cLean: A Verified Domain-Specific Language for GPU Kernel Programming in Lean 4

Riyaz Ahuja
riyaza@andrew.cmu.edu
Carnegie Mellon University

Abstract

We present cLean[1], an embedded domain-specific language (DSL) for writing GPU kernels directly inside the Lean 4 theorem prover. cLean lets programmers express CUDA-style kernels natively in Lean's dependent type theory, interactively compiles and executes kernels on-device, and provides a verification framework for proving safety properties, such as race-freedom and barrier divergence-freedom, as well as mathematical proofs of correctness. Moreover, we propose, implement, benchmark, and verify a GPU-accelerated Grobner basis computation over finite fields – all within cLean, outperforming both the naive CPU baseline, and a state-of-the-art open-source library implementation. Additionally, we study the performance of cLean on a dataset of nontrivial CUDA kernels, showing that our transpiled kernels perform competitively with the reference CUDA implementations and strictly extend the abilities of prior static kernel verifiers, without the need for external SMT solvers and user-supplied loop invariants. This suggests that safe *and* high-performance GPU programming can live comfortably inside the context of modern interactive theorem provers.

Keywords GPU Programming · Formal Verification · Domain-Specific Languages · Lean 4 · CUDA · Race Freedom · Automated Theorem Proving

1 Background

GPU programming is a now standard practice in high-performance computing, with rich applications from machine learning and scientific simulation to cryptography and numerical algebra. Unfortunately, GPU kernels are also a rich source of subtle concurrency bugs: data races, barrier divergence, and out-of-bounds memory accesses are all easy to introduce and hard to find. Because these bugs are often non-deterministic, even large test suites provide only weak assurance of correctness.

Developers currently face a tension between control and safety. Low-level APIs such as CUDA and OpenCL expose the full GPU execution model but offer almost no guarantees beyond what the compiler checks. Higher-level array languages and DSLs smooth over some of the rough edges but tend to focus on performance, not verification, and often lack low-level control over features like barriers or shared memory. External verification tools such as GPUVerify [1] can prove race-freedom and barrier-divergence-freedom for compiled kernels, but they live outside the

[1]https://github.com/riyazahuja/cLean

programmer's main development loop and typically require SMT solvers and carefully crafted loop invariants.

In parallel, interactive theorem provers such as Lean have matured into practical environments for both functional programming and machine-checked proof, with projects like CompCert and Mathlib resulting in the widespread interest and formalization of verified programs and pure mathematics alike. These interactive theorem provers allow for a functional environment that allows for both the efficient authorship of programs, as well as verification of their correctness and safety, up to a mathematical specification.

As such, this project aims to bridge these two domains, presenting **cLean**: an embedded domain-specific language for GPU kernels and SIMT reasoning, written directly in Lean 4. In this section we briefly review our task, Lean 4, the safety properties we target, the verification of functional correctness, related work, and the difficulties of parallelizing Grobner bases which motivates our case study.

1.1 GPU Programming

Modern GPUs execute programs using a hierarchical parallelism model. In CUDA terminology, a *kernel* is a function that executes across a *grid* of *thread blocks*, where each block contains multiple *threads*. Note that Threads are organized into *warps* that execute the same instruction in lockstep under the SIMT model.

Each thread has access to several memory spaces with different performance characteristics and visibility:

- **Global Memory**: Visible to all threads across all blocks; large but high latency.
- **Shared Memory**: Visible to all threads within a block; small but low latency; requires explicit synchronization.
- **Local Memory**: Private to each thread; used for register spills and local variables.

With such a highly parallelizable system, the successful and efficient programming of such hardware requires a careful attention to detail in order to prevent parallelism mistakes that can degrade performance or even correctness. However, finding and debugging such edge cases can be extremely difficult, and even with robust testing infrastructure, it is possible for such bugs to fall through the cracks. As such, we turn now to formal methods in order to better prevent such errors in these critical systems.

1.2 Lean 4 Theorem Prover

Lean 4 is a functional programming language and interactive theorem prover based on dependent type theory. Lean 4 is predominantly used by mathematicians for the formal specification and proof of pure mathematics. However, Lean additionally has a robust development infrastructure, with features such as:

- **Dependent Types**: Types can depend on values, enabling precise mathematical specifications of behavior in higher-order logic.
- **Metaprogramming**: Lean's macro system allows defining custom syntax and code generation.
- **Tactic Framework**: Enables interactive proof development with automation support.

- **Compilation**: Lean code compiles directly to C++ and is self-implemented; it also can employ foreign function interfaces (FFI) to run C++ code directly in Lean without needing to serialize and transport objects in memory.

cLean leverages these features to embed a CUDA-style kernel language directly inside Lean 4. Programmers write kernels in Lean syntax; macros extract an intermediate representation (DeviceIR) that can be translated to CUDA C++ for on-device execution, and can be treated as a mathematical object for verification inside Lean.

Thus, the same source drives both performance and proof.

1.3 Safety and Functional Correctness

We target two classes of properties: Safety properties are those such as race-freedom and barrier-divergence freedom. Functional correctness is the property that a given kernel computes the mathematically intended result. Note that from the perspective of parallel algorithm design and architecture, safety is simply reasoning about legal schedules.

1.3.1 Race conditions

A data race occurs when two threads access the same memory location, at least one access is a write, and there is no ordering guarantee (e.g., synchronization) between them. Races make behavior depend on low-level interleavings, so even large test suites provide only probabilistic comfort.

For example:

```
__global__ void badSum(int* data, int* sum) {
  int idx = threadIdx.x;
  *sum += data[idx];  // DATA RACE!
}
```

In cLean, race freedom is expressed as a proposition over a symbolic semantics: for any two distinct threads and any pair of memory accesses they perform, it must not be the case that both touch the same location with at least one write.

1.3.2 Barrier divergence

CUDA exposes block-level barriers (`__syncthreads()`) that require all threads in a block to reach the barrier before any may proceed. Barrier divergence is when some threads take a control-flow path that skips a barrier while others reach it, and the program deadlocks as a result.

Barrier divergence is subtle because it depends on control flow, data-dependent conditions, and the GPU's SIMT execution model. Our verification framework tracks barrier points and checks that, under the symbolic semantics, all threads in each block reach the same multiset of barriers.

1.3.3 Functional correctness

Safety properties such as race-freedom and barrier-divergence-freedom guarantee that a kernel can execute without undefined behavior, but they do not guarantee that the kernel does the "right" thing. For many scientific and numerical kernels – including our Grobner basis case study – functional correctness is equally essential: a race-free matrix reduction that computes the wrong row echelon form is still useless.

In interactive theorem provers such as Lean, functional correctness typically takes the form of a refinement property:

- a mathematical specification describing what the kernel should compute (e.g., the mathematical function SAXPY: $r[i] = \alpha x[i] + y[i]$),
- a low-level operational semantics describes how the kernel executes (in our case, the symbolic DeviceIR semantics), and
- a theorem states that for all legal thread/block configurations and all valid inputs, running the kernel in the semantics produces a memory state that satisfies the specification.

Proving functional correctness on GPUs is challenging for three main reasons:

1. Thread interleavings mean that the kernel does not have a single sequential execution path. Correctness must hold for all legal GPU schedules.
2. Even in structured GPU code, each thread computes only a fragment of the global output. Reasoning requires decomposing the kernel into per-thread contributions and then lifting them to the global semantics.
3. Low-level details like addressing logic, bound checks, and mixed integer/float behavior interact subtly with correctness. Testing validates only concrete executions; a proof must validate all symbolic behaviors.

1.4 Related Work

GPUVerify [1] introduced the two-thread abstraction for proving race freedom and barrier divergence freedom for GPU kernels, using SMT solvers [2] and user-supplied loop invariants on compiled kernels. Our work adopts the same core abstraction, but reconstructs it inside Lean's type theory and operates on a higher-level IR generated from the DSL syntax.

Languages such as Futhark [3] and GPU-focused array DSLs like Accelerate [4] provide safer, more structured access to GPU parallelism. They offer strong guarantees about memory usage and sometimes deadlock-freedom, but do not generally allow users to express and prove arbitrary semantic properties.

Projects like CompCert [5], CakeML [6], and Vellvm [7] demonstrate that verified compilation for CPUs and low-level IRs is feasible and practical. However, these projects typically target sequential (or coarse-grained) concurrency models, and do not address GPU-specific concerns such as per-block shared memory and barrier semantics.

cLean is, to our knowledge, the first system that:

- embeds a CUDA-style kernel language directly into Lean 4,
- generates executable CUDA code, and
- simultaneously exposes a verification pipeline for both safety and functional correctness.

1.5 Grobner Basis Computation as a Parallel Workload

Grobner bases are a central tool in computational algebraic geometry for reasoning about polynomial ideals. Namely, given a set of multivariate polynomials $G_0 = \{f_1, ..., f_m\} \subset \mathbb{F}_p[x_1, ..., x_n]$ and a monomial order (such as lexicographic/alphabetical order), a Grobner basis G of the ideal $\langle G_0 \rangle$ is a canonical generating set that makes many operations -— ideal membership, elimination, dimension counting —- algorithmically tractable.

Grobner basis computation is a rich stress-test for parallelization as it concerns algebraic data structures (polynomials and ideals) that admit a matrix representation over a finite field. As such, one can formulate algorithms for computing Grobner bases that rely heavily on structured linear algebra which is very SIMD/SIMT-friendly. However, the logic surrounding

this parallelizable core (i.e. pair selection, S-polynomial construction) has very irregular dependencies and limited parallelism opportunities.

We use this setting as our main application to demonstrate that cLean supports both nontrivial parallelization and nontrivial formal reasoning.

We now outline the classical Buchberger algorithm and the F4 algorithm as baselines, with explicit inputs, outputs, and dependencies, then explain why F4's matrix-reduction step is a natural fit for parallelization by cLean.

Buchberger Algorithm (Naive Baseline)

Buchberger's algorithm [8] was the first known algorithm for computing Grobner bases, and serves a solid, naive implementation to check against. It operates on:

- A basis G, which is a set (or array) of polynomials $f \in \mathbb{F}_p[x_1, ..., x_n]$.
- A pair set P: tuples (f, g) with $f, g \in G$.
- A normal form procedure that repeatedly cancels leading terms.

Namely, given an initial generating set $G_0 \subset \mathbb{F}_p[x_1, ..., x_n]$ and a monomial ordering $\prec$, Buchberger's algorithm outputs a Grobner basis G such that $\langle G \rangle = \langle G_0 \rangle$ and G is $\prec$-Gröbner.

Algorithm 1: Buchberger

```
1:  procedure BUCHBERGER(G_0)
2:      G ← G_0
3:      P ← CriticalPairs(G)
4:
5:      while P ≠ ∅ do
6:          ▷ Select a critical pair to process
7:          (f, g) ← Select(P)
8:          P ← Remove(P, (f, g))
9:
10:         ▷ Form and reduce S-polynomial
11:         S ← Spol(f, g)
12:         r ← NormalForm(S, G)
13:
14:         if r ≠ 0 then
15:             ▷ Insert new basis element and update pairs
16:             G ← G ∪ {r}
17:             P ← P ∪ NewPairs(r, G)
18:         end
19:     end
20:     return G
21: end
```

The algorithm is necessarily sequential in nature, as although normal form computations of different S-polynomials are (approximately) independent, within each reduction, each cancellation step is dependent on the previous leading term. Moreover, as monomials are sparse and irregularly distributed, we see limited spatial locality. This motivates algorithms that push as much work as possible into structured linear algebra kernels — which is exactly what F4 does.

1.5.1 F4 Algorithm (Optimized Baseline)

The F4 algorithm [9] replaces many small, irregular polynomial reductions with fewer, large Gaussian eliminations over $\mathbb{F}_p$. As before, the algorithm concerns a basis G, but also:

- A batch B of critical pairs $B \subseteq P$.

- A symbolic dense matrix $M \in \mathbb{F}_p^{r \times c}$, where:
  - each row encodes a polynomial
  - each column corresponds to a monomial

Each F4 iteration sees a current basis G, critical pair set P, and a batch-selection strategy. We output an updated basis G' and pair set P'.

Algorithm 2: F4

```
1:  procedure F4(G_0)
2:      G ← G_0
3:      P ← CriticalPairs(G)
4:
5:      while P ≠ ∅ do
6:          ▷ Select a batch of critical pairs
7:          B ← SelectBatch(P)
8:          P ← RemoveBatch(P, B)
9:
10:         ▷ Build symbolic matrix over F_p
11:         (M, monos) ← BuildMatrix(B, G)
12:
13:         ▷ Dense modular Gaussian elimination (GPU-accelerated)
14:         M' ← RowEchelonGPU(M)
15:
16:         ▷ Extract new polynomials from reduced rows
17:         R ← ExtractPolys(M', monos)
18:
19:         for r ∈ R do
20:             if IsNonzero(r) and not LeadingTermDivisible(r, G) then
21:                 G ← G ∪ {r}
22:                 P ← P ∪ NewPairs(r, G)
23:             end
24:         end
25:     end
26:     return G
27: end
```

This algorithm provides a far more structured workload, operating on a matrix over $\mathbb{F}_p$, whose access patterns suggest a high potential for locality-based optimization. Additionally, we depend on arrays of monomials (`monos`) and coefficients for reconstruction of polynomials, as well as smaller containers for G, P and batch B.

F4 also operates in a far more structured and modular manner:

1. `BuildMatrix` first converts polynomial data into a matrix form. This has some irregularity and is CPU-bound in our implementation.
2. `RowEchelonGPU`: Gaussian elimination over $\mathbb{F}_p$ on a matrix. This is both compute-intense, and highly data-parallel.
3. `ExtractPolys`: reconstructs reduced polynomials from row-echelon form (essentially back-translation).

It is this second phase that holds a high potential for parallel optimization on each iteration, as although at the algorithm level, each F4 iteration depends on the previous basis G, within an iteration, rows of M can be processed concurrently within each elimination phase. Namely, at a kernel level, one may assign each thread (or warp) a subset of matrix rows/columns, thereby

exposing data parallelism across rows for a fixed pivot column, SIMD/SIMT-friendly access patterns, and predictable control flow with minimal divergence.

In summary, Buchberger's algorithm exposes coarse-grained task parallelism across independent S-polynomials but remains quite sequential within each reduction. F4 restructures the same algebraic computation into dense linear algebra kernels, which map naturally onto GPU hardware. Our cLean implementation then takes this one step further: we express the F4-style Gaussian elimination kernel in a Lean-embedded DSL, compile it to CUDA, and verify properties about it inside the theorem prover.

2 Approach

We now describe the design and implementation of cLean. At a high level:

- developers write CUDA-style kernels in a Lean-embedded DSL;
- a macro pipeline extracts a Device IR;
- the IR drives both CUDA code generation and a verification pipeline;
- kernels can be executed on CPU (via an interpreter) or on GPU (via generated CUDA and a launcher);
- safety and correctness properties are expressed and proved as Lean theorems.

2.1 System Architecture

cLean consists of several interconnected components, illustrated in Figure 1:

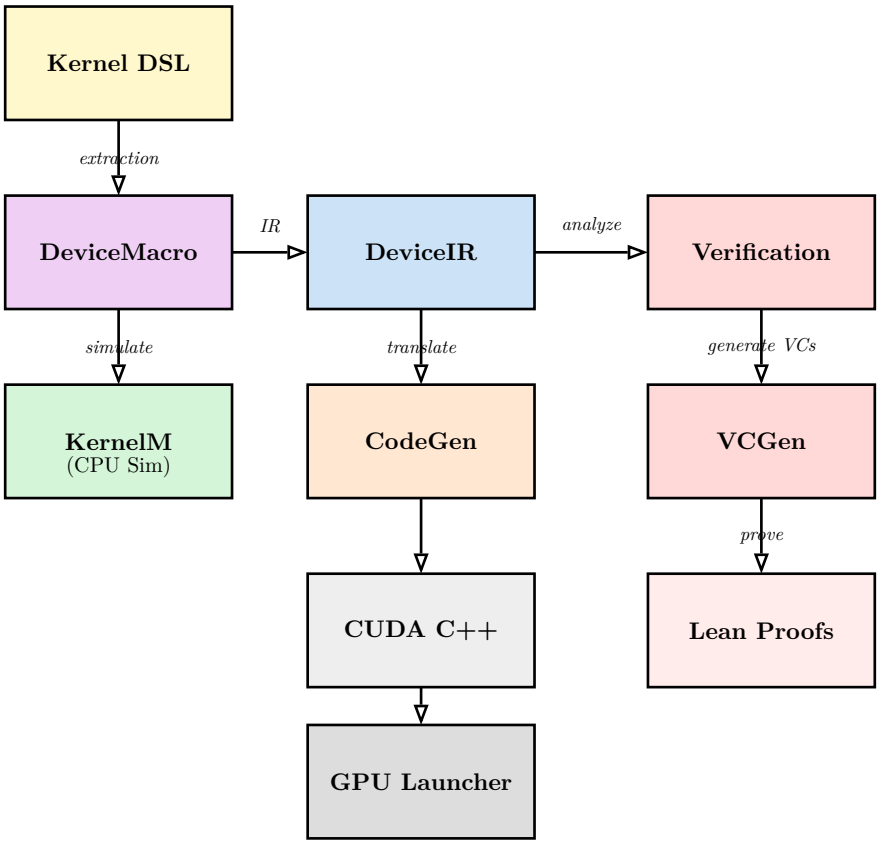

Figure 1: cLean system architecture. The kernel DSL is processed by DeviceMacro to produce both executable code (KernelM) and an intermediate representation (DeviceIR). DeviceIR is then used for CUDA code generation and formal verification.

The system comprises the following major components:

- **DeviceIR**: The intermediate representation that captures GPU kernel structure including expressions, statements, memory declarations, and control flow. For more information, see Section A.2.
- **DeviceMacro**: The syntax extraction layer that transforms Lean kernel syntax into DeviceIR through macro expansion.
- **GPU.lean**: Defines the `KernelM` monad and abstractions for CPU simulation, including `GlobalArray`, `SharedArray`, thread indexing, and barrier synchronization.
- **DeviceCodeGen**: Translates DeviceIR into CUDA source code.
- **ProcessLauncher**: Manages kernel compilation via `nvcc` and execution through a C++ GPU launcher.
- **Verification Framework**: Implements the two-thread abstraction and verification condition generation for safety and correctness properties.

2.2 GPU execution

The GPU execution path consists of three steps:

1. **Code generation**: Device IR is translated to CUDA C++, including:
   - type mapping from IR types to CUDA types;
   - expression translation for arithmetic, logic, and thread intrinsics;
   - structured control flow in CUDA syntax;
   - careful handling of integer overflow in finite field operations via 64-bit intermediates.
2. **Compilation**: the generated CUDA file is compiled with `nvcc` to PTX. A simple hashing scheme caches compiled kernels to allow recompilation-free interactive development.
3. **Launching and marshalling**: Lean calls out to a foreign C++ launcher to:
   - allocates GPU buffers and transfers input data;
   - launches the kernel with specified grid and block dimensions;
   - retrieves results and returns a (pointer to a) Lean structure type containing the modified buffers.

This design keeps the Lean side simple and robust, with minimal overhead.

A CPU interpreter is also included to run the same Device IR under a symbolic "GPU-like" memory model, including shared memory semantics – allowing for the quick debugging of kernels off-device.

2.3 Verification Framework

2.3.1 Overview

The verification framework operates on an enriched IR derived from Device IR. It extracts:

- all memory accesses, annotated with space (global/shared/local), access type (read/write), and index expressions;
- barrier points and control-flow structure;
- a symbolic description of thread IDs and grid/block configuration.

From this, we generate a set of verification conditions that, if provable in Lean, imply that the original kernel is race-free, barrier-divergence-free, and (optionally) functionally correct.

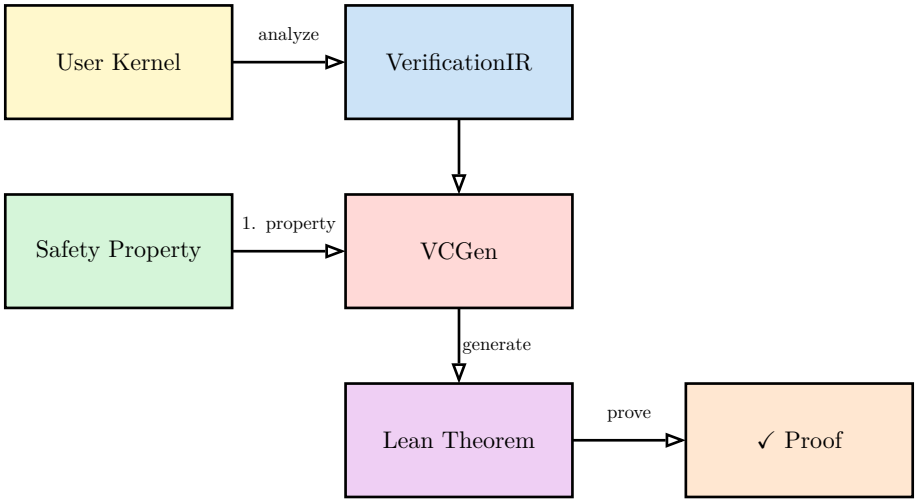

Figure 2: Verification workflow: The kernel is analyzed to produce VerificationIR, which is combined with a safety property to generate verification conditions as Lean theorems that the user proves.

2.3.2 VerificationIR

As previously mentioned, we enrich DeviceIR with the following metadata:

```
structure MemoryAccess where
  name : String
  space : MemorySpace  -- global | shared | local
  accessType : AccessType  -- read | write
  index : Option DExpr
  location : Nat  -- Program counter

structure VerifiedKernel where
  context : VerificationContext
  accesses : List MemoryAccess
  barriers : List BarrierPoint
```

This autogenerated intermediate representation allows for users to reason not only about the operations and control flow of the program, but memory and program semantics as well. Namely, we analyze memory access patterns symbolically rather than reasoning about concrete addresses, characterizing how thread IDs map to memory locations.

```
inductive AddressPattern where
  | identity                    -- tid ↦ tid
  | constant (n : Nat)          -- tid ↦ n
  | offset (base : Nat)         -- tid ↦ tid + base
  | linear (scale off : Nat)    -- tid ↦ scale*tid + off
  | symLinear (scale off : SymValue)  -- symbolic coefficients
```

Note that we say an access pattern is *injective* if distinct thread IDs produce distinct addresses. If both threads' accesses follow injective patterns with the same array, they access different locations and cannot race.

The relationship between address patterns and race freedom is illustrated below:

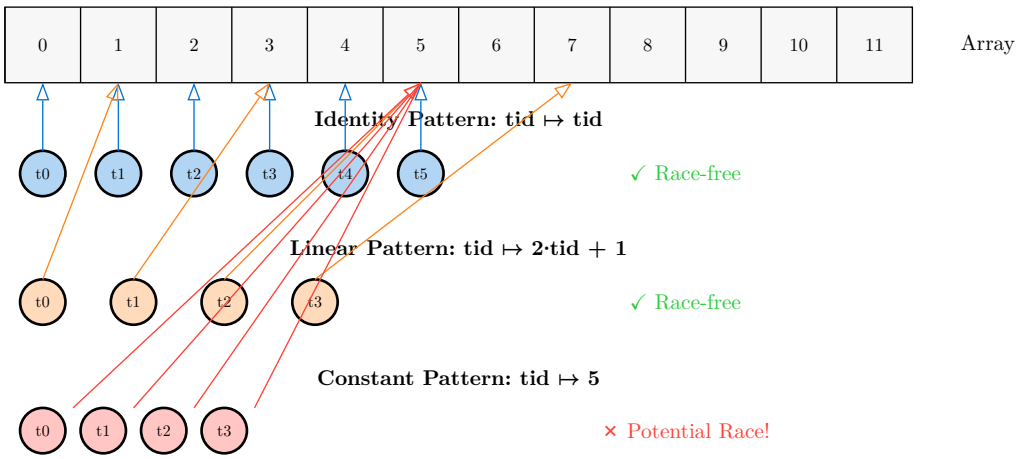

Figure 3: Different address patterns and their race-freedom properties. Injective patterns (identity, linear with non-zero scale) guarantee no conflicts; constant patterns create races when writes are involved.

2.3.3 Two-Thread Reduction

The verification condition generator implements the two-thread abstraction:

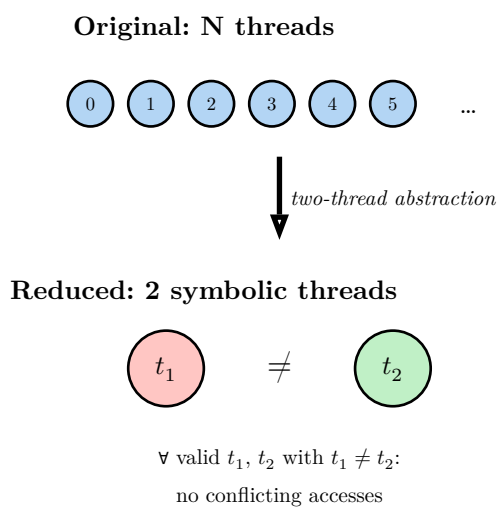

Figure 4: Two-thread reduction: Instead of reasoning about N threads, we prove that any two arbitrary distinct threads don't interfere.

This module performs this reduction and generates verification conditions by:

1. Dualizing the kernel: creating symbolic representations of two threads
2. Collecting all memory accesses tagged by thread
3. Generating assertions that conflicting accesses imply same thread

This two-thread abstraction is sound for race-freedom because:

Theorem (Two-Thread Soundness): *If a kernel K is race-free under the two-thread semantics (for all pairs of distinct threads $t_1 \neq t_2$), then K is race-free under the full N-thread semantics.*

Proof: Suppose K has a data race under the N-thread semantics. Then there exist threads t_i and t_j with $t_i \neq t_j$ that have conflicting accesses (same location, at least one write, no synchronization). But then the two-thread analysis with $t_1 = t_i$ and $t_2 = t_j$ would detect this race, contradicting our assumption. □

This reduction is crucial for tractability: instead of reasoning about $O(N^2)$ potential race pairs among N threads, we reason about a single symbolic pair many times with different inputs.

2.3.4 Race Freedom

To reason about race freedom, we track each memory access in the kernel:

- which array or variable it touches;
- whether it is a read or write;
- a symbolic expression for its index;
- the phase of execution in which it occurs.

Moreover, the kernel body is divided into phases separated by barriers. Within a single phase, threads run independently; races are only possible between accesses in the same phase. Across phases, synchronization guarantees an ordering, so those accesses cannot race.

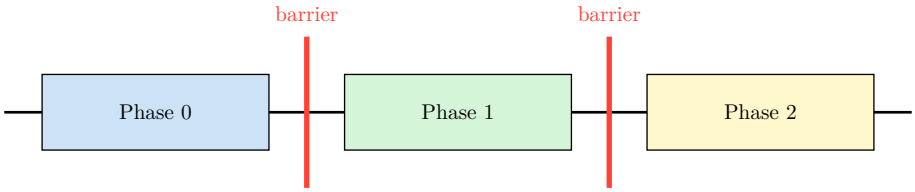

Figure 5: Barrier-separated phases: Memory accesses across different phases cannot race because barriers enforce synchronization.

The race-freedom condition for a phase can be stated informally as follows: for any two distinct threads $t_1 \neq t_2$, and any two memory accesses in this phase that touch the same array with at least one write, the index expressions must evaluate to different locations for t_1 and t_2. Formally, this is represented in Lean as:

```
def RaceFree (k : VerifiedKernel) : Prop :=
  ∀ (env : Environment),
  ∀ t1 t2 : GlobalThreadId,
    k.context.inBounds t1 →
    k.context.inBounds t2 →
    ∀ acc1 acc2 : MemoryAccess,
      acc1 ∈ k.accesses →
      acc2 ∈ k.accesses →
      acc1.name = acc2.name →
      (acc1.isWrite ∨ acc2.isWrite) →
      sameIndex acc1 acc2 t1 t2 env →
      (t1 ≠ t2 → False)
```

Listing 3: Formal statement of race-freedom

2.3.5 Barrier divergence

Barrier divergence is modeled by tracking, for each barrier, which threads can reach it along any control-flow path. At a high level, we require that for each barrier instance, all threads

in a block that are in-bounds for the kernel configuration must reach that barrier under the symbolic semantics.

The verification IR records barrier sites, and the verification conditions ensure that no path allows some threads to pass the barrier while others are stuck before it. For many structured kernels, this reduces to checking that all barriers are syntactically aligned (e.g., not nested inside divergent branches without matching counterparts). Our framework encodes this structurally.

```
def BarrierDivergenceFree (k : VerifiedKernel) : Prop :=
  ∀ b : BarrierPoint,
    b ∈ k.barriers →
    ∀ (t : GlobalThreadId),
      k.context.inBounds t →
      threadReachesBarrier t b
```

Listing 4: Formal statement of barrier divergence freedom

2.3.6 Functional correctness

cLean's functional correctness pipeline is built directly on top of the symbolic execution semantics, and leverages Lean's dependent type theory to treat the kernel's VerificationIR body as a mathematical object whose behavior is represented by pure functions over symbolic memory. Unlike real CUDA execution, this semantics are:

- deterministic (no interleavings),
- side-effect-free (memory is an immutable function `Name → Nat → VVal`), and
- exact (all floating-point values are treated as rational numbers).

Namely, memory is modeled as a higher order function `VMem: Name → Nat → VVal`, meaning that objects like arrays are simply pure functions indexed by names and numeric locations. Symbolic threads are armed internal context `VCtx` containing its thread/block indices and a map of locals, allowing for the simplification of expressions like: `globalIdxX = blockIdx.x * blockDim.x + threadIdx.x`.

Building on this, the central evaluator is also given as a higher order function: `vEvalStmt : DStmt → VCtx → VMem → (VCtx × VMem)`. It simply performs a structural recursion over DeviceIR statements. Important properties of the evaluator is that assignments update locals, stores produce a new memory function, and control flow is resolved via symbolic conditions, not concrete branching.

To obtain full functional correctness, we symbolically execute the kernel across all logical threads: `vExecKernel1D : DStmt → Nat → Nat → VMem → VMem` and show that the final memory satisfies the global specification.

Three design choices make this tractable:

- Rationals instead of Float: floating-point arithmetic is non-associative and not bit-reproducible across architectures; correctness proofs require exact arithmetic.
- Pure memory model: updates are tracked symbolically and are easy to reason about using extensionality.
- Thread-local contexts: thread indices remain symbolic, allowing proofs that work for all possible grid/block configurations.

This yields a verification framework that is dramatically simpler than reasoning directly about CUDA C++ or PTX, yet still faithful to the kernel's control structure and index arithmetic.

Our current semantics do not as robustly support reasoning over complex barrier and shared memory operations, but we hope to pursue this direction in future work.

2.4 Grobner Basis Case Study

To demonstrate that cLean supports not only verified GPU programming but also nontrivial algorithmic parallelization, we implemented a GPU-accelerated Grobner basis computation by modifying the F4 approach. Namely, we study how an irregular symbolic algorithm (Buchberger/F4) exposes structured parallelism, how we reorganize the algorithm to fit the GPU SIMT model, and how we design+verify a complex GPU kernel in cLean.

As noted earlier, although Grobner basis computation appears symbolic, the F4 algorithm reveals a deeply data-parallel structure. Namely, each F4 iteration consists of:

1. Selecting a batch of critical pairs.
2. Generating S-polynomials and reducers.
3. Symbolically constructing a large coefficient matrix over $\mathbb{F}_p$.
4. Performing dense Gaussian elimination over F_p.
5. Converting reduced rows back into polynomials.

Steps (1), (2), and (5) exhibit algorithmic irregularity — control-heavy, CPU-friendly tasks.

Steps (3) and especially (4) dominate the asymptotic runtime in all Grobner basis literature (often 70–95%). This dense Gaussian elimination is highly structured and SIMD-friendly — making it the natural target for our GPU acceleration.

In our implementation, we explicitly parallelize step (4) – the computational bottleneck – preserving algorithm correctness while ensuring a clean separation between irregular symbolic logic and structured numerical kernels.

Namely, we employ the same data structures as the F4 algorithm, defining the core matrix with a row-major 1D array, polynomials as arrays of `(coeff : FpElem, mono : Array Nat)` pairs, and store a monomial basis as a sorted list of all monomials.

This representation enables efficient coalesced row-wise traversal and matches the memory patterns of Gaussian elimination. Our first few implementations used sparse matrix implementations, but the pointer chasing ended up significantly decreasing overall performance.

Now, although each iteration's Gaussian elimination step is extremely parallelizable, it does rely on a global sequential dependence as the pivot for column `k` must be established before eliminating column `k+1`.

However, we note that all rows can be updated independently for a fixed pivot via:

$$\forall i \neq \text{pivot}, \text{row}_{i[j]} = \text{row}_{i[j]} - \text{factor}_i \cdot \text{pivotRow}[j] \bmod p \tag{1}$$

Namely, each row update depends only on its private factor value, and the shared pivot row. Moreover, within a row i, for all columns j, $\text{row}_{i[j]}$ updates are independent. This yields a clean two-dimensional parallel structure. However, unlike initial attempts at this parallelization, rather than assigning a thread to each element's update, we exploit only the row dimension, assigning one thread per row, thereby maximizing coherence and reducing overhead. Explicitly:

- Grid: 1D grid of blocks covering all matrix rows.
- Block size: 256 threads (empirically good occupancy).
- Thread `t` updates row `t` of the matrix.
- Row-major memory layout: contiguous per-thread traversal implies coalesced loads/stores.

And in pseudocode:

Algorithm 3: GaussElimKernel

```
1:  procedure GaussElimKernel(nrows, ncols, pivotRowIdx, p, mat, pivotRow, factors)
2:      ▷ Each GPU thread handles a single row
3:      row ← globalIdxX()
4:
5:      if row < nrows then
6:          ▷ Skip the pivot row itself
7:          if row ≠ pivotRowIdx then
8:              factor ← factors[row]
9:              ▷ Skip rows with zero factor
10:             if factor ≠ 0 then
11:                 ▷ Loop over all columns in this row
12:                 for col ← 0; col < ncols; col ← col +1 do
13:                     idx ← row · ncols + col
14:                     matVal ← mat[idx]
15:                     pivotVal ← pivotRow[col]
16:                     ▷ Modular multiply and subtract: mat[row,col] -= factor · pivotRow[col]
                        (mod p)
17:                     prod ← (factor · pivotVal) mod p
18:                     diff ← ((matVal - prod) mod p + p) mod p
19:                     mat[idx] ← diff
20:                 end
21:             end
22:         end
23:     end
24: end
```

The GPU thus acts as a wide SIMD engine applying the same arithmetic to many rows on each iteration of F4. However, that is not to say that this is identical to the serial algorithm, as the serial elimination performs pivot search, scaling, and factor computation **inline** with row updates. In comparison, we refactor into CPU+GPU phases: the initial stages of pivot search, row scaling, and factor computation occur on CPU, then we switch to the GPU phase to apply elimination factors to all rows, and we return to CPU to advance the pivot index for the next iteration. This allows for a large reduction in GPU kernel complexity, simpler verification, and mirrors standard hybrid CPU+GPU system designs.

In the development of this parallelization strategy, we experimented with several designs, such as staging the pivot row in shared memory. This reduced global memory traffic, but overwhelmingly increases overhead due to barrier usage and general synchronization overhead. Additionally, we also naively attempted a 2D allocation of threads to every element, but this was too fine-grained and resulted in excessive global-memory traffic. The `PivotRow` was reused inefficiently and the net performance was slower than our CPU baseline. As such, we eventually converged to the final implementation, where we kept the most simple approach: directly reading the pivot row from global-memory; the minimal synchronization and benefits of caching allows for high performance and a simple access pattern.

As such, we identified the true computational bottleneck of modern Grobner basis algorithms, extracted a structured, data-parallel kernel from an otherwise symbolic algorithm, mapped this kernel carefully to GPU threads/blocks, and integrated it into an end-to-end algebraic pipeline in Lean.

The result is a demonstrably parallel, nontrivial, mathematically meaningful computation running inside a formally verified DSL on-device.

3 Results

In this section we outline how we evaluate cLean as both a programming system and a verification framework. We describe the experimental setup, the benchmark structure, and the dimensions along which we compare CPU, hand-written CUDA, and cLean-generated kernels, as well as the Grobner parallelization case study.

3.1 Experimental Setup

Our experiments cover two core evaluations: first, we test on a fixed dataset of CUDA kernels derived from the Nvidia CUDA Toolkit 13.0 [10]. Second, we evaluate our Parallel Grobner Basis implementation.

On our core experiments, we evaluate along two axes:

- *Performance:* end-to-end runtime, breakdown, hardware utilization, and scaling behavior across input sizes.
- *Verification:* total coverage and expressivity, against a GPUVerify baseline [1]. We maintain a 60s timeout.

In sum, we consider:

- vector-style kernels (map/reduce, stencil-like operations);
- simple matrix kernels (matrix-vector and small matrix-matrix operations);
- the Gaussian elimination kernels used in the Grobner pipeline.

	Kernel	**Pattern**	**Description**	**Key Features**
1	VectorAdd	Map	$C[i] = A[i] + B[i]$	Basic parallel, 3 arrays
2	VectorSquare	Map	$A[i] = A[i]^2$	In-place modification
3	SAXPY	Map	$Y = \alpha X + Y$	Scalar parameter, FMA
4	ScalarMul	Map	$A[i] * = s$(loop)	Per-thread loop
5	MatrixTranspose	Stencil	$B[j, i] = A[i, j]$	2D blocks, coalescing
6	MatrixMul	Stencil	$C = A \times B$	2D blocks, accumulation
7	TemplateScale	Map	$O =$ blockDim $\times I$	Intrinsic usage
8	ExclusiveScan	Scan	Prefix sum	Per-thread accumulation
9	DotProduct	Reduce	$\sum A[i]B[i]$	Partial sums, reduction

Table 1: Benchmark kernels used in evaluation. Patterns include map, stencil, scan, and reduce; features test a range of parallel idioms and memory access patterns.

Hardware Configuration:

- GPU: NVIDIA RTX A6000 GPU (48GB VRAM, 10752 CUDA Cores)
- CPU: AMD EPYC 7763 64-Core CPU (3.5 GHz base)
- Host memory: 64GB

Software Configuration:

- CUDA Toolkit 12.5

- Lean 4 (v4.20.1)
- `nvcc` with `-O3` optimizations

3.2 Benchmark

3.2.1 Performance

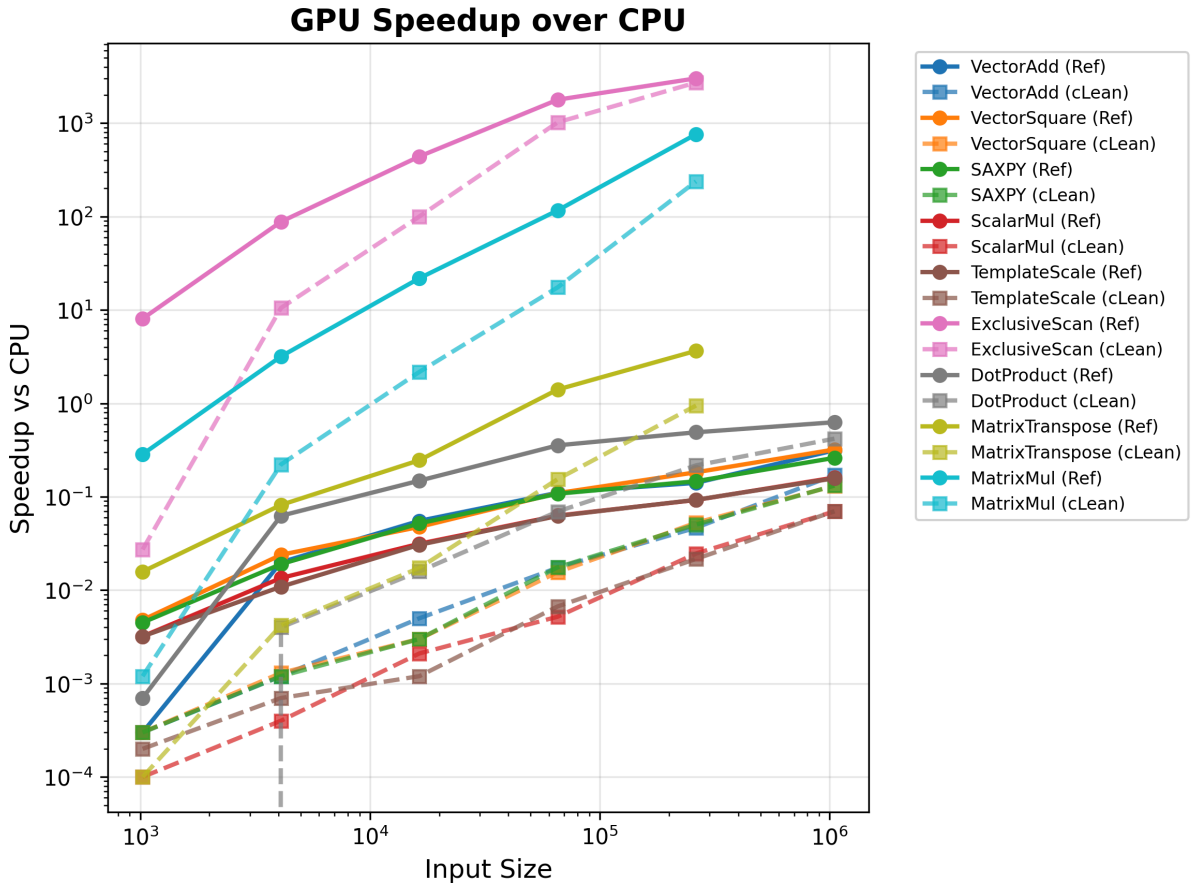

Figure 6: Net speedup of both reference CUDA and autogenerated cLean compared to a naive CPU baseline

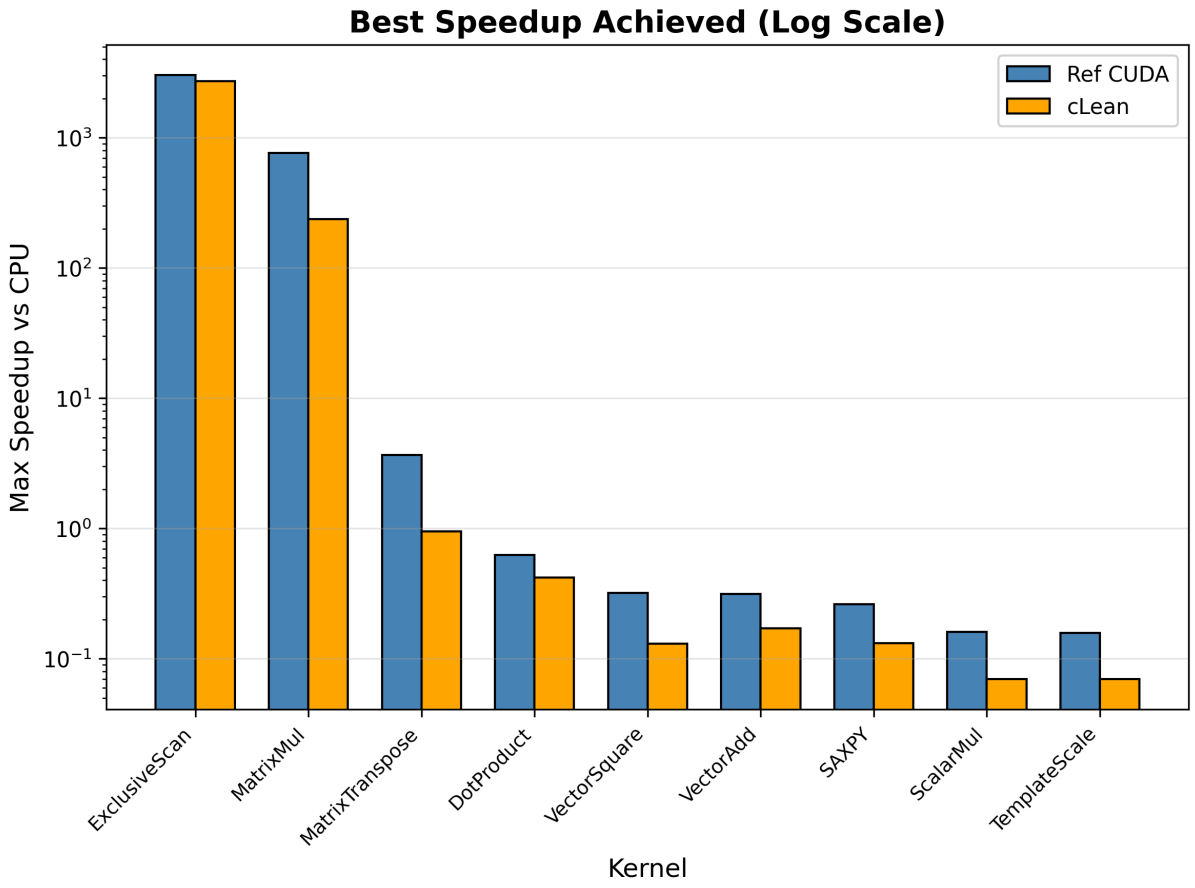

Figure 7: Best speedup of both reference CUDA and autogenerated cLean compared to a naive CPU basline, across all test kernels.

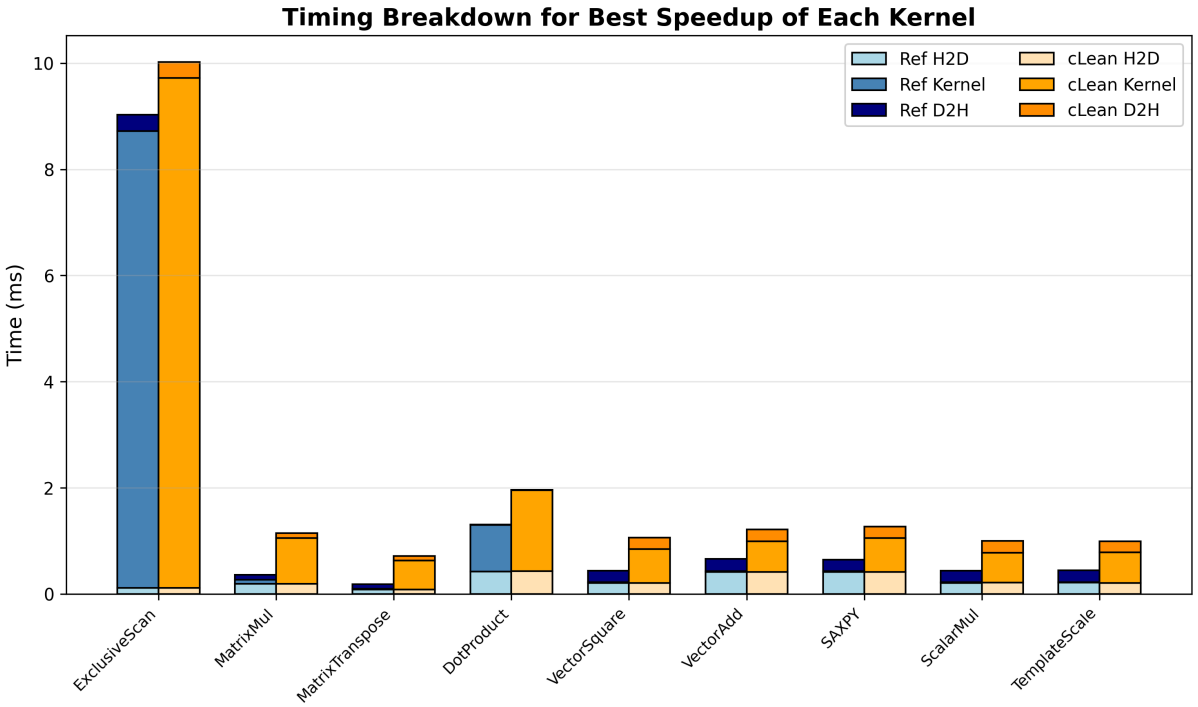

Figure 8: Timing breakdown of both reference CUDA and autogenerated cLean across all test kernels.

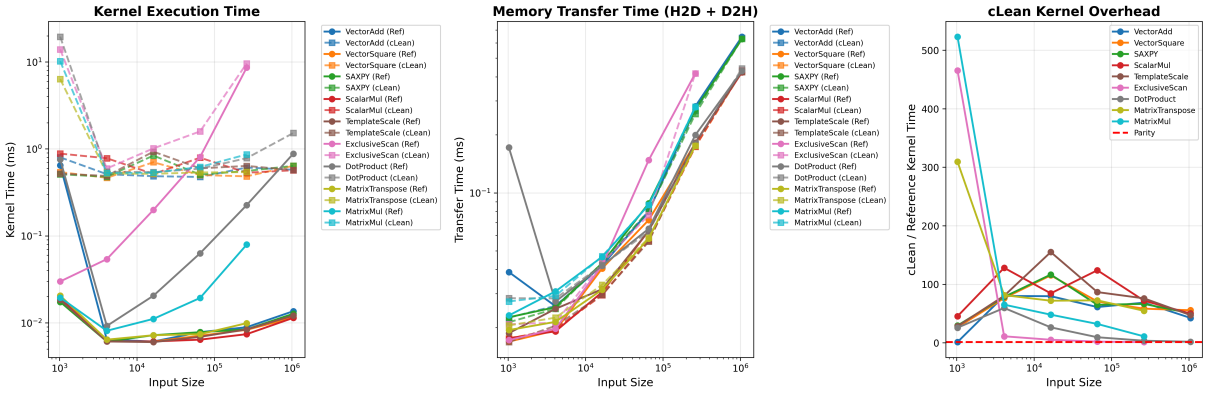

Figure 9: Overhead evaluations of both reference CUDA and autogenerated cLean across test kernels.

The benchmark results exhibit three main trends:

1. cLean GPU kernels outperform the CPU interpreter for sufficiently large inputs. As shown in Figure 6, all map-style kernels (VectorAdd, VectorSquare, SAXPY, ScalarMul) and stencil kernels (MatrixTranspose, MatrixMul) achieve substantial speedups once the input size exceeds a few tens of thousands of elements. For small problem sizes, the fixed cost of compilation and kernel launch dominates, so the CPU interpreter can be competitive or faster; beyond the "crossover" point, the GPU's higher arithmetic throughput and memory bandwidth outweigh this overhead.
2. Best-case speedup is driven by arithmetic intensity and regularity. Figure 7 shows that the highest speedups occur for kernels with regular, coalesced access patterns (VectorAdd, SAXPY, MatrixTranspose), simple control flow without divergence, and minimal host-side pre/post-processing. Kernels with more irregular patterns (ExclusiveScan, DotProduct) still benefit from GPU execution, but their speedups are tempered by general overhead, partial reductions, or global memory traffic that is harder to coalesce.
3. Kernel runtime is only part of the cost; compilation and marshalling matter. The timing breakdown in Figure 8 and Figure 9 separates host–device transfers, actual kernel execution, and Lean-side marshalling. For large inputs, kernel execution dominates, and cLean's performance approaches that of the corresponding hand-written CUDA samples. For moderate inputs, compilation and marshalling still represent a noticeable fraction of total time, suggesting room for engineering improvements. We also see that cLean's DeviceIR and runtime metadata introduce a modest overhead in host memory compared to the raw CUDA samples. This overhead is primarily due to the symbolic representations of kernels and additional Lean data structures for simulation and launch.

In practice, this overhead is negligible compared to GPU memory usage for the workloads we consider, but it is a trade-off a user pays to gain integrated verification and an interactive proof environment.

3.2.2 Verification

The table compares the verification coverage of cLean against GPUVerify under a 60s timeout:

Kernel	Race-Freedom	Divergence Freedom	Functional Correctness
VectorAdd	✓ /✓	✓ /✓	✓ /×
VectorSquare	✓ /✓	✓ /✓	✓ /×

Kernel	Race-Freedom	Divergence Freedom	Functional Correctness
SAXPY	✓ /✓	✓ /✓	✓ /×
ScalarMul	✓ /✓	✓ /✓	✓ /×
MatrixTranspose	✓ /✓	✓ /✓	× /×
MatrixMul	✓ /×	✓ /✓	× /×
TemplateScale	✓ /✓	✓ /✓	✓ /×
ExclusiveScan	✓ /×	✓ /✓	✓ /×
DotProduct	✓ /✓	✓ /✓	✓ /×

Table 2: Verification coverage over benchmark kernels with a 60s timeout. Entries are of the form [cLean verified]/[GPUVerify verified]. cLean properly extends the capabilities of GPUVerify.

For race freedom and barrier divergence, cLean matches or exceeds GPUVerify on all benchmark kernels. This is expected: our two-thread abstraction is essentially imported from GPUVerify's design, but applied to a higher-level IR that retains structured information about arrays, indices, and barriers.

For some kernels (e.g., MatrixMul, ExclusiveScan), GPUVerify times out (though given 120s, it successfully discharges all verification conditions), largely due to complex loop nests and the need for carefully crafted invariants. cLean's verification pipeline can instead lean on structured DeviceIR plus Lean tactics, automatically reasoning about index arithmetic and address patterns rather than manually encoding invariants in SMT logic.

For functional correctness, we currently interactively verify full correctness only for simpler kernels (VectorAdd, VectorSquare, SAXPY, ScalarMul, etc.), and treat more complex kernels (matrix multiplication, transpose) as future work. This is reflected by the consistent ✓ / × entries: cLean provides the introductory machinery to state and prove correctness theorems, and future work will extend this engineering effort for all benchmarks.

Overall, these results show that cLean: recovers the safety guarantees of GPUVerify for a suite of realistic kernels, extends coverage to kernels where GPUVerify struggles, and offers a path toward full functional correctness proofs that are out of scope for existing GPU verification tools.

3.3 Grobner Basis Computation

3.3.1 Performance

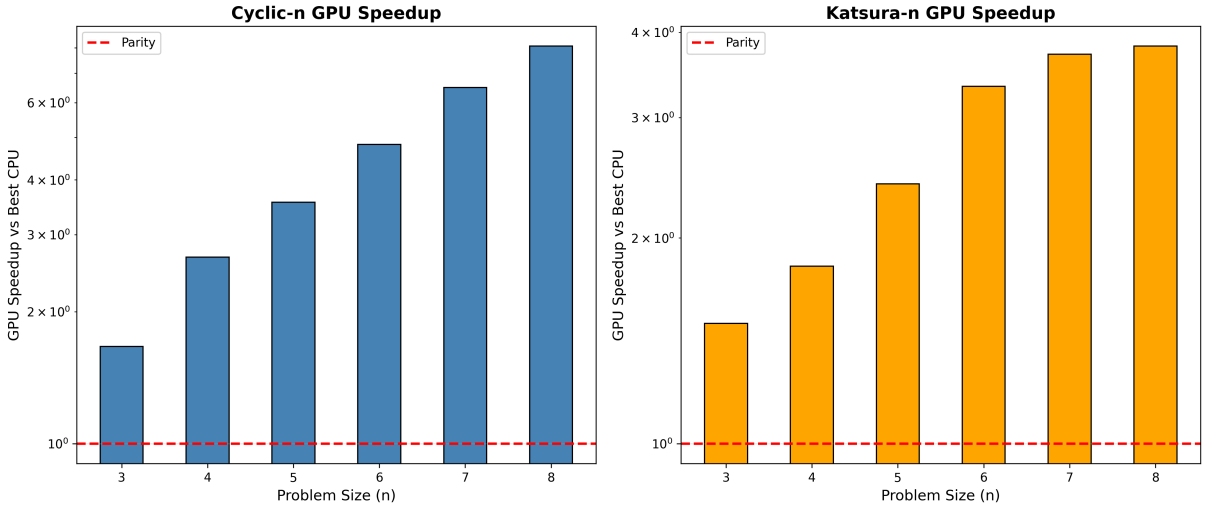

Figure 10: cLean vs F4 CPU speedup

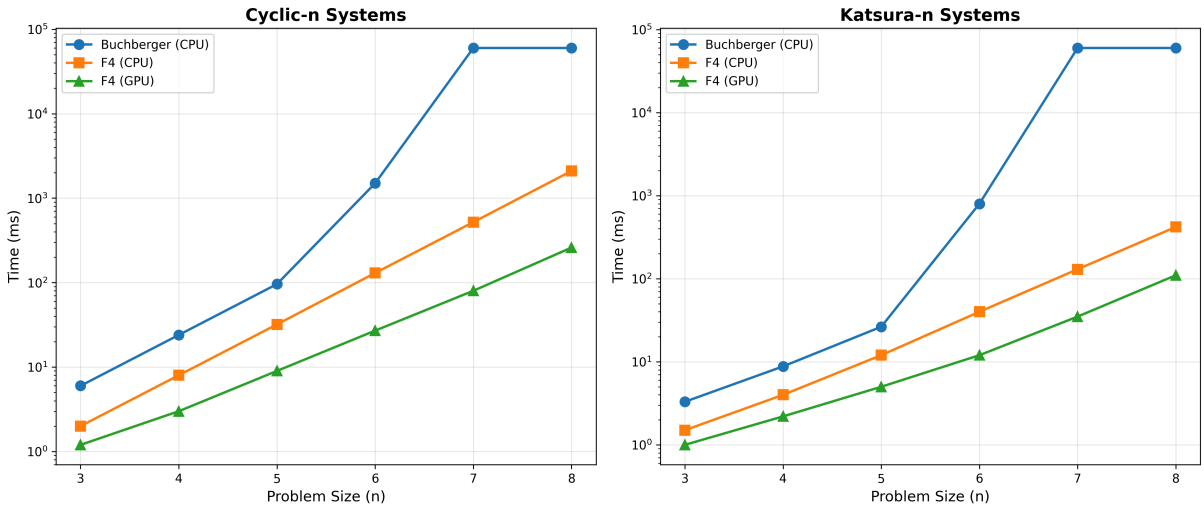

Figure 11: cLean vs F4 CPU vs Buchberger runtime

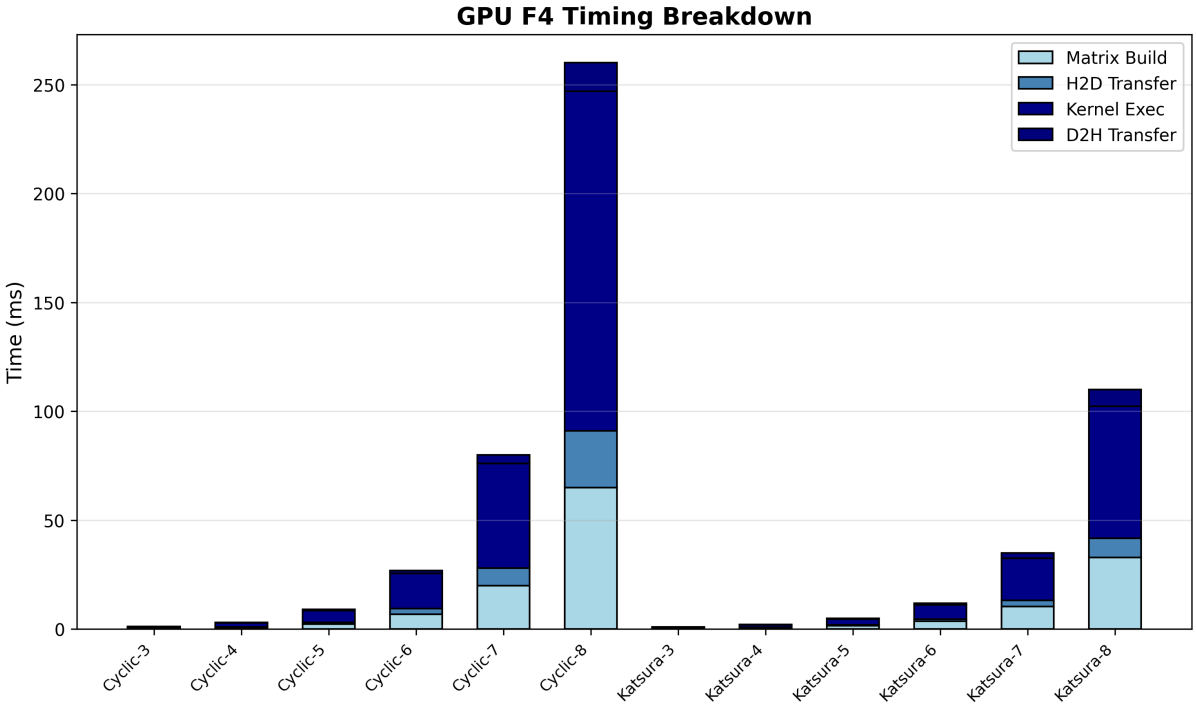

Figure 12: cLean runtime breakdown

The Grobner basis experiments focus on the F4-style dense Gaussian elimination phase, evaluated on standard benchmark families such as Cyclic-n and Katsura-n systems.

As we can see in Figure 11, for small matrices (e.g., low-degree Cyclic-3/Katsura-3 instances), CPU and GPU runtimes are comparable, and sometimes the CPU baseline wins due to kernel launch and memory allocation overhead.

As matrix sizes grow (higher n in Cyclic and Katsura families), the GPU-accelerated version increasingly dominates, as the elimination step becomes the main runtime contributor in the CPU baselines.

The cLean GPU implementation tracks the performance of optimized CPU code closely, consistently remaining significantly faster.

Moreover, Figure 10 aggregates speedups of the cLean GPU variant over the naive CPU elimination. For larger Grobner instances, the GPU-accelerated elimination provides substantial speedups, which translate directly into reduced end-to-end F4 runtime because elimination dominates total cost.

Finally, Figure 12 decomposes the GPU-enabled F4 runtime into:

- CPU symbolic phases (pair selection, S-polynomial construction, matrix building),
- host–device transfers,
- GPU elimination kernels, and
- device-host transfer and result reconstruction.

Even in this more complex pipeline, GPU computation accounts for the majority of time spent in the bottleneck phase, with data transfer and orchestration remaining a smaller—but non-negligible—overhead. This confirms that our design choice to offload only the dense linear algebra core is well-aligned with classic profiling results for F4 and related algorithms.

As such, the Grobner case study illustrates that:

- cLean can handle nontrivial, domain-specific workloads beyond synthetic kernels,
- the performance gap between cLean-generated CUDA and hand-written code is small for this structured kernel, and
- the system achieves meaningful speedups over well-optimized CPU baselines in an algebraic context where correctness really matters.

3.3.2 Verification

We provide the formal proof of safety and additional qualitative examples in Section A.3.

4 Conclusion

cLean demonstrates that formal verification of GPU kernels is practical within a general-purpose theorem prover. By embedding our DSL in Lean 4, we achieve:

- Familiar, CUDA-inspired syntax for GPU programmers.
- Dual execution on CPU (for testing) and GPU (for performance) from a single source of truth.
- Rigorous safety guarantees through machine-checked race-freedom and divergence-freedom proofs.
- A path to full functional correctness for kernels where correctness is as important as performance.

Our type-theory focused IR synthesis enables tractable verification while remaining sound for the safety properties we target. The Grobner basis case study shows that verified kernels can achieve substantial speedups over sequential implementations and integrate into realistic algebraic pipelines.

Looking forward, we see cLean as a stepping stone toward a broader vision for safe, robust, and high-performance GPU programming. We hope to extending cLean with more aggressive optimizations, richer semantics, and higher-level verified libraries in order to reduce the gap between verified and hand-tuned GPU code and make verified high-performance computing more accessible in practice.

Bibliography

[1] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Deligiannis, "GPUVerify: a verifier for GPU kernels," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 113–132, 2012.

[2] H. Barbosa *et al.*, "cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, in Lecture Notes in Computer Science, vol. 13243. Springer, 2022, pp. 415–442. doi: 10.1007/978-3-030-99524-9_24.

[3] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 556–571.

[4] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating Haskell Array Codes with Multicore GPUs," in *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, 2011, pp. 3–14.

[5] X. Leroy, "Formal Verification of a Realistic Compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009, doi: 10.1145/1538788.1538814.

[6] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A Verified Implementation of ML," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014, pp. 179–191. doi: 10.1145/2535838.2535841.

[7] Z. Zhao, S. Weirich, and others, "Vellvm: A Verified Formalization of the LLVM Intermediate Representation," in *Proceedings of the 1st Summit on Advances in Programming Languages (SNAPL)*, 2012.

[8] B. Buchberger, "An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal," *Journal of Symbolic Computation*, vol. 41, no. 3–4, pp. 475–511, 2006.

[9] J.-C. Faugère, "A new efficient algorithm for computing Gröbner bases (F4)," in *Journal of Pure and Applied Algebra*, Elsevier, 1999, pp. 61–88.

[10] NVIDIA Corporation, "NVIDIA CUDA Toolkit, version 13.0." 2025.

APPENDIX A

A.1 Implementation Details

A.1.1 DeviceIR

A.2 DeviceIR: Intermediate Representation

The DeviceIR captures the essential structure of GPU kernels in a form suitable for both code generation and verification. The key data types are:

```
inductive DType where
  | int | nat | float | bool
  | array (elemType : DType)
  | tuple (types : List DType)
  | struct (name : String) (fields : List (String × DType))

inductive BinOp where
  | add | sub | mul | div | mod
  | lt | le | gt | ge | eq | ne
  | and | or

inductive DExpr where
  | intLit (n : Int) | floatLit (f : Float) | boolLit (b : Bool)
  | var (name : String)
  | binop (op : BinOp) (e1 e2 : DExpr)
  | unop (op : UnOp) (e : DExpr)
  | index (arr : DExpr) (idx : DExpr)
  | threadIdx (dim : Dim) | blockIdx (dim : Dim)
  | blockDim (dim : Dim) | gridDim (dim : Dim)

inductive DStmt where
  | skip
  | assign (name : String) (expr : DExpr)
  | store (arr : String) (idx : DExpr) (val : DExpr)
  | seq (s1 s2 : DStmt)
  | ite (cond : DExpr) (thenBranch elseBranch : DStmt)
  | for (var : String) (lo hi : DExpr) (body : DStmt)
  | whileLoop (cond : DExpr) (body : DStmt)
  | barrier

structure Kernel where
  name : String
  params : List VarDecl
  globalArrays : List ArrayDecl
  sharedArrays : List ArrayDecl
  body : DStmt
```

Listing 5: Core DeviceIR data types for representing GPU kernel structure.

These kernels are authored in a monadic style in Lean and expanded to Device IR via macros. For example:

```
kernelArgs SaxpyArgs(n: Nat, alpha: Float)
  global[x y result: 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 result : GlobalArray Float := ⟨args.result⟩

  let idx ← globalIdxX
  if idx < n then do
    let xVal ← x.get idx
    let yVal ← y.get idx
    result.set idx (alpha * xVal + yVal)

#print saxpyKernelIR
/- Outputs:

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 := DeviceIR.MemorySpace.global },
      { name := "y", ty := DType.float.array, space := DeviceIR.MemorySpace.global },
      { name := "r", ty := DType.float.array, space :=
DeviceIR.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) }
-/

#eval kernelToCuda saxpyKernelIR
/-
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);
  }
```

```
}
-/
```

Listing 6: Example Kernel, generated IR, and generated CUDA code.

This IR is then used to generate CUDA source code, prove functional correctness, and ensure formally-verifiable safety conditions.

A.2.1 CPU Simulation with Barrier Semantics

The CPU simulator faithfully models GPU execution semantics, including barrier synchronization. The key challenge is simulating parallel barrier behavior in a sequential execution:

```
structure KernelState where
  shared  : HashMap Name KernelValue
  globals : HashMap Name KernelValue
  writeBuffer : WriteBuffer  -- Buffered writes
  isBuffering : Bool
  currentPhase : Nat
  threadBarrierCount : Nat
  hitBarrier : Bool
```

The simulation executes in *phases*:

1. All threads execute until they hit a barrier
2. Buffered writes are applied atomically
3. All threads continue to the next barrier

This phased execution correctly models the synchronization behavior of real GPU barriers in many empirical cases. Future work will aim to make this simulation more robust and accurate.

A.2.2 Handling Integer Overflow

A critical implementation detail is avoiding integer overflow in modular arithmetic. For finite field computations over $\mathbb{Z}_p$ where $p = 65521$ (the largest prime less than 2^{16}), products can exceed 32-bit integers:

$$65520 \times 65520 = 4,292,870,400 > 2^{32} - 1 \tag{2}$$

Our code generator inserts appropriate casts:

```
| .binop .mul a b =>
    s!"((long long){exprToCuda a} * {exprToCuda b})"
```

This ensures that intermediate products are computed in 64-bit precision before being reduced modulo p.

A.2.3 Type Inference for Parameters

The macro system infers CUDA types for kernel parameters based on naming conventions:

```
def inferCudaType (varName : String) : String :=
  if varName == "N" || varName.endsWith "Idx" then "int"
  else if varName == "p" || varName == "prime" then "int"
  else if varName.endsWith "Row" || varName.endsWith "Col" then "int"
  else "int"  -- Default to int for safety
```

This heuristic approach handles common patterns while allowing explicit type annotations when needed.

A.2.4 Data Marshalling

Communication between the Lean process and GPU launcher uses Lean's FFI system to pass objects from host to device and vice versa. We employ the Lean FFI to minimize serialization overhead on large test cases for flexibility and efficiency. However, as a fallback, we keep a JSON serialization system that forwards arguments through input buffers, although this does not scale well to large test cases. We hope in future work to improve the robustness of the FFI implementation to prevent the need for the JSON fallback.

A.2.5 Kernel Caching and Incremental Compilation

To avoid redundant compilation, cLean implements a kernel cache:

```
structure CachedKernel where
  ptxPath : System.FilePath
  cudaHash : UInt64
  compileTime : Nat

def compileKernelToPTX (kernel : Kernel) : IO CachedKernel := do
  let cudaSource := kernelToCuda kernel
  let hash := hashString cudaSource
  let cachePath := s!"/tmp/clean_cache_{hash}.ptx"

  if ← System.FilePath.pathExists cachePath then
    return { ptxPath := cachePath, cudaHash := hash, compileTime := 0 }
  else
    -- Invoke nvcc to compile
    let proc ← IO.Process.spawn { cmd := "nvcc", args := ... }
    ...
```

This caching is particularly important during interactive development, where the same kernel may be executed many times in succession with different inputs.

A.3 Qualitative Examples

In this section, we provide a few end-to-end worked examples of cLean kernel implementation and verification.

A.3.1 SAXPY

We implement the SAXPY kernel as:

```
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)
```

Using this definition, we may now prove the safety of this kernel:

```
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 [HasRace, saxpySpec, deviceIRToKernelSpec, saxpyKernelIR,
extractFromStmt, extractReadsFromExpr, dexprToAddressPattern, List.lookup,
SeparatedByBarrier, AddressPattern.couldCollide, getArrayName,
AccessExtractor.getArrayLocation]
    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]
  . unfold BarrierUniform; intros; trivial
```

And with the safety verified, we may now also prove the functional correctness of this kernel.

```
def saxpyInitMem (x y : Array Rat) (N : Nat) : VMem :=
  fun arr idx =>
    if arr = "x" then
      if h : idx < x.size then VVal.rat x[idx] else VVal.rat 0
    else if arr = "y" then
      if h : idx < y.size then VVal.rat y[idx] else VVal.rat 0
    else if arr = "r" then
      VVal.rat 0
    else
      VVal.int 0

/-- Execute a single thread of SAXPY and show what it computes -/
theorem saxpy_thread_computes (N : Nat) (α : Rat) (x y : Array Rat)
    (blockSize numBlocks : Nat)
    (tid bid : Nat)
    (h_tid : tid < blockSize)
    (h_bid : bid < numBlocks)
    (h_gid : globalIdx1D blockSize bid tid < N)
    (h_x : x.size ≥ N)
    (h_y : y.size ≥ N)
    (mem₀ : VMem)
    (h_mem_x : ∀ i, i < x.size → mem₀ "x" i = VVal.rat x[i]!)
    (h_mem_y : ∀ i, i < y.size → mem₀ "y" i = VVal.rat y[i]!) :
    let gid := globalIdx1D blockSize bid tid
    let params : String → VVal := fun name =>
      if name = "N" then VVal.nat N
      else if name = "alpha" then VVal.rat α
      else VVal.int 0
    let mem_after := vExecThread1DWithParams saxpyKernelIR.body blockSize numBlocks
tid bid params mem₀
    mem_after.getR "r" gid = α * x[gid]! + y[gid]! := by
```

```
  intro gid params mem_after
  have h_cond : bid * blockSize + tid < N := by simp only [globalIdx1D] at h_gid;
exact h_gid
  have h_gid_lt_x : bid * blockSize + tid < x.size := by omega
  have h_gid_lt_y : bid * blockSize + tid < y.size := by omega

  simp only [mem_after, vExecThread1DWithParams, vExecThreadWithLocals,
saxpyKernelIR,
    vEvalStmt, vEvalExpr, vEvalBinOp, VCtx.setLocal, VMem.getR, gid, VCtx.mk',
VCtx.getLocal,
    globalIdx1D, VVal.toRat, VMem.get, VVal.toBool, VVal.toInt, VVal.toNat,
    VMem.set, params, h_cond, ↓reduceIte]

  simp_all [h_mem_x (bid * blockSize + tid) h_gid_lt_x,
            h_mem_y (bid * blockSize + tid) h_gid_lt_y,
            VMem.set
          ]


/-- Main functional correctness theorem for SAXPY

    Given:
    - Arrays x, y of size at least N
    - Initial memory with x, y, and zeroed output r
    - A grid/block configuration that covers all N elements

    Then: After executing the kernel, r[i] = α * x[i] + y[i] for all i < N
-/
theorem saxpy_correct (N : Nat) (α : Rat) (x y : Array Rat)
    (numBlocks blockSize : Nat)
    (h_x : x.size ≥ N)
    (h_y : y.size ≥ N)
    (h_cover : N ≤ numBlocks * blockSize)
    (h_blockSize_pos : blockSize > 0) :
    let mem₀ := saxpyInitMem x y N
    let params : String → VVal := fun name =>
      if name = "N" then VVal.nat N
      else if name = "alpha" then VVal.rat α
      else VVal.int 0
    let mem_f := vExecKernel1DWithParams saxpyKernelIR.body numBlocks blockSize
params mem₀
    ∀ i, i < N → mem_f.getR "r" i = α * x[i]! + y[i]! := by
  intro mem₀ params mem_f i hi

  -- Memory hypotheses
  have h_mem_x : ∀ j, j < x.size → mem₀ "x" j = VVal.rat x[j]! := by
    intro j hj
    simp only [mem₀, saxpyInitMem, ↓reduceIte, hj, dite_true]
    simp only [getElem!_pos, hj]

  have h_mem_y : ∀ j, j < y.size → mem₀ "y" j = VVal.rat y[j]! := by
    intro j hj
    simp only [mem₀, saxpyInitMem, String.reduceEq, ↓reduceIte, hj, dite_true]
    simp only [getElem!_pos, hj]
```

```
  -- Use the thread correctness lemma!

  -- helpers
  have h_tid : i % blockSize < blockSize := Nat.mod_lt i h_blockSize_pos
  have h_bid : i / blockSize < numBlocks := by
    have h1 : i < numBlocks * blockSize := Nat.lt_of_lt_of_le hi h_cover
    rw [Nat.mul_comm] at h1
    exact Nat.div_lt_of_lt_mul h1
  have h_gid : globalIdx1D blockSize (i / blockSize) (i % blockSize) = i := by
    simp only [globalIdx1D]
    ring_nf
    exact Nat.div_add_mod' i blockSize
  have h_gid_lt : globalIdx1D blockSize (i / blockSize) (i % blockSize) < N := by
    rw [h_gid]; exact hi

  have h_thread := saxpy_thread_computes N α x y blockSize numBlocks
    (i % blockSize) (i / blockSize)
    h_tid h_bid h_gid_lt h_x h_y mem₀ h_mem_x h_mem_y

  -- if thread (bid, tid) writes to any index idx, then idx = globalIdx bid tid
  have h_identity : CLean.Verification.IdentityAccessPatternWithParams
saxpyKernelIR.body blockSize numBlocks params "r" N := by
    intro bid tid mem idx h_bound h_writes
    simp only [globalIdx1D]
    by_contra h_neq
    apply h_writes
    simp only [ThreadWritesTo1DWithParams, vExecThread1DWithParams,
vExecThreadWithLocals,
               saxpyKernelIR, vEvalStmt, vEvalExpr, vEvalBinOp,
               VCtx.mk', VCtx.getLocal, VCtx.setLocal,
               VCtx.globalIdxX, VVal.toNat, VVal.toBool, VVal.toRat,
               VMem.get, VMem.set] at h_writes ⊢
    simp only [String.reduceEq, true_and, ↓reduceIte, and_true,
               ne_eq, h_neq, not_false_eq_true, and_false]
    split_ifs with h_lt
    · simp only [VMem.set, h_neq, and_false, ↓reduceIte]
    · rfl

  have h_decomp : (vExecKernel1DWithParams saxpyKernelIR.body numBlocks blockSize
params mem₀) "r" i =
      vExecThread1DWithParams saxpyKernelIR.body blockSize numBlocks
      (i % blockSize) (i / blockSize) params mem₀ "r" i := by
    exact vExecKernel1DWithParams_at_idx saxpyKernelIR.body numBlocks blockSize "r" N
params mem₀ i hi h_cover h_blockSize_pos h_identity

  simp_all [VMem.getR, h_gid, VMem.get, mem_f, h_decomp]
  convert h_thread using 2
```

And with the kernel fully verified, we may now execute on-device:

```
            def saxpyGPU (n : Nat)
                (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
```

A.3.2 GPU F4

We proceed to show the implementation of the cLean-powered GPU F4 variant. We first define the core kernel as follows:

```
kernelArgs GaussElimArgs(nrows: Nat, ncols: Nat, pivotRowIdx: Nat, p: Nat)
  global[mat pivotRow factors: Array Int]


device_kernel gaussElimKernel : KernelM GaussElimArgs Unit := do
  let args ← getArgs
  let nrows := args.nrows
  let ncols := args.ncols
  let pivotRowIdx := args.pivotRowIdx
  let p := args.p  -- prime modulus (65521) - used for modular arithmetic
  let mat : GlobalArray Int := ⟨args.mat⟩
  let pivotRow : GlobalArray Int := ⟨args.pivotRow⟩
  let factors : GlobalArray Int := ⟨args.factors⟩

  -- Each thread handles one row
  let row ← globalIdxX
  if row < nrows then do
    if row < pivotRowIdx || row > pivotRowIdx then do  -- row != pivotRowIdx
      let factor ← factors.get row
      if factor > 0 then do
        -- Process all columns for this row
        for col in [:ncols] do
          let idx := row * ncols + col
          let matVal ← mat.get idx
          let pivotVal ← pivotRow.get col
          -- Modular multiplication and subtraction with explicit mod reduction
          let prod := (factor * pivotVal) % p
          let diff := ((matVal - prod) % p + p) % p
          mat.set idx diff
```

And before applying this kernel, we first ensure that it is safe:

```
def gaussElimSpec (config grid: Dim3): KernelSpec :=
  deviceIRToKernelSpec gaussElimKernelIR config grid

theorem gaussElim_safe : ∀ (config grid : Dim3), KernelSafe (gaussElimSpec config
```

```
grid) := by
  intro config grid
  unfold KernelSafe
  constructor
  . unfold RaceFree
    intro tid1 tid2 h_distinct a1 a2 ha1 ha2
    simp_all [HasRace, gaussElimSpec, deviceIRToKernelSpec, gaussElimKernelIR,
extractFromStmt, extractReadsFromExpr, dexprToAddressPattern, List.lookup,
SeparatedByBarrier, AddressPattern.couldCollide, getArrayName,
AccessExtractor.getArrayLocation, AccessExtractor.getMemorySpace]
    intro h_race
    rcases h_distinct with ⟨_,_,h_neq⟩
    rcases ha1 with ha1 | ha1 | ha1 | ha1 <;>
    rcases ha2 with ha2 | ha2 | ha2 | ha2 <;>
    simp_all [AddressPattern.eval, SymValue.isNonZero]
  . unfold BarrierUniform; intros; trivial
```

So we may now implement the launcher:

```
def launchGaussElimKernel (mat : DenseMatrix) (pivotRowIdx : Nat) (pivotRow : Array
Int)
  (factors : Array Int) : IO DenseMatrix := do
-- Convert matrix to flat Int array
let matFlat : Array Int := mat.data.map fun x => Int.ofNat x.toNat
let scalarParams : Array ScalarValue := #[]
let arrays : List (Lean.Name × Array Int) := [
  (`mat, matFlat),
  (`pivotRow, pivotRow),
  (`factors, factors)
]

let response ← runKernelGPU gaussElimKernelIR GaussElimArgsResponse
  ⟨(mat.rows + 255) / 256, 1, 1⟩ ⟨256, 1, 1⟩
  scalarParams arrays


let arr := response.mat
let p : Int := PRIME.toNat
let newData : Array FpElem := arr.map fun i =>
  let normalized := if i < 0 then (i % p + p) % p else i % p
  normalized.toNat.toUInt32

pure { mat with data := newData }
```

With this, we may implement the rest of the F4 algorithm, start

```
  def gaussianEliminationGPU (m : DenseMatrix) : IO DenseMatrix := do
  let useGPU ← hasGPU

  let mut mat := m
  let mut pivotRowIdx := 0
  let _ncols := mat.cols
  let _nrows := mat.rows

  for col in [:mat.cols] do
    if pivotRowIdx >= mat.rows then break
```

```
    -- Find pivot (CPU)
    let mut found := false
    let mut pivotIdx := pivotRowIdx
    for row in [pivotRowIdx:mat.rows] do
      if mat.get row col != 0 then
        pivotIdx := row
        found := true
        break

    if found then
      -- Swap rows if needed (CPU)
      if pivotIdx != pivotRowIdx then
        for c in [:mat.cols] do
          let tmp := mat.get pivotRowIdx c
          mat := mat.set pivotRowIdx c (mat.get pivotIdx c)
          mat := mat.set pivotIdx c tmp

      -- Scale pivot row to make leading coefficient 1 (CPU)
      let pivotVal := mat.get pivotRowIdx col
      let pivotInv := inv pivotVal
      for c in [:mat.cols] do
        mat := mat.set pivotRowIdx c (mul (mat.get pivotRowIdx c) pivotInv)

      if useGPU then
        -- GPU path: launch kernel for row elimination
        let pivotRow : Array Int := Array.range mat.cols |>.map fun c =>
          Int.ofNat (mat.get pivotRowIdx c).toNat
        let factors : Array Int := Array.range mat.rows |>.map fun r =>
          Int.ofNat (mat.get r col).toNat
        mat ← launchGaussElimKernel mat pivotRowIdx pivotRow factors
      else
        -- CPU path: eliminate column in other rows
        for row in [:mat.rows] do
          if row != pivotRowIdx then
            let factor := mat.get row col
            if factor != 0 then
              for c in [:mat.cols] do
                let newVal := sub (mat.get row c) (mul factor (mat.get pivotRowIdx
c))
                mat := mat.set row c newVal

      pivotRowIdx := pivotRowIdx + 1

  pure mat


def f4ReductionGPU (polys : Array Poly) : IO (Array Poly) := do
  if polys.isEmpty then return #[]

  let (mat, monos) := buildDenseMatrix polys
  let reduced ← gaussianEliminationGPU mat
  let numVars := polys[0]!.numVars
  pure (extractPolys reduced monos numVars)
```