  constructor
  . unfold RaceFree
    intro tid1 tid2 h_distinct a1 a2 ha1 ha2
    simp_all [saxpySpec, HasRace, SeparatedByBarrier, deviceIRToKernelSpec,
extractFromStmt, saxpyKernelIR,extractReadsFromExpr, dexprToAddressPattern,
List.lookup, AccessExtractor.getArrayLocation, getArrayName,
AddressPattern.couldCollide]
```

```
    intro h_race
    rcases h_distinct with ⟨_,_,h_neq⟩
    rcases ha1 with ha1 | ha1 | ha1 <;>
    rcases ha2 with ha2 | ha2 | ha2 <;>
    simp_all [AddressPattern.eval, SymValue.isNonZero]
  . simp [BarrierUniform]
```

These results give us confidence that the core architecture is sound and that it makes sense to invest the remaining time into functional correctness and evaluation.

## Open Issues and Main Concerns

The major remaining issues and concerns to be resolved, with respect to technical and engineering risks, are as follows:

1) **Functional Correctness End-to-End:** While we have many pieces (semantics, specs, IR translation), we still need to:
   - Finalize the functional correctness pipeline.
   - Design a user-friendly interface for specifying high-level specifications.
   - Ensure the generated goals are tractable for Lean's automation.
2) **Gröbner Basis Kernel Complexity:** We still must design a GPU-friendly Gröbner basis kernel (or subroutine) that:
   - Is expressive enough to be interesting
   - Fits nicely into our DSL
   - Can be verified (even partially)
3) **Proof Automation Robustness**: For more complex kernels, VCs may involve nontrivial reasoning that automation cannot handle. We may need to iterate on our tactics and possibly simplify/revise the IR to keep it manageable.

Overall, there are no fundamental unknowns left in terms of feasibility (we have a working system), but there is a nontrivial amount of engineering and tuning left to do.

## Updated Detailed Schedule

Below is an updated, more granular schedule from the milestone to the final deadline.

| Date Range | Task(s) |
|---|---|
| **Dec 2 – Dec 4** | **Functional Correctness Infrastructure**<br>• Finalize the translation from DeviceIR to a semantic representation suitable for functional specs.<br>• Implement a first end-to-end example:<br>‣ Define a pure Lean spec (e.g., saxpy_spec).<br>‣ Generate a correctness theorem for saxpyKernel.<br>‣ Write / refine tactics to prove it. |
| **Dec 5 – Dec 7** | **Benchmarking Suite & Safety Evaluation**<br>• Design and implement a small suite of kernels with varying complexity<br>• Implement timing harnesses for final evaluations<br>• Collect performance data for a subset of kernels and parameter ranges.<br>**Gröbner Basis Kernel**<br>• Settle on the final algorithm and implement. |
| **Dec 8** | **Finalize Evaluation & Plots – Riyaz**<br>• Finish final experiments, clean data, and produce performance + verification graphs.<br>• Select key kernels for side-by-side comparison vs GPUVerify and highlight interesting cases.<br>**Finalization**<br>• Make demo<br>• Make poster<br>• Make report<br>• Cleanup website. |