Fearless Concurrency on the GPU

Melih Elibol
melibol@nvidia.com
NVIDIA
USA

Jared Roesch
jroesch@nvidia.com
NVIDIA
USA

Isaac Gelado
igelado@nvidia.com
NVIDIA
USA

Eric Buehler
eric@huggingface.co
Hugging Face
USA

Michael Garland
mgarland@nvidia.com
NVIDIA
USA

Abstract

Rust has made safe systems programming practical on the CPU, but writing custom GPU kernels in Rust still forces programmers outside the language's ownership guarantees. We present cuTile Rust, a tile-based system for safe, idiomatic GPU kernel authoring in Rust that compiles kernels to Tile IR. cuTile Rust extends Rust's ownership discipline to tile-based GPU kernels: mutable outputs are split into disjoint pieces, kernel launches preserve the host-side ownership contract, and the Rust compiler enforces the same ownership rules inside the kernel. We prove the safe surface data-race-free under Tile IR's memory model. However, bounds safety still requires runtime checks. The compiler therefore eliminates checks it can prove redundant and, where possible, moves others out of the kernel into host-side launch preconditions. On the host, the same ownership contract carries through a composable execution model that runs the same operations synchronously, under async/await, or as CUDA graph replay, with async at parity with synchronous execution.

Our evaluation shows that these abstractions preserve performance on high-end GPUs. On the NVIDIA B200 GPU, cuTile Rust achieves 7 TB/s for element-wise operations and 2.1 PFlop/s for GEMM (98% of cuBLAS), on par with cuTile Python. Grout, a Qwen3 inference engine built on cuTile Rust, reaches 171 generated tokens/s for Qwen3-4B on the NVIDIA GeForce RTX 5090 and 82 for Qwen3-32B on the B200 in batch-1 decode, at parity with vLLM and SGLang and consistent with a memory bandwidth roofline sanity check.

1 Introduction

The Rust language makes programming with static safety guarantees both practical and efficient. These qualities are particularly attractive in applications that must coordinate the activity of many threads working on shared data objects, and modern AI frameworks and LLM inference engines are important examples [10, 11, 19]. Such applications perform computations where parallelism is abundant, and significant portions of their workloads are built around tensor operations. These tensor workloads are well suited to GPUs. However, GPUs do not fit comfortably into the current Rust ecosystem.

GPU-accelerated applications contain code for both the CPU (host) and GPU (device) processors. The host code allocates data, builds kernel arguments, and submits work for execution on the device. Device kernels execute in a single-program multiple-data (SPMD) fashion: thousands of threads start at the same entry point, scheduled onto the GPU's processors in thread blocks whose threads share on-chip memory and synchronize explicitly. Kernel arguments cross this boundary as raw pointers and scalars, so whatever a safe host language such as Rust established about ownership stops at the launch site, and correct synchronization and the prevention of data races inside the kernel are left in the hands of the programmer.

The Rust compiler (`rustc`) is able to generate device code for NVIDIA GPUs via the LLVM PTX backend, and other projects have experimented with other code generation paths [13, 15, 32]. However, in each of these cases, kernels still require `unsafe` for basic performance techniques, such as the use of shared memory. As GPU kernels are increasingly generated by AI tools rather than written by hand, this gap grows more consequential: The burden shifts from writing kernels to trusting them.

With cuTile Rust [33], we extend Rust's safety guarantees across both host and device code. Many high-performance CUDA kernels partition their outputs among thread blocks and share read-only inputs. We therefore adopt a tile-based model for kernels [30, 31], which exposes this pattern at a granularity where Rust's ownership rules apply directly. Instead of launching a grid of thread blocks, a tile kernel is launched as a grid of single-threaded tile programs, each operating on immutable, fixed-size tiles loaded from and stored to tensors in memory. The compilation stack realizes each tile program as a thread block and manages the shared memory and synchronization within it. Kernels compose loads, stores, and whole-tile operations whose memory footprints are visible in their types. Rust's aliasing discipline therefore maps directly onto the units the hardware schedules.

Existing tile DSLs, however, leave the destination of each store to the program. In the head-permutation example of §3.3, the store's destination index omits the head coordinate.

The kernel compiles and runs, but tile programs processing different heads write to the same destination, causing a data race. We define tensor types that allow immutable tensors to be passed from host to device as is, while mutable tensors are partitioned before launch and mapped to mutable sub-tensors within the kernel grid. This ensures that no two tile programs can write to the same sub-tensor.

Bounds safety also requires checks. Like Rust, cuTile Rust checks every access, but a GPU kernel is on the hot path, so our compiler resolves each check obligation where it is cheapest: At specialization time, at the launch site on the host, at a loop preheader in the kernel, or, when necessary, at the access site. In the GEMM of §4, every check resolves before the kernel runs, and in attention the remaining checks move to a loop preheader, so neither kernel has a check in its hot loop.

The tensor operations performed in host code compose into lazy GPU work that can be executed synchronously, submitted asynchronously to a stream, or captured as a CUDA Graph for later replay. The result is a GPU programming model that is safe end to end, from host allocation through kernel execution and back, with performance on par with state-of-the-art GPU libraries and inference engines (§6).

Our main programming-language contribution is an ownership-checked tile programming model with bounds-check placement. Our systems contribution is a host execution model that combines bounded spin polling with a completion reactor, making asynchronous GPU execution as fast as synchronous execution at microsecond timescales. We make the following contributions:

1. A safe, high-performance programming model for Rust: host tensors become device-side tensor and partition views, tile kernels execute with single-threaded semantics, and a data-race-freedom theorem holds for safe kernels and their launches under Tile IR's weakly ordered memory model (Theorem A.1, Appendix A). The partition map keeps each program's mutable sub-tensors disjoint, and branded partition indices let a program iterate over them with every store proven in bounds at no runtime cost. Unsafe low-level memory operations remain available for building new safe abstractions.
2. A bounds-check placement mechanism that migrates checks across the CPU/GPU boundary. Unoptimized checks add 17–22% to runtime on the evaluated GEMM and attention kernels; optimized placement brings performance to parity with their unchecked versions (§6.2).
3. A composable host execution model whose generated launch code prevents host access while GPU work is in flight and returns ownership in the same form after completion, with async at parity with synchronous execution at microsecond scale.

```
1  use cutile::prelude::*;
2  #[cutile::module]
3  mod kernel {
4    use cutile::core::*;
5    #[cutile::entry()]
6    fn add<const B: i32>(
7      z: &mut Tensor<f32, {[B]}>,   // exclusive write
8      x: &Tensor<f32, {[-1]}>,      // shared read
9      y: &Tensor<f32, {[-1]}>,      // shared read
10   ) {
11     let tx = x.load_like(z);
12     let ty = y.load_like(z);
13     z.store(tx + ty);
14   }
15 }
16 fn main() -> Result<()> {
17   let x = api::ones::<f32>([1024]);
18   let y = api::ones::<f32>([1024]);
19   let z = api::zeros::<f32>([1024]).partition([128]);
20   let (_z, _x, _y) = kernel::add(z, x, y).sync()?;
21   Ok(())
22 }
```

Listing 1. Element-wise addition in cuTile Rust. The host (lines 16–22) partitions the output and passes inputs as shared reads; the kernel (lines 2–15) operates on tiles with single-threaded semantics. The program contains no `unsafe` keywords.

Our benchmarks on an NVIDIA DGX B200 reach roughly 7 TB/s for element-wise operations and 2.1 PFlop/s for GEMM, about 98% of cuBLAS and on par with cuTile Python. End-to-end, Grout, a Qwen3 LLM inference engine built upon cuTile Rust, runs at parity with vLLM and SGLang, with decode throughput consistent with the memory bandwidth roofline. Beyond our own case study, cuTile Rust is in use by two Rust inference stacks: Candle [19], Hugging Face's machine-learning framework, exposes it through an opt-in feature for writing JIT-compiled CUDA kernels as custom operations, shipping a fused mixture-of-experts grouped matrix multiplication written in it; and mistral.rs [10] depends on it for optional kernel acceleration.

2 Overview

This section walks through a complete cuTile Rust program, Listing 1: host code constructs tensors and partitions the mutable output, and device code runs a tile program for each output partition. Sections 3–5 present the design. §3 gives the safety model, which determines what the compiler proves statically and which bounds checks remain at run time. §4 describes where the compiler resolves those checks, at specialization time, on the host as launch preconditions, or in the kernel. §5 describes the host execution model that performs the launch and returns ownership when the work completes.

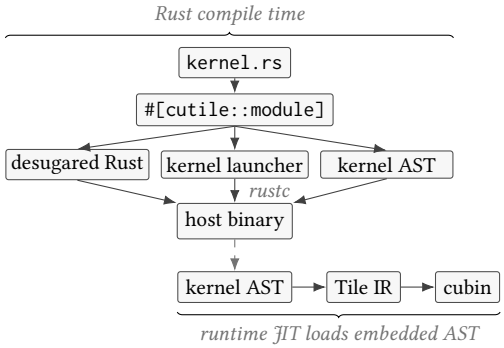


Figure 1. A kernel from source to GPU: what the module macro emits at Rust compile time (top) and what the runtime JIT compiles at first launch (bottom, dashed arrow).

On the host (lines 16–22), the program creates two input tensors, partitions the mutable output into 128-element chunks, and calls `kernel::add(z, x, y)`, which returns a value that represents the device operation and holds its arguments. Nothing runs until an execution mode is chosen: `.sync()` launches the operation, waits for the stream, and returns the same tensors. The same value can instead be awaited as a future or captured into a CUDA graph (§5).

The kernel signature (lines 6–10) carries Rust's ownership rules into the device code. The output parameter is an exclusive mutable sub-tensor (`&mut Tensor`), while the inputs are shared read-only tensors (`&Tensor`). The body (lines 11–13) loads a tile from each input, adds the tiles, and stores the result through the exclusive output reference. Here, `x.load_like(z)` partitions `x` like `z` and loads the tile at the partition index this program owns in `z`.

Two compilers process this program (Figure 1). `rustc` type-checks the host code and the macro-generated Rust version of the kernel. The module macro also embeds the kernel's original AST in the host binary. At first launch, cuTile Rust's just-in-time (JIT) compiler lowers that AST to Tile IR and enforces the same access rules on the device, inserting runtime checks where bounds cannot be proved statically (§4). The implementation is about 90 thousand lines of Rust, tests included: the proc macro (7.5k), the JIT compiler and Tile IR layer (50k), the tensor runtime (13k), and the CUDA driver and async layers (20k).

Rust's relevant rule is aliasing XOR mutability: A value may have either one mutable reference (`&mut T`) or any number of immutable references (`&T`), but not both simultaneously. Rust provides safe abstractions where the compiler can verify that invariant, runtime checks when it is not known statically, and explicit `unsafe` code where the programmer supplies it. cuTile Rust follows the same structure (§3) and places the runtime checks where they are cheapest (§4).

2.1 GPU Execution Model

GPU programming splits host-side launch from device-side execution: Rust's typed values collapse to raw pointers and scalars at the launch boundary, and device kernels run many SIMT threads whose address disjointness and visibility are programmer-managed invariants. cuTile Rust keeps CUDA's host launch and device execution structure but raises the device program to the tile level: The programmer writes sequential code over multi-dimensional tiles, and the compilation stack maps tile operations to thread blocks, manages shared memory, and performs the parallel decomposition. Element-wise kernels, matrix multiplication, reductions, normalization, and attention are naturally tile-shaped. Schedules outside the safe surface are built from the unsafe operations of §3. The tile model's safety payoff is that the operations a kernel performs are loads, stores, matrix multiplies, and reductions over whole tiles, whose memory footprints the compiler can see. There is no per-thread shared-memory protocol for it to reason about.

A thread block is a unit of work scheduled onto a GPU streaming multiprocessor. CUDA does not guarantee the order in which blocks run or that they are resident at the same time. Its ordinary programming model therefore requires blocks to execute independently, in any order, in parallel or in series [29]. Threads within a block can cooperate through shared memory and barriers; coordination across blocks requires additional mechanisms or separate kernel launches. To avoid that coordination, programmers structure many high-performance tensor kernels so that each block has exclusive access to the regions it mutates and shared read access to the remaining inputs.

The tile model exposes each block's computation as a single-threaded program. At that level, the memory partitioning encouraged by CUDA's execution model is exactly the aliasing-XOR-mutability discipline that Rust's ownership rules enforce. The partitioned form that safe cuTile Rust requires is therefore the form these high-performance kernels already take. With tensors partitioned on the host, cuTile Rust can check ownership of mutable output regions statically.

The order of a program's own loads and stores is the other half of the safety argument, and Tile IR [30] leaves it to the program, exposing memory ordering through *tokens*: A token produced by one memory operation and consumed by another orders the two, and operations without token dependencies may be reordered. cuTile Rust threads tokens through mutable tensor operations, so exclusive accesses carry ordering through the mutable view while shared reads remain free for the compiler to reorder (§3.3).

3 Safety Model

Our safety model extends Rust's across the launch boundary and into the kernel. Host code is ordinary safe Rust and

Table 1. Types at the kernel launch boundary: the host type, the kernel parameter it becomes, and the semantics the borrow checker enforces. The grammar below is the legend. `P`, `MP`, and `MT` abbreviate `Partition`, `MappedLaunchPartition<Partition<..>>`, and `MappedPartitionMut`; M is the partition map of §3.2.

Host type	Kernel parameter	Semantics
`P<Tensor<T>>`	`&mut Tensor<T, S>`	move; exclusive
`P<&mut Tensor>`	`&mut Tensor<T, S>`	borrow; exclusive
`MP<Tensor<T>>`	`MT<T, S, M>`	move; mapped excl.
`MP<&mut Tensor>`	`MT<T, S, M>`	borrow; mapped excl.
`Tensor<T>`	`&Tensor<T, S>`	move; shared ref.
`Arc<Tensor<T>>`	`&Tensor<T, S>`	shared; immutable
`&Tensor<T>`	`&Tensor<T, S>`	borrow; non-`'static`
`&TensorView<T>`	`&Tensor<T, S>`	borrow; 0-copy view
`T`	`T`	by value; scalar
`*const T`, `*mut T`	unchanged	`unsafe` only; raw pointer

Interface grammar

T ::= `bool` | `int` | `float` $\quad D ::= n \in \mathbb{Z}^+ \mid$ `-1`
$S, M ::= [D_0, D_1, \ldots, D_{N-1}] \mid [D; N], N \in \mathbb{Z}_{\geq 0}$
Param ::= `&mut Tensor<T, S>` | `&Tensor<T, S>` | `MT<T, S, M>`
| `T` | `*const T` | `*mut T`

keeps Rust's own guarantees. Inside the kernel, partitioning gives each tile program a disjoint mutable view and token ordering gives it happens-before over its own mutable operations. At the launch boundary, the generated interfaces preserve ownership as host types become kernel parameters. §3.3 states the data-race-freedom theorem this structure supports. Leaving the safe surface is an explicit choice at one of two tiers: An `unsafe fn` entry point may take raw pointers or disable checks with `unchecked_accesses`, making the whole kernel unsafe code, or a safe kernel may perform individual raw-pointer loads and stores in unsafe blocks, keeping every other access checked. In both, `unsafe` has the same semantics as in Rust: The programmer upholds the invariants the compiler no longer checks, and the low-level operations it exposes are the base for new safe abstractions.

3.1 Tile Programming Model

A cuTile Rust program is an ordinary Rust program whose GPU kernels live in a *tile module* (`#[cutile::module]`). Functions marked `#[cutile::entry()]` are *entry points*, the only functions invocable from host code, and cuTile Rust generates a typed launch interface for each. A kernel body loads tiles from its tensor parameters, computes on them (tiles are immutable values), and stores tiles through its exclusive output parameter. Kernel parameters take one of the six forms of Table 1. Tensor writes go through a partition because partitioning is what gives each program a disjoint sub-tensor (§3.2). Read-only accesses do not conflict with one another, so tensors that the kernel does not mutate arrive as shared references. Mutable tensors are partitioned on the host and arrive in the kernel as exclusive `&mut Tensor<T, S>` views over one sub-tensor per program, or as the mapped partitions of §3.2. Immutable tensors, whether owned, borrowed, or behind an `Arc`, arrive as shared `&Tensor<T, S>` views of the whole tensor. The proc macro checks these mappings at compile time, and the launcher returns every host value in the type it was passed: An `Arc<Tensor<T>>` comes back as an `Arc<Tensor<T>>`, a borrow as the borrow, so a launch reads like an ordinary call that borrows or moves its arguments (Figure 5).

Host tensors are dynamically shaped, since their extents come from data and configuration at run time. Kernel parameters carry a shape in their type, and the launcher checks the one against the other. The shape `S` is a *const generic array*: an array of dimensions carried as a type parameter, each a static extent or a dynamic `-1`, and any array expression over const generics (e.g. `{[BM, BN]}` in Listing 2) can be written wherever a type parameter is expected. Stable Rust does not support array-valued const parameters, so the module macro lowers each shape parameter to a rank-specialized *shadow type* with one scalar const generic per dimension. Rank-polymorphic operations become traits with one implementation per rank, which `rustc` resolves as usual. Shapes are therefore checked with ordinary const generics: `mma` takes `Tile<T, {[M, K]}>` and `Tile<T, {[K, N]}>` and returns `Tile<T, {[M, N]}>`, so a mismatched inner dimension is a type error. These shapes are the concrete variadic types Tile IR expects, and the JIT, which works from the original source, constructs them directly.

3.2 Partitioning

A tile program's mutable tensors are partitioned before launch. In Listing 1, the host splits the 1024-element output into 128-element sub-tensors with `.partition([128])`. On the device, the generated entry function hands each tile program an exclusive `&mut Tensor` over one sub-tensor, while the immutable inputs `x` and `y` become shared `&Tensor` inputs visible to all programs. Both are Tile IR views. A view is a structured window into GPU memory with a shape and an element type. The exclusive one is a *partition view* and the shared one a *tensor view*. Rust uses the same idiom for CPU parallelism: Rayon gives each closure one disjoint chunk of a mutable slice, and `partition` gives each tile program one disjoint sub-tensor.

The launcher derives the launch grid from the partition shape. CUDA grids are three-dimensional, so mutable tensors are capped at rank 3, while immutable tensors are not. Safe code constructs mutable partitions only through the partition APIs, from an owned tensor (`Partition<Tensor<T>>`, moved into the launch) or an exclusive borrow (`Partition<&mut Tensor<T>>`). Either way, host access is unavailable until execution completes.

A *mapped partition* specifies a *partition map*, a function chosen by the programmer from output sub-tensors to the tile programs that own them. The plain `&mut Tensor<T, S>` is the special case of one sub-tensor per program. `MappedPartitionMut<T, S, M>` lets the map `M` assign several

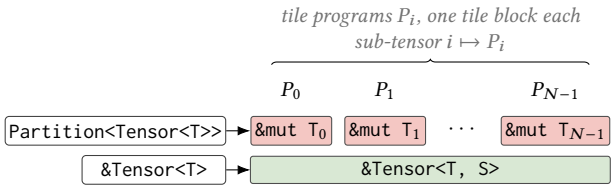


Figure 2. Host-to-device partitioning: a `Partition<Tensor<T>>` gives each tile program its own exclusive sub-tensor (rose). A `&Tensor<T>` is shared by every program as a read-only view (green).

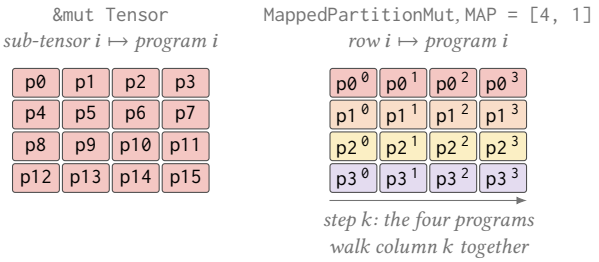


Figure 3. The partition map on a 4×4 grid of output sub-tensors. Left: the plain partition, one program per sub-tensor. Right: the map `[4, 1]` with four persistent programs, row i to program i, visited left to right (step k). Each sub-tensor still has exactly one owner.

sub-tensors to one program, which then owns a bounded *sequence* of them (Figure 3). The mapped form lowers to the same Tile IR partition view and adds only the obligation that a program write only sub-tensors assigned to it by the map. The launcher checks the map's program count against the number of sub-tensors. On the host, the mapped form is the plain partition plus one call:

```
// one sub-tensor per program (Listing 1)
let z = api::zeros::<f32>([M, N]).partition([BM, BN]);
// row i -> program i, four persistent programs
let z = api::zeros::<f32>([M, N]).partition([BM, BN])
  .map([4, 1], 4);
```

The second arrives in the kernel as `MappedPartitionMut<f32, {[BM, BN]}, {[4, 1]}>`.

A mapped partition lets GEMM express an efficient schedule. With one program per output sub-tensor, each program reloads its share of the K-dimension operands. Fewer, persistent programs, each walking a sequence of sub-tensors, reuse operands across them. In the map of Figure 3, the four programs walk one column together and share its `y` tile, the schedule of the GEMM of §4 (Listing 2). There, `MAP` instantiates the partition map and `z.iter_indices()` yields compiler-branded `PartitionIndex` values for the sub-tensors mapped to the executing program [23, 26, 41]. A store checks that its index came from `z`'s map before lowering to Tile IR, so the write stays safe although one program writes many sub-tensors.

The general object behind this API is the *partition mapping function*: Any function from output sub-tensors to tile programs yields disjoint ownership, because each sub-tensor has exactly one image. An implementation must supply each program's preimage as the sequence of sub-tensors it iterates. `MappedPartitionMut` restricts these functions to the family described by `M`, the grouped schedules that GEMM-like kernels use for operand reuse, and its iterator enumerates each program's preimage directly, so the bounds of every store are proven at compile time. That family covers 22 of the 25 kernels of the Qwen3 inference engine of §6.4. The remaining three express schedules outside it with six one-line unsafe blocks inside otherwise safe kernels. A user-defined mapping function from a tensor partition to tile programs could be evaluated on the host before launch to produce the same preimages. Generalizing the iterator machinery to it is future work (§9).

3.3 Data-Race Freedom

The store below is from a kernel that permutes an attention tensor from shape (B, H, M, D) to (B, M, HD), merging the head and feature axes into the destination's third dimension. The store is shown at both tiers:

```
// indexed store: the programmer carries the obligation
unsafe { dst.store_unchecked(tile, [b, m, 0]) }; // BUG
// safe store: the target is the partition view itself
dst.store(tile);
```

The first form addresses the destination by index. It is an unsafe store, and the programmer carries the obligation that no two tile programs write to the same location. The correct tile index is `[b, m, h]`. This instance omits the head coordinate, so the H programs that share (b, m) each load a different source tile and write it to the same destination. Nothing orders these writes (§2.1), so under the Tile IR memory model, the program has a data race and undefined behavior. Over ten runs, 7.7–8.7% of output elements differ between consecutive runs (61–70% in the contended tiles) and 98–99% differ from the intended result. The indexed store is the store interface of mainstream tile DSLs [31, 38], and nothing in those DSLs rules this bug class out, because the program chooses the destination.

cuTile Rust makes this store impossible to construct: `dst` is a partition view (`&mut Tensor<f32, {[1, BM, BD]}>`) over the destination, each tile program owns one sub-tensor, and it stores to the view itself, with no index to omit or mix up.

Disjoint partitions rule out conflicts between programs. Within a program, mutable operations must also keep their order, and Tile IR's token-ordered operations have no default program order (§2.1), so the compiler threads a token chain through the mutable tensor operations it emits. For a kernel that loads from and then stores to a mutable tensor `c`:

$$t_0 \xrightarrow{\texttt{c.load()}} t_1 \xrightarrow{\texttt{c.store()}} t_2$$

The store consumes the token the load produced, so it observes the load. Loads through immutable references carry no token, and the Tile compiler reorders them freely. Token threading plays the role that sequenced-before plays for ordinary Rust.

Every execution of a kernel written in safe cuTile Rust is data-race-free under Tile IR's memory model, and the property extends across launches to safe host programs. Two tensor accesses conflict when they access the same memory location and at least one is a write. In safe code, that write goes through the issuing program's partition view. An access from another program cannot touch that view, since partitions are disjoint. An access from the same program through an immutable view cannot touch the partition either, since immutable views cannot alias partitions. The remaining case, two accesses through the program's own `&mut` view, is ordered by the token chain, so the pair is morally strong under Tile IR's definition. Across launches, the host execution model preserves operand lifetimes and access rights until completion. Conflicting accesses across launches follow stream order, and dependencies are recorded during graph capture to preserve that order on replay. This establishes happens-before where the API permits reuse (Theorem A.1, Appendix A).

4 Bounds-Check Placement

Every load from and store to a partition in a kernel carries a proof obligation per axis: The coordinate must be non-negative and must select a tile inside the tensor, $0 \le i$ and $i \cdot t < e$ for tile size t and extent e. These are tile-index bounds: an edge tile may extend past the tensor. Tile IR masks stores for those elements; loads use the view's padding value, or unspecified values if none is set [30]. Read-only `Tensor::partition` uses zero padding. A check that discharges this obligation at the access site may sit in the hot loop of the kernel, on the path between the loads and the tensor-core operation, where it consumes registers and instruction slots. On the two kernels of §6.2, checks left at their access sites slow the kernel by 17–22%. The compiler lowers that cost by *bounds-check placement*, code motion for in-kernel checks whose targets include the host. Check placement resolves each obligation once during *specialization*, the step of JIT compilation that fixes the kernel to the generic parameters of its first launch (e.g. tile shapes) while dynamic extents remain launch-time values (Figure 1). Those generic parameters are the compiler's *assumptions*. When the assumptions cannot discharge an obligation, placement emits a check at the least costly point where all required values are available.

Resolution has four outcomes (Figure 4). An obligation is *discharged at specialization* when it follows from the assumptions, and no runtime check is emitted anywhere. Checks are discharged in three cases: the index is the iterand of the axis it indexes, as the K loop of Listing 2 iterates `0..num_tiles(&part_x, 1)`; a literal coordinate is within the bounds of a static extent; or a mapped-partition store uses a branded index, which is in bounds by construction (§3.2).

An obligation becomes a *launch precondition* when every operand is known at launch (e.g. tensor extents and tile counts). The check moves into the generated launcher, runs once per launch before any work is queued, and returns an error on failure. The cross-tensor tie of Listing 2, $\lceil dim(x, 1)/BK \rceil \le \lceil dim(y, 0)/BK \rceil$, is one such check.

An obligation is hoisted to a *loop preheader* when it cannot become a launch precondition, because one of its operands exists only inside the kernel, but its check can still leave the loop. Such an operand is a value the kernel computes (e.g. a coordinate decoded, inside a persistent loop, from the partition index of the current sub-tensor). An index that is affine in the loop's iterand turns the per-iteration check into one inequality over the loop's range. A loop-invariant index is tested directly. The check moves outward to the preheader of the outermost loop at which all of its operands exist, guarded so that a loop with no iterations cannot trap.

An obligation stays *in place*, checked at the access site against the actual index, when the access may not execute on every iteration, under a runtime conditional or in a loop body with an early exit, or when our analysis cannot bound a non-affine index. The entry attribute `deny_in_kernel_checks` turns the last two outcomes into errors during specialization, so a kernel that compiles with it contains no bounds-check assert.

Listing 2 carries four obligations, two loads over two axes each, and none reach the kernel. The K-loop index into `x` is discharged at specialization, since the loop's range is that axis's tile count. The other three become launch preconditions, each rejected on the host with its own message when violated. The kernel compiles under `deny_in_kernel_checks`, and its Tile IR contains no asserts.

The causal prefill attention kernel of §6.2 exercises the other outcomes (Figure 4). Of its nine obligations, three are discharged at specialization, two become launch preconditions, and four are hoisted to the preheader of its inner loop, where they run once per work step of the persistent loop instead of on each of the loop's 256 iterations. Our current analysis leaves those four at the preheader, because their operands are decoded inside the persistent loop from the partition index that `iter_indices` yields for the current sub-tensor. Appendix B lists every obligation against the kernel's source.

Obligations whose operands are decoded from that partition index, such as attention's four, stop at the preheader today. Because `iter_indices` covers a map known at launch and the operand chains are monotone in its coordinates, a universally quantified launch check over the map space could decide them before launch. That would be a launch

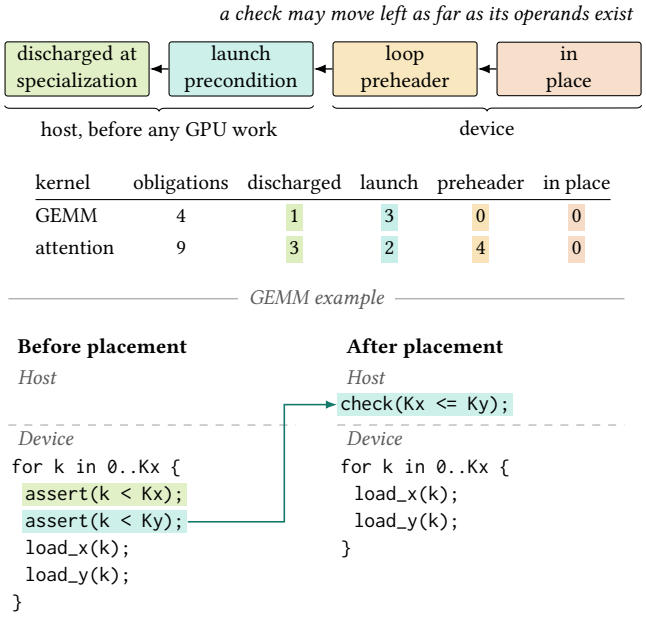


Figure 4. Bounds-check placement: resolution sites, the kernel census (Appendix B), and a worked GEMM example. K_x and K_y are the operands' reduction-axis tile counts. The loop bound proves $0 \le k < K_x$, eliminating the X check; the Y check becomes one host launch precondition. Neither check remains in the GPU loop. The pseudocode omits other axes and computation.

precondition of a new form, and we leave it to future work. We return to the other limits of placement in §7.

5 Execution Model

On the host side, efficient GPU work scheduling is asynchronous. The type system therefore has to keep host access and in-flight device writes apart without forcing the host to block.

Host-side GPU work is built around `DeviceOp`, a trait for lazy, typed GPU work that composes with other operations before anything is submitted to CUDA. Like Rust's iterators, it builds work statically. The generated launcher accepts either a host value or a device operation for each parameter and returns a device operation, so a kernel call whose operands are device operations is itself a device operation. Combinators such as `then`, which chains a dependent operation, compose in the same way. The result is a nested operation type that forms a directed acyclic graph of operations. Nothing runs until a method such as `.sync()` executes it. A generated launcher's `execute` impl runs the operations that produce the kernel's arguments, launches the kernel on the stream of its execution context, and returns its arguments. Composition is efficient because each operation in the graph is submitted to the stream while its predecessors are still in flight. The returned tensors may therefore name memory

```rust
#[cutile::entry(deny_in_kernel_checks = true)]
fn gemm_persistent<T: ElementType, const BM: i32,
    const BN: i32, const BK: i32, const MAP: [i32; 2]>(
  mut z: MappedPartitionMut<T, {[BM, BN]}, MAP>,
  x: &Tensor<T, {[-1, -1]}>,
  y: &Tensor<T, {[-1, -1]}>,
) {
  let part_x = x.partition(shape![BM, BK]);
  let part_y = y.partition(shape![BK, BN]);

  for out_idx in z.iter_indices() {
    let (bid_m, bid_n) = out_idx.components();
    let mut tz = constant(T::ZERO, shape![BM, BN]);
    for k_tile in 0i32..num_tiles(&part_x, 1) {
      let tx = part_x.load([bid_m, k_tile]);
      let ty = part_y.load([k_tile, bid_n]);
      tz = mma(tx, ty, tz);
    }
    z.store(tz, out_idx);
  }
}
```

Listing 2. Persistent GEMM over a mapped output partition. `z.iter_indices()` yields the branded `PartitionIndex` values assigned to this program. The kernel carries `deny_in_kernel_checks`.

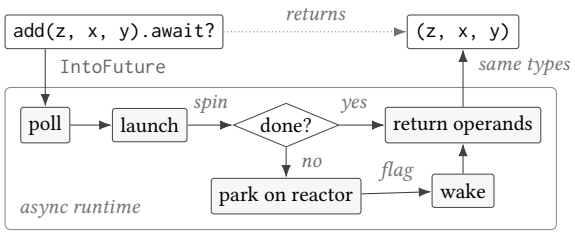
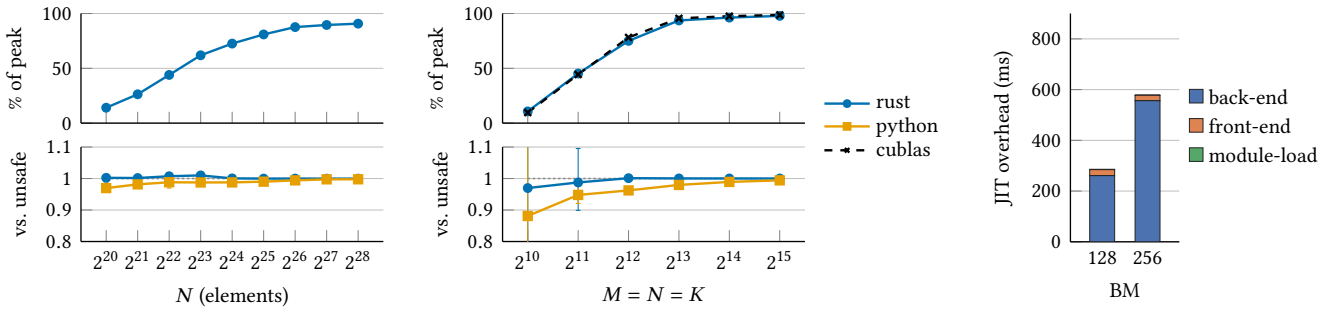


Figure 5. An awaited device operation returns the operands it was given. A poll spins on the stream within the budget and otherwise parks the future on the reactor.

whose writes have not completed, so `execute` is unsafe. The `sync` method executes the graph of operations safely by synchronizing the stream before it returns the arguments. In pseudocode:

```rust
trait DeviceOp {
  type Out: Send;
  unsafe fn execute(self, cx: &Ctx) -> Result<Self::Out>;
  fn sync(self) -> Result<Self::Out> {  // safe
    let cx = Ctx::new(stream()?);
    let out = unsafe { self.execute(&cx) };
    cx.stream().synchronize()?; out
  }
}
```

A `DeviceOp` executes in one of three modes: `.sync()` blocks until the work completes; `.await` (via `IntoFuture`) suspends the current task until it completes; and `.graph()` captures



(a) Element-wise add (memory-bound). **(b)** GEMM (compute-bound). **(c)** JIT overhead.

Figure 6. Checked versus unchecked kernels, and JIT overhead. Upper strips: safe Rust as a fraction of peak, cuBLAS against the same peak. Lower strips: safe Rust and cuTile Python relative to unsafe Rust. Whiskers are the numerator's spread (p_{25}–p_{75} element-wise, ±1 sd GEMM), the dotted line parity.

the composition as a CUDA graph for low-latency replay of work launched repeatedly. A replay of the captured graph is itself a device operation, so it executes synchronously or asynchronously like any other.

5.1 Asynchronous Execution

Every device operation implements `IntoFuture`. A future holds an operation's operands until an `.await` returns them (Figure 5). A kernel that outlives the block that launched it, as a spawned task does, must own its arguments or hold `'static` references, as any async runtime's spawn requires. The asynchronous form costs the programmer one method on the same device operation:

```
let (z, x, y) = add(z, x, y).sync()?;    // blocks
let (z, x, y) = add(z, x, y).await?;     // suspends
let t = spawn(add(z, x, y).into_future()); // awaitable
```

Short GPU work leaves asynchronous execution little room for host overhead. Decode-step kernels run for one to a few tens of microseconds (Appendix C) while CUDA's host callback for completion notification costs tens of microseconds per operation. To reduce completion overhead, we combine two techniques used in microsecond-scale systems: polling and event-driven completion [5, 22]. The future first polls the stream for a bounded time, then falls back to a completion reactor. The default budget of 20 μs is set an order of magnitude above the reactor's wake cost and above the duration of most decode-step kernels, so most waits resolve at the poll site. The reactor is one thread that wakes a registered future once its stream writes a completion flag into host memory. Short waits therefore avoid a reactor wakeup, while longer waits release the host thread after the spin budget. The roughly 2 μs wake cost is then amortized over the longer GPU execution (Appendix C). Async matches sync across the evaluated pipeline lengths (§6.3, Figure 8a).

5.2 CUDA Graphs

`CudaGraph::scope` provides a closure-based API over a scope handle `s`. Each `s.record(op)` adds the operation `op` to the graph as a node. Recording a node does not execute its GPU work. Releasing a borrow after `record()` therefore cannot race with the recorded operation. The graph preserves capture order on replay, so buffers may be reused between nodes. A `GraphNode` marker trait restricts capture to non-allocating operations, so replayed nodes do not depend on addresses that change across replays.

6 Evaluation

We measure the runtime cost of safety (§6.1), the cost that bounds-check placement recovers (§6.2), the host overhead of each execution mode (§6.3), and the end-to-end performance of an inference engine built on cuTile Rust (§6.4). Experiments run on one GPU of an NVIDIA DGX B200 or on a workstation with an NVIDIA GeForce RTX 5090. Each experiment states which. All measurements use default GPU clocks and report medians over repeated samples. These timings predate the host-runtime safety fixes.

6.1 Checked vs. Unchecked Kernels

To measure what safety costs on the GPU, we use a compute-bound and a memory-bound workload on the B200: GEMM and element-wise add. These runs use CUDA 13.3 and driver 595.58.03, with the benchmark process pinned to logical CPU 0.

For GEMM (Figure 6b), we compare Rust, unsafe Rust, cuTile Python, and cuBLAS over M=N=K in powers of two from 1024 to 32768 at `f16` precision, with tile shapes tuned once per size and shared across Rust and Python (as an autotuner would). Rust is the mapped-partition kernel of Listing 2. Unsafe Rust runs the same schedule with unchecked raw-pointer output access and a hand-written persistent loop. cuTile Python runs the same schedule on the same Tile IR backend. cuBLAS is the fastest of the algorithms cuBLASLt's

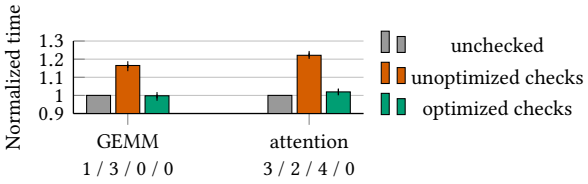


Figure 7. Bounds-check placement ablation, normalized to the unchecked arm. Under each kernel, its census from §4: discharged / launch / preheader / in place. Whiskers are p_{25}–p_{75} over rounds.

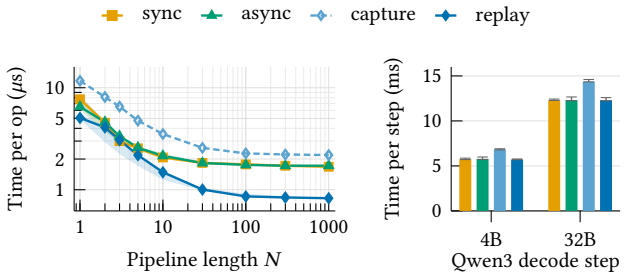
heuristics propose at each size. Unsafe Rust on the same backend isolates the cost of the safety machinery (a comparison with Triton [38] would measure backend code quality instead). cuBLAS and the hardware peak anchor the results in absolute terms. At M=N=K=8192, Rust reaches 2.11 PFlop/s, 94% of the B200's 2.25 PFlop/s dense `f16` peak and 98.0% of cuBLAS. Unsafe Rust matches it within 0.1%. The safe partition-index formulation therefore imposes no measurable cost over manual pointer access. cuTile Python reaches 2.06 PFlop/s (96.0% of cuBLAS), which places both frontends on the same backend performance envelope. Safe and unsafe Rust agree within 0.2% from 4096^3 up. At 1024^3 and 2048^3, where the kernels run for 9 and 17 μs, the safe kernel trails by 3% and 1%. cuTile Python, whose launch path costs more per call, trails Rust by 9% at 1024^3 (2–3% at element-wise sizes up to 2^{22}) and by 1–4% from 2048^3 up.

Element-wise add (Figure 6a) isolates memory-bound tile loads and stores. At $N = 2^{28}$, unsafe Rust and Rust both reach 6.97 TB/s and cuTile Python reaches 6.96 TB/s, near the 7.68 TB/s peak.

Figure 6c reports the first-use compilation cost separate from these steady-state results and shows it growing with tile size because larger GEMM tiles produce larger Tile IR kernels. An application that installs the on-disk kernel cache pays this cost once per machine.

6.2 Bounds-Check Placement

§4 claims that the cost of bounds safety depends on where each check runs. To measure that dependence, we compare three builds of the same kernel: *unchecked* compiles the accesses without checks; *unoptimized checks* evaluates every obligation at its access site, with discharge, launch preconditions, and hoisting disabled; and *optimized checks* is the shipped placement. The kernels are the persistent GEMM of Listing 2 (`f16`, M=N=K=8192, 128×128×64 tiles) and the causal prefill attention kernel of §4 (`f16`, q_{len}=8192, 32×32 tiles, Listing 3). The arms are interleaved in paired rounds on the RTX 5090 at default clocks, 11 rounds for GEMM and 9 for attention. Each round is normalized to its unchecked arm. All three arms produce bitwise-identical outputs.



(a) Per-op latency: lines are medians, bands p_{25}–p_{75}. Capture is construction plus one replay.

(b) Decode-step replay: medians with p_{25}–p_{75} whiskers.

Figure 8. Sync, async, and graph execution (f16, RTX 5090). (b) replays synthetic decode steps built from the Qwen3-4B (437 kernels) and Qwen3-32B (838 kernels) profiles.

Unoptimized checks cost 1.17× on GEMM and 1.22× on attention (Figure 7). Optimized placement runs at 1.00× and 1.02×. After placement, the GEMM kernel contains no check, and attention's four remaining checks run in the preheader of its inner loop, once per loop entry instead of once per iteration.

6.3 Sync, Async, and Graph Execution

We next measure how host-side launch overhead scales with the length N of a *pipeline* under the sync, async, and graph modes. A pipeline is a chain of N operations. N runs from 1 to 1000 on the RTX 5090 over a small element-wise $y = x \cdot g$ kernel on a length-2048 f16 tile, so launch overhead dominates. The pipeline-length sweep pins the benchmark process to logical CPU 8.

Sync runs a `.then()` chain of N launches with one `.sync()`, which isolates launch cost from synchronization cost. Async runs the same chain through `.await`, yielding the host task while the GPU runs, with the staged completion path of §5.1. Graph captures the pipeline and replays all N kernels with one driver call.

Figure 8a separates two host-cost regimes. Synchronizing after every kernel, with individual `.sync_on(stream)` calls, costs 6 to 7 μs per operation at every length and is left off the plot. Sync and async pay one launch per kernel and one wait per pipeline. With spin-polled completion they track each other at every pipeline length and converge to about 1.7 μs per operation (Appendix C). Graph replay removes the host-side per-kernel launch overhead and approaches the rate at which the GPU itself dispatches kernels, about 0.8 μs per operation on this hardware. The capture line is a graph built and replayed once. Graphs are meant for repeated replay of one sequence, so a single replay never repays the construction cost of about 1.4 μs per node.

Figure 8b measures the same execution modes on synthetic decode steps calibrated from the inference engine's profiles.

Table 2. Qwen3 single-request performance (`f16`, batch 1, medians; prefix caching disabled): decode throughput at its peak over the generation sweep and at generation length t_g=8192 (prompt length p_p=18), and prefill latency at p_p=8192 (t_g=36). The full sweeps with spreads are in Appendix D.

		decode (tok/s)		prefill (ms)
	engine	peak	t_g=8192	p_p=8192
RTX 5090	Grout	**170.9**	**154.7**	**318.5**
Qwen3-4B	vLLM	164.3	151.3	404.6
	SGLang	166.2	154.3	432.8
B200	Grout	**81.6**	**80.1**	513.8
Qwen3-32B	vLLM	78.8	77.5	**491.6**
	SGLang	77.7	76.5	507.1

From kernel-level profiles of the Qwen3-4B and Qwen3-32B decode steps (§6.4), each kernel class's per-step launch count and median duration drive a chain of calibrated element-wise kernels, which run with cold caches as their real counterparts do. Sync, async, and graph replay agree within 2% on both steps. Each step waits once at its end like an autoregressive loop. Within a step, the host submits launches faster than the GPU executes them and all three modes are bound by GPU time. Synchronizing after every kernel (not shown) is 33–38% slower. Graph capture is a one-time cost of 1.1–2.1 ms. At this scale the choice among sync, async, and graph is one of ergonomics and amortization.

Async yields the host thread once the spin budget expires. This helps when the host has concurrent work to overlap, as in tool calling, where the tool runs on a core that would otherwise be occupied by a synchronous wait, or interruptible voice-to-text, where the host prepares the next audio window while the device processes the current one.

6.4 Case Study: Qwen3 Inference Engine

Grout [20] is an open-source, lean batch-1 Qwen3 engine built on cuTile Rust. Every kernel is written in cuTile Rust except the dense projections, which dispatch to cuBLAS, as in the engines we compare against. Of its 25 kernels, 22 are safe end to end, including all attention and fused-norm kernels. Migrating the engine's twelve legacy unsafe kernels to their safe forms left prefill within 1% and decode within 0.2% of the unsafe versions on the RTX 5090, each form at its own tuned tile shapes, with identical outputs. The remaining three build schedules outside the mapped-partition family from six one-line unsafe blocks, each inside an otherwise safe kernel (§3). Prefill runs through a cached sequence of typed operations over a reusable tensor pool. The decode pass consists of a fused normalization and cache-update kernel, decode attention, and split-K merges. It is recorded once as `DeviceOp` graph nodes inside `CudaGraph::scope` and replayed for subsequent tokens.

Grout performs on par with the two most widely used open-source inference engines vLLM [25] and SGLang [42] (Table 2). We evaluate their batch-1 performance. Its decode throughput is at or above theirs at every point of the generation sweep on both platforms, by up to 7%. Its prefill latency is 11–28% below vLLM's on the RTX 5090. On the B200 it is 10–15% below vLLM's at short prompts, matches it at p_p=512 and 2048, and trails it by 4.5% at 8192. The remaining gap at long prompts on the B200 comes mostly from the prefill attention kernel. vLLM and SGLang run attention kernels hand-tuned and precompiled for the B200; Grout's is one kernel whose source is the same on both GPUs. The difference is in the code the backend generates on the B200's sm_100 architecture.

The memory bandwidth roofline $R = \beta/(W + \bar{K})$ is the generated-token rate when decode is bound only by device-memory bandwidth, for peak device-memory bandwidth β, model weights W in bytes, and average KV-cache bytes $\bar{K}$ read per generated token. At t_g=8192, decode runs at 75% (RTX 5090) and 67% (B200) of it. Every measured engine sits in that regime (Appendix D).

7 Limitations

The tile model supports sparse matrix multiplication, stencils, and convolutions as well as the dense tensor kernels evaluated here. Tile IR controls the mapping to SIMT threads, shared-memory layout, and thread-level synchronization. Code that requires explicit control of these mechanisms is written in CUDA C++ and composed through host-side FFI.

The current safe tensor API requires mutable outputs to be partitioned among tile programs before launch. It does not provide shared atomic access to tensor outputs, so shared-output reductions currently require `unsafe` atomic operations. Data-dependent scatters and histograms also require `unsafe` operations when their writes cannot be confined to the assigned partitions. Additional safe tensor types could support these patterns; for example, a type permitting shared atomic updates could provide a safe interface for cross-program reductions.

Bounds-check placement moves a check only as far as its operands and the control flow around it allow. A check under a runtime conditional, or in a loop body with an early exit, stays at the access site, because the access may not execute on every iteration. A check also stays there when our analysis cannot bound a non-affine index. Checks whose operands are decoded from the partition index of the current sub-tensor stop at the enclosing loop's preheader, and checks whose operands are loaded from device memory never leave the kernel. A launch precondition is unconditional: it tests only whether the check's operands are known at launch, not whether the access would execute, so a launch can be rejected for a violation the kernel would not have committed.

Three implementation limitations motivate the future work in §9. The partition-map family covers the grouped schedules of §3.2. Three of Grout's 25 kernels use six one-line view constructions to express schedules outside this family. A general mapping function would let these kernels express their schedules without unsafe view construction. A kernel body is the subset of Rust that the JIT lowers from the source AST, type-checked by `rustc` but not compiled by it. The shape types rely on the shadow encoding of §3.1 until Rust supports array-valued const parameters directly.

The evaluation has a fixed scope. Every inference measurement is a single request at batch 1 in f16 on one model family, Qwen3, on two GPUs at default clocks. The safe-versus-unsafe comparisons of §6.1 and §6.2 cover GEMM, element-wise add, and attention, and the engine-wide comparison of §6.4 covers its remaining kernels once. Microbenchmarks report medians over repeated samples, and the placement ablation reports medians and interquartile ranges over paired rounds.

8 Related Work

GPU programming in Rust. Rust-CUDA, rust-gpu, and cuda-oxide [13, 15, 32] compile Rust to GPU device code and established Rust as a viable kernel language, and GPU Offload in Rust [14] integrates GPU code generation into `rustc` through LLVM Offload and carries ownership across host-device transfers. CubeCL [39] adds a SIMT-style kernel model across backends, and Candle and Burn [11, 19] provide safe host-side tensor APIs for ML applications. cuTile Rust is complementary to all of these: It preserves Rust's `&mut`/`&` aliasing contract across the launch boundary and into a tile-based kernel body (§3), so the safety argument covers the kernel as well as the host. RustBelt [21] gives Rust's guarantees formal foundations, Rudra [3] shows that `unsafe` Rust is an ecosystem-scale bug source, and Abdi et al. [1] ask when Rust parallelism is fearless and zero-cost. §6 answers that question for tile kernels.

Tile-based GPU programming. Triton [38] popularized tile-level GPU programming. Pallas, ThunderKittens, and TensorIR [16, 36, 37] bring tile-level models to JAX, CUDA C++, and tensor compilers. CUDA Tile C++ and cuTile Python [30, 31] share cuTile Rust's Tile IR backend. In cuTile Rust, tensor and partition types let Rust check ownership across host code and GPU kernels.

Safe heterogeneous programming. Descend and Mojo [24, 28] bring ownership to GPU programming through new languages. Descend uses type-level tracking of the thread hierarchy and memory regions for SIMT code. cuTile Rust works inside Rust's existing type system and targets the tile model, where the ownership invariant is tractable to check. GPUVerify [7] verifies kernels after the fact, Faial [12] checks data-race freedom of SIMT kernels compositionally, and Regent, Halide, Futhark, and Prism [4, 18, 34, 35] structure parallelism through regions, schedules, purity, or thread granularity. cuTile Rust constrains which memory a program may access and in what order, by construction in the launch and tile APIs.

Deferred execution. Task-based runtimes such as StarPU, PaRSEC, and Legion [2, 6, 9] defer GPU work behind runtime dependency tracking over registered data. `DeviceOp` expresses dependencies through typed operation composition, with Rust checking ownership of the operands.

Bounds-check placement. Bounds-check elimination in optimizing and JIT compilers proves checks away or hoists them within the compiled program by range analysis and loop-invariant code motion [8, 17, 40]. The placement of §4 applies the same reasoning at kernel specialization time, with the launch parameters known, and adds a tier outside the kernel: a launch precondition evaluated once per launch.

9 Conclusion

cuTile Rust extends Rust's ownership discipline across the launch boundary and into GPU kernels: Mutable parameters become disjoint partitioned accesses, immutable references become shared tensor views, and the generated launch interface carries that contract between host and device. This contract supports the data-race-freedom theorem. Bounds-check placement brings the evaluated checked kernels to parity with their unchecked versions. Asynchronous execution matches synchronous execution in the pipeline benchmarks, and graph replay reduces per-launch cost. On GEMM, the safe kernel matches a raw-pointer variant and reaches 98% of cuBLAS on the B200. Grout, a Qwen3 inference engine developed in collaboration with Hugging Face and built on cuTile Rust, reaches single-request throughput on par with vLLM and SGLang on the RTX 5090 and the B200.

The future direction of this work is a safe SIMT model that composes with tile programming, reusing the same tensor types, partitioning, and launch contract across arbitrary levels of the thread hierarchy. SIMT code is hard to make safe statically because hundreds of threads in a thread block share memory and synchronize explicitly; safe code must know which threads execute an instruction together and which values are uniform across them. A second direction is the general partition mapping function of §3.2, which would cover the three Grout kernels that still carry unsafe blocks. Two enabling directions lie in Rust itself: the stabilization of array-valued const parameters for shape types; and lowering kernels from rustc's mid-level IR (MIR) once it is stable, to enable the complete Rust language for cuTile Rust's tile programming model.

Acknowledgments

We thank Gonzalo Brito for design discussions and support for tile programming in Rust, Simon Cooksey (Tile IR memory model author) for confirming that our data-race-freedom argument is correct, Jason Knight for support and direction, Nihal Pasham for early contributions, and the PSA and ARG research groups and the Tile IR and CUDA teams for support. We also thank Jed Brown and Sarah El Kazdadi for discussions on safe GPU programming in Rust.

References

[1] Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C. Jeffrey. 2024. When Is Parallelism Fearless and Zero-Cost with Rust?. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2024, Nantes, France, June 17-21, 2024*, Kunal Agrawal and Erez Petrank (Eds.). ACM, 27–40. https://doi.org/10.1145/3626183.3659966

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[3] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP 2021*. ACM, 84–99. https://doi.org/10.1145/3477132.3483570

[4] Manya Bansal, Daniel Sainati, Joseph W. Cutler, Saman Amarasinghe, and Jonathan Ragan-Kelley. 2025. Modular GPU Programming with Typed Perspectives. arXiv:2511.11939 [cs.PL] https://arxiv.org/abs/2511.11939

[5] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (2017), 48–54. https://doi.org/10.1145/3015146

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[7] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012*. ACM, 113–132. https://doi.org/10.1145/2384616.2384625

[8] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/349299.349342

[9] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[10] Eric Buehler. 2024. mistral.rs: Blazingly fast LLM inference in Rust. https://github.com/EricLBuehler/mistral.rs.

[11] Burn Contributors. 2023. Burn: Dynamic Deep Learning Framework in Rust. https://github.com/tracel-ai/burn.

[12] Tiago Cogumbreiro, Julien Lange, Dennis Liew, and Hannah Zicarelli. 2021. Checking Data-Race Freedom of GPU Kernels, Compositionally. In *Computer Aided Verification (CAV)*.

[13] Riccardo D'Ambrosio. 2021. Rust-CUDA: CUDA programming in Rust. https://github.com/Rust-GPU/Rust-CUDA.

[14] Manuel S. Drehwald, Marcelo Domínguez, Kevin Sala, Alán Aspuru-Guzik, and Johannes Doerfert. 2026. GPU Offload in Rust: Portable, Safe, and Fast. arXiv:2608.13759

[15] Embark Studios. 2023. rust-gpu: Making Rust a first-class language and ecosystem for GPU shaders. https://github.com/EmbarkStudios/rust-gpu.

[16] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023*. ACM, 804–817. https://doi.org/10.1145/3575693.3576933

[17] Rajiv Gupta. 1990. A Fresh Look at Optimizing Array Bound Checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 272–282. https://doi.org/10.1145/93542.93581

[18] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 556–571. https://doi.org/10.1145/3062341.3062354

[19] Hugging Face. 2023. Candle: Minimalist ML Framework for Rust. https://github.com/huggingface/candle.

[20] Hugging Face. 2026. Grout: A cuTile-Rust Qwen3 Inference Engine. https://github.com/huggingface/grout/tree/cutile-rs-paper-v2. Tag cutile-rs-paper-v2.

[21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34. https://doi.org/10.1145/3158154

[22] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1–16.

[23] Oleg Kiselyov and Chung chieh Shan. 2007. Lightweight Static Capabilities. *Electronic Notes in Theoretical Computer Science* 174, 7 (2007), 79–104. https://doi.org/10.1016/j.entcs.2006.10.039

[24] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. 2023. Descend: A Safe GPU Systems Programming Language. In *Proceedings of the ACM on Programming Languages*, Vol. 7. ACM, 1–23. https://doi.org/10.1145/3591269

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626. https://doi.org/10.1145/3600006.3613165

[26] John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*. 24–35. https://doi.org/10.1145/178243.178246

[27] Daniel Lustig, Sameer D. Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 257–270. https://doi.org/10.1145/3297858.3304043

[28] Modular Inc. 2023. Mojo: a programming language for heterogeneous compute. https://www.modular.com/mojo.

[29] NVIDIA. 2024. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[30] NVIDIA. 2025. CUDA Tile MLIR. https://github.com/NVIDIA/cuda-tile.

[31] NVIDIA. 2025. cuTile: CUDA Tile Programming in Python. https://github.com/NVIDIA/cutile-python.
[32] NVIDIA. 2026. cuda-oxide: A custom rustc backend for compiling GPU kernels in pure Rust. https://github.com/NVlabs/cuda-oxide.
[33] NVIDIA. 2026. cuTile Rust: Tile-Based GPU Kernel Programming in Safe Rust. https://github.com/nvlabs/cutile-rs/tree/cutile-rs-paper-v2. Tag cutile-rs-paper-v2.
[34] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530. https://doi.org/10.1145/2491956.2462176
[35] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: a high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015*. ACM, 81:1–81:12. https://doi.org/10.1145/2807591.2807629
[36] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. 2024. ThunderKittens: Simple, Fast, and Adorable AI Kernels. arXiv:2410.20399 [cs.LG] https://arxiv.org/abs/2410.20399
[37] The JAX Authors. 2023. Pallas: a JAX kernel language. https://docs.jax.dev/en/latest/pallas/index.html.
[38] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. https://doi.org/10.1145/3315508.3329973
[39] Tracel AI. 2024. CubeCL: Multi-platform high-performance compute language extension for Rust. https://github.com/tracel-ai/cubecl.
[40] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2007. Array Bounds Check Elimination for the Java HotSpot Client Compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ)*. 125–133. https://doi.org/10.1145/1294325.1294343
[41] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating Permissions from Data in Rust. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473597
[42] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems, NeurIPS 2024*. https://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html

A Data-Race Freedom

cuTile Rust's safe tensor API wraps Tile IR's load and store operations. Beyond data-race freedom, those operations carry only Tile IR's *validity* requirement, in-bounds access to live, initialized memory, which is the ordinary memory-safety obligation of the tensor API. This section discharges the one requirement specific to Tile IR's weakly ordered memory model, data-race freedom, for any kernel and launch written in safe Rust. The unsafe tiers of §3 (`unchecked_accesses` and raw pointers) carry the requirement manually and are excluded. The argument's trusted base is the proc-macro-generated launcher and kernel entry, which realize the disjoint partitions and token threading of §3.2 and §3.3; the runtime JIT, which correctly lowers the kernel to Tile IR; the Tile IR backend, which honors the token ordering it is given; the tensor and execution runtimes, which preserve storage validity and access rights; and CUDA's stream and graph execution ordering. We assume these components and do not prove them.

Tile IR memory model. We use Tile IR's definitions [30], whose *morally strong* relation follows the formalized PTX memory model [27]. A kernel runs a grid of tile programs (§2.1), each a single logical thread that executes as one *tile block*, the granularity at which the model orders memory. Two accesses *conflict* when they touch the same location and at least one is a write. Ordering is carried by tokens: An operation *waits-for* another when it depends on a token the other produced, and *restricted program order* is the intersection of program order and this token order. Two same-location accesses are *morally strong* when they are related in restricted program order, or each names a scope that contains the other's tile block. A *data race* is a conflicting pair that is not related by happens-before and not morally strong. A program containing a data race has undefined behavior.

Safe Rust. Safe tensor accesses supply two properties the definition needs. First, *alias-XOR-mutability across tile programs*: A tile program writes tensor memory only through its partition view, and the borrow checker keeps that view's sub-tensors disjoint from every other program's accessed memory. This rests on the partition map being a function from output sub-tensors to tile programs (§3.2): Each sub-tensor has exactly one owner. The launch grid is derived from the partition shape, so each tile program is assigned its own sub-tensor, and mutable partitions can be constructed in safe code only through the partition APIs, starting from an owned tensor or an exclusive borrow. Mutable views map distinct in-bounds coordinates to distinct memory locations. For the mapped case, the launcher validates the tile-program count against the partition grid, and accesses go through branded `PartitionIndex` values (§3.2), which the front end accepts only from the program's own map [23, 41], so every store lands in bounds on a sub-tensor the map assigned to that program. An immutable `&Tensor` is read-only and cannot alias any partition, because a partition derives from an owned tensor or an exclusive borrow (§3.1). Second, *ordering within a tile program*: The compiler threads a token chain through every load and store on a `&mut` view (§3.3), so accesses from distinct operations are related in both program order and token order.

Safe accesses to shared scalar globals are non-weak and use device or system scope. Their storage cannot alias tensor views in safe code.

Theorem A.1. *Every execution of a kernel written in safe cuTile Rust is data-race-free under Tile IR's memory model.*

Proof. Consider a conflicting pair a, b. At least one, say a, is a write. If a accesses a global, so does b. Each access specifies a scope that contains the other's tile block, so the pair is morally strong. Otherwise, a is a tensor write. In safe code, tensor stores exist only on partition views, so a writes through the partition view of the tile program that issued it. Element writes within one tile store cannot conflict, so a and b come from distinct tile operations. If b is issued by a different tile program, alias-XOR-mutability makes a's partition view disjoint from every location b can access, so a and b cannot share a location, contradicting conflict. If b is issued by the same tile program through an immutable view, the same contradiction follows, because an immutable view aliases no partition. In the only remaining case, a and b both access the issuing program's `&mut` view, so they are related in program order and token order, hence in restricted program order, and are therefore morally strong. Every conflicting pair is thus morally strong, so no execution contains a data race. □

For tensor accesses, this is the argument that makes ordinary Rust data-race-free [21]: Alias-XOR-mutability removes conflicts between threads and ordering removes them within a thread, with Tile IR's token order ("waits-for") in the role that sequenced-before plays in the C++ and Rust memory models. The store-index bug of §3.3 is the canonical execution it excludes: A store through the plain partition view has no programmer-chosen destination index, so the conflicting write to another program's sub-tensor is unconstructible.

Host execution. Across launches, the host runtime is trusted to preserve operand storage and access rights until device work completes, including when operations are cancelled or forgotten. Conflicting tensor accesses are excluded or ordered by completion, stream order, or graph dependencies. Global accesses remain morally strong at device or system scope. Under this runtime contract, data-race freedom extends to safe host programs.

B Bounds-Check Census

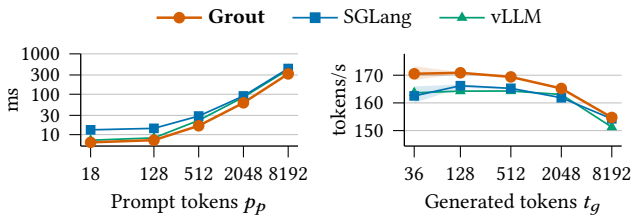
Table 3 lists every bounds-check obligation of the attention kernel of §4 and where the compiler resolves it. GEMM's four obligations are walked in §4. GEMM's dispositions were read from the compiled launcher's check list and confirmed by launching each violating shape, which the host rejects with the predicate's own message while the kernel's Tile IR contains no assert. Attention's dispositions were read from the Tile IR dump, in which the four hoisted asserts sit immediately before the inner loop, and from the launcher's check list. The attention census is the same at every tile shape measured. The evidence is in the paper's repository under `benchmarks/exp4_hoisting`. In Listing 3, the literal third coordinates are within the static head dimension D and are discharged at specialization. The `q` coordinates come from the output's map and become launch preconditions. The four checks on the `k` and `v` loads are hoisted to the preheader of the `j` loop: `kvh` is loop-invariant there, and `j` is affine with extreme $tc-1$. Both `kvh` and `tc` are decoded from the partition index inside the persistent loop, so the `j` preheader is the earliest point at which they exist.

Table 3. Causal prefill attention (Listing 3): nine obligations, three discharged at specialization, two launch preconditions, four hoisted to the preheader of the `j` loop. q_i abbreviates $dim(q, i)$. D is the static head dimension.

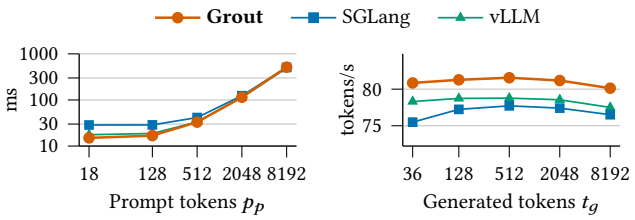
access	index	where	why
q axis 0	qm	launch	$\lceil out_0/BM \rceil \le \lceil q_0/BM \rceil$
q axis 1	qh	launch	$\lceil out_1/1 \rceil \le \lceil q_1/1 \rceil$
q axis 2	0	discharged	literal within D
k,v axis 0	kvh	preheader	loop-invariant
k,v axis 1	j	preheader	affine; tests $tc-1$
k,v axis 2	0	discharged	literal within D
store	branded	none	proven by the brand

C Completion-Path Costs

Table 4 isolates what each completion path costs a single pipeline. The three paths run in rotation for ten rounds of 50 samples at default clocks. A round's value is its lower quartile, which selects the same one of the workstation's two timing states for all three paths, and each entry is the median over rounds (for async, of that round's difference to sync). The launch-share bands come from the decode profiles of §6.4. In the first two rows, the work remaining at the await fits the spin budget and spin-polled async matches sync within 0.2 μs: The wait resolves at the poll site. From the third row on, the spin expires and both async paths pay the reactor's wake path, about 2 μs over sync, whether or not the spin ran. The expired spin consumes CPU time and adds no latency of its own. The fraction column shows the amortization: The wake path is a fifth of an 8 μs launch and wait, two percent at 123 μs, and about one percent at 290 μs, while the spin removes it for the short kernels where it matters. The launch columns place the Qwen3 decode kernels in these bands: About 60% of launches are shorter than the first row, where the spin path costs under 0.1 μs and the reactor path alone would cost a fifth, and nearly all the rest fall in the next three rows, where the default spin keeps async within 2.4 μs, under four percent, of sync. A decode step chains 437 or 838 such kernels with one await at the end (Figure 8b), so the step pays the wake path once per 6 to 12 ms of GPU work, and the replayed steps agree within the 2% measurement spread. For an isolated pipeline the budget is therefore harmless at any kernel size. The default setting gives individual pipelines the same latency as sync (Figure 8a).



(a) RTX 5090 / Qwen3-4B.



(b) B200 / Qwen3-32B.

Figure 9. Qwen3 single-request performance (f16, batch 1, median with IQR shading; prefix caching disabled): Prefill latency vs. prompt length p_p (t_g=36) and decode throughput vs. generation length t_g (p_p=18).

```
for index in out.iter_indices() {  // persistent loop
  let [qm, qh, _] = index.coords();
  let kvh = qh / grp;              // kv head of qh
  let tq = q_part.load([qm, qh, 0]);
  let tc = ceil_div(min(q0 + (qm + 1) * BM, kv_len), BN);

  for j in 0..tc {          // 4 checks hoist to here
    let tk = k_part.load([kvh, j, 0]);
    // ... QK^T, causal mask, online softmax ...
    let tv = v_part.load([kvh, j, 0]);
    // ... accumulate ...
  }
  out.store(acc, index);
}
```

Listing 3. The obligation-carrying lines of the causal prefill attention kernel of §4: three loads over three axes inside a persistent loop over the output map, with the inner `j` loop over key/value tiles.

Table 4. Single-pipeline completion-path cost (RTX 5090): one launch and wait of the element-wise kernel of §6.3 over d elements. Columns: sync wall time; async's increase over sync with the default 20 μs spin and with the spin disabled, in μs and as a fraction of sync; the share of Qwen3-4B and Qwen3-32B decode launches whose kernels fall in the duration band ending at each row's sync time.

	sync	async, over sync		decode launches	
elements d	μs	spin	no spin	4B	32B
2^{21}	7.8	+0.0 (0.3%)	+1.6 (21%)	59%	62%
2^{24}	20.3	+0.1 (0.7%)	+1.9 (9.2%)	8%	23%
$17 \cdot 2^{20}$	43.3	+1.7 (3.9%)	+1.7 (4.0%)	25%	8%
2^{25}	122.9	+2.4 (1.9%)	+2.4 (1.9%)	8%	8%
$9 \cdot 2^{23}$	291.1	+2.5 (0.8%)	+2.5 (0.9%)	0%	0.1%

D Qwen3 Inference Sweeps

Figure 9 shows the full prompt-length and generation-length sweeps behind Table 2 of §6.4, with interquartile spreads. All measurements are single-request, batch 1. On the B200, each cell is the median of 10 repetitions. On the RTX 5090, each cell is the median of 10 repetitions up to p_p, t_g=512 and 3 repetitions beyond. Prefill latency is each engine's per-request timing. vLLM's offline API does not expose it, so we use time-to-first-token from a `max_tokens=1` probe. vLLM 0.18.0 and SGLang 0.5.9 run the same f16 weights with CUDA graphs enabled and prefix or radix caching disabled. Every cell follows three warmup requests and uses greedy decoding with the end-of-sequence token ignored to fix the generation length. At t_g=8192 decode reaches 75%, 73%, and 74% of the memory bandwidth roofline of §6.4 on the RTX 5090 and 67%, 65%, and 64% on the B200 for Grout, vLLM, and SGLang, and each engine's fraction moves by at most 3 points across the generation sweep.