Exo-GPU: Safe, Imperative, User-schedulable Programming for Tensor Cores

ANONYMOUS AUTHOR(S)

Modern GPUs require not only SIMT-style parallelism but also software-managed concurrency between compute and data movement to reach maximum performance. Performance engineers must reason about subdividing work into the hierarchy of computation resources (threads, warps, warpgroups, blocks, clusters), and, in many cases, also must use asynchronous tensor core and memcpy instructions on different levels of the memory hierarchy (registers, tensor core accumulators, shared memory, global memory). Unlike CPUs, where out-of-order execution is managed by hardware and hidden from programmers, GPUs expose explicit instruction reordering to software through these asynchronous instructions. Well-established GPU programming languages generally offer either direct low-level control without safety guarantees (e.g., CUDA C++ inline assembly or intrinsics) or easier-to-analyze, high-level abstractions (e.g., Triton's tile-based model) that hide asynchronous instructions in the compiler backend, which may prevent performance engineers from maximizing performance by tuning critical details.

We propose Exo-GPU, an imperative, low-level language that creates minimal abstraction over CUDA. Our key idea is to treat parallelism and synchronization as mere annotations on sequential code rather than as fundamental control flow primitives, enabling verification that these constructs do not alter the program semantics. The benefit is two-fold: programmers can reason about code without hidden control flow or mutation, while allowing the Exo-GPU compiler to verify *sequential-parallel equivalence*—guaranteeing that parallel execution is functionally equivalent to its sequential interpretation. We used Exo-GPU to author GEMM kernels for the H100 GPU, using wgmma, TMA, and split-k. Our kernels achieved over 80% of theoretical peak on large problem sizes, in some cases outperforming the vendor-provided CUBLAS library.

1 INTRODUCTION

Starting with Tensor Cores introduced in NVIDIA's Volta generation of GPUs [13], the menu of tensor-specialized features in NVIDIA GPUs has expanded to encompass automated bulk movement of multidimensional data, distributed shared memory, and new asynchronous tensor cores supporting large-scale (up to 64 × 256) matrix accumulation in a single instruction [18]. This poses ever-increasing challenges for authoring correct, top-performing CUDA code. Not only must algorithms be rewritten to migrate work from traditional SIMT-style code to target these specialized instructions, but also, these instructions expose explicit out-of-order instruction execution to the programmer, moving synchronization that was traditionally the responsibility of the hardware (e.g., through scoreboarding) to the programmer.

High-level kernel programming languages, such as Triton [24], provide a popular approach for taming this complexity. Triton's tile-based, automatically-parallelized programming model reduces the risk of programmer error by moving responsibility for correct usage of bulk data movement and synchronization instructions into the compiler. However, in practice, groundbreaking algorithms, such as the original FlashAttention 3 [21], are often written directly in CUDA C++ for greater control. On the one hand, CUDA C++ provides direct control over features such as asynchronous data movement and warp specialization required for optimal GPU usage. On the other hand, CUDA C++ exposes programmers to a variety of potential bugs, which we broadly categorize as:

1. *Dataflow Logic Bugs*: Program optimization usually requires changing the storage or computation order of intermediate values, e.g. caching data in a faster memory, or reordering instructions to hide latency. A flawed optimization, such as an indexing bug in copying a tile, or exchanging two statements that don't commute safely, can change the functional behavior of the program.
2. *Instruction Logic Bugs*: Accelerator instructions won't work as intended if not used as specified by the hardware vendor. For example, `wgmma` tensor core instructions require data in a certain input format and uniform execution by a full warpgroup (128 aligned threads).

(3) *Inter-Thread Synchronization Bugs*: Garden-variety synchronization bugs involve two threads accessing the same memory location (with one access being a write) without ordering enforced by synchronization.

(4) *Intra-Thread Synchronization Bugs*: If one thread issues an asynchronous (async) instruction, and that *same thread* later issues another instruction (async or not) that operates on the same memory location, overlapped or reordered instruction execution can undermine correctness.

Our goal is to provide a programming model that gives CUDA-like control *without* accepting the risk of these bugs as a fact of life. Existing CUDA C++ libraries such as CuTe [14] and ThunderKittens [22] provide close-to-the-metal abstractions that minimize the risk of logic bugs, particularly data movement and instruction input format bugs; however, none of them support large-scale program analysis and synchronization checking of arbitrary input programs.

As a starting point for the Exo-GPU language design, we postulate that programs with *sequential semantics* are far easier to analyze, both by humans and by compilers. We define program behavior using standard sequential semantics, treating language features for parallelism and synchronization–including parallel for-loops across CUDA thread hierarchies and barrier statements–simply as *annotations* on sequential code. This design provides two key benefits: programmers can intuitively reason about their code without hidden control flow or side effects, and more importantly, compilers can verify whether synchronizations are legal; that is, they can verify these annotations do not alter the program behavior relative to its baseline sequential interpretation.

The Exo-GPU program, interpreted sequentially, serves as as the semantic baseline—defining *what* the user intends to compute. Parallelism annotations specify *which threads* execute which statement instances, while synchronization constructs specify *how* the user intends to uphold baseline behavior. This raises a question: what degree of safety and control does Exo-GPU provide? We define safety as *sequential-parallel equivalence*–ensuring that the parallel interpretation of a program matches the sequential one. Parallelism is explicitly controlled by the programmer's annotations, rather than being hidden and automated by the compiler. The Exo-GPU compiler's role is limited to *verifying* that the user-provided synchronization ensures equivalence.

Through collective analysis (Section 4), we annotate each statement with a *collective tiling*, specifying which thread(s) execute each statement instance. This enables static analysis of the thread assignment counts, ensuring instructions with thread convergence requirements (e.g. warp MMA) are used correctly. To ensure sequential-parallel equivalence, we simulate execution on an *abstract machine* (Section 5). The Exo-GPU program converts to an Abstract Machine program where data variables store access histories (reads and mutations) with their thread and async instruction *visibility*, rather than numeric values. Synchronization statements become meaningful only in the abstract machine, conditionally expanding the visibility of all prior recorded accesses.

We built Exo-GPU as an extension to Exo [7, 8], a user-schedulable language providing composable, checked rewrite rules for transforming sequential programs. Exo lacks GPU support, and we significantly extended its frontend language design, static analysis, semantics, and codegen to enable safe parallel program transformation. Treating parallel constructs as annotations over a sequential semantic baseline enabled us to reuse Exo's over 60 verified rewrites as-is, which allowed us to transform a simple sequential GEMM program into a complex GEMM kernel for the NVIDIA Hopper architecture, with every step of transformation verified via a *chain of equivalence*: Exo verified equivalence between the original p_1 and the final p_n while ignoring all the synchronization constructs, and Exo-GPU verified functional equivalence between the parallel and sequential interpretations of p_n, checking the legality of the parallel and synchronous annotations (Section 2.3). Combining Exo's pre-existing checked rewrite rules with Exo-GPU's new language features and synchronization checking, we created a GEMM kernel achieving over 80% of theoretical peak for large problem sizes, in some cases outperforming the vendor-supplied CUBLAS library (Section 6).

2 EXAMPLES AND SYSTEM OVERVIEW

2.1 Exo-GPU Language Overview

Exo-GPU is built on fundamental design goals of safe, performance transparent, and imperative GPU programming. Unlike other GPU abstractions that hide complexity, we make threads and most CUDA instructions explicit in the IR. We introduce three first-class frontend language features derived from our design principles: (i) parallel loops that explicitly map work to blocks/warps/threads, (ii) a distributed memory model with sharding and explicit memory space annotations, and (iii) explicit synchronization, including fences, split-barriers, and asynchronous operations.

Parallel loops. Exo-GPU exposes three kinds of loops. (1) `seq`(c_{lo}, c_{hi}) is a sequential loop that iterates i from c_{lo} to $c_{hi}-1$. (2) `cuda_tasks`(lo, hi) distributes iterations across CUDA clusters; on pre-`sm_90` (H100) GPUs a cluster coincides with a single CTA (thread block). (3) `cuda_threads`($lo, hi,$ `unit` $= \tau_u$) assigns iterations to subsets of threads inside the current cluster. The `unit` takes a parameter τ_u that specifies the shape of the participating thread set but not concrete indices—for example, $\tau_u =$ `cuda_warp` denotes contiguous 32-thread groups with thread IDs 0–31, 32–63, 64–95, etc. (inclusive), while $\tau_u =$ `cuda_thread` denotes a single thread (see Figure 11).

Each loop body, conditional branch, `with`-block, or procedure defines a *program scope*. As part of *collective analysis* (Section 4), we annotate each scope with a *collective tiling* $\omega \in \Omega$. This describes a mapping between the control variable values and the *thread collective* (set of threads) assigned to execute statement instances in the scope (i.e. which threads execute which loop iterations). We say that a scope is a τ_u-scope when τ_u accurately describes the thread sets produced by the mapping. In the inset figure, the outermost parallel loop `cuda_tasks(0,37)` creates `scope-1` (which is a cluster scope, and, assuming `clusterDim=1`, also a CTA scope); nesting `cuda_threads(..., unit=cuda_warp)` creates `scope-2` (warp scope); and nesting `cuda_threads(..., unit=cuda_thread)` creates `scope-3` (single-thread scope). Statement instances in `scope-3` are executed by one thread.

```
for blk in cuda_tasks(0, 37):
  # scope-1 (CTA-scope)
  for w in cuda_threads(0, 4,
                 unit=cuda_warp):
    # scope-2 (warp-scope)
    for t in cuda_threads(0, 32,
                 unit=cuda_thread):
      # scope-3 (thread-scope)
```

Distributed Memory: Exo-GPU exposes explicit allocation statements for data and barrier variables, annotated by memory type, e.g. `x : f32[8] @ CudaRmem` allocates a data array `x` of type `f32` and size 8 in GPU register memory. Allocations can be *distributed*: a single logical object may be sharded across threads at multiple CUDA levels (e.g., per-thread register shards or per-CTA-in-cluster shared-memory shards). Threads must access only their owned shards except via explicit communication instructions (e.g., warp shuffles) or TMA multicast. This sharding is not annotated explicitly at the allocation, but deduced from the usage pattern of the variable; all deductions must be consistent, as enforced by distributed memory analysis (Section 4.1).

Inset right illustrates the motivation for this design. If the first loop was executed sequentially, `tmp` would hold `A[31]` after the first loop, so the second loop broadcasts that value to every `B[i]`. Languages like CUDA that implicitly duplicate local variables per thread would interpret the same code as a vector copy (`B[i] = A[i]`). While not necessarily error-prone, this design would be inappropriate for our goal of defining program behavior with sequential semantics. Hence, in Exo-GPU, storing a temporary value per-thread requires a sharded allocation of `tmp`; as written, the program is rejected by the analysis because a register-resident `tmp` cannot be read by different threads without (for example) a warp shuffle. If a user intends to express a vector copy (`B[i] = A[i]`) through register `tmp`, `tmp` requires explicit *distrubuted* allocation per thread (`tmp:f32[32]`) and the accesses must be per-thread (`tmp[i]`).

```
tmp: f32 @ CudaRmem  # Register
for i in cuda_threads(0, 32,
             unit=cuda_thread):
  tmp = A[i]
for i in cuda_threads(0, 32,
             unit=cuda_thread):
  B[i] = tmp
```

Explicit Synchronization: Many GPU instructions require additional explicit synchronization, but overly conservative barriers can cancel the performance benefits of concurrency. Exo-GPU provides two forms of synchronization: a blocking barrier via Fence, and split barriers via paired Arrive/Await. As with other statements, all synchronization statements execute in program order.

A Fence causes all threads in the current thread collective (warp, CTA, or cluster) to rendezvous. At warp scope it maps to __syncwarp; at CTA scope it maps to __syncthreads (as shown in the right-hand code inset). A Fence takes two synchronization–timeline parameters, τ_s^{pre} and τ_s^{post} (Figure 16). These timelines specify which memory operations (sync and/or async) before and after the fence must be ordered, and the compiler emits the minimal combination of synchronization and memory-fence instructions required to satisfy them. The cuda_in_order timeline specifies that the effects of async instructions are *not* synchronized, but only regular synchronous instructions. If we were to, for example, replace the loads into shared memory with cp.async operations, the first parameter must be set to Sm80_cp_async so that the copy operations preceding the fence are completed before execution proceeds.

```
# Code at CTA-scope, with blockDim=128
buf: f32[128] @ CudaSmemLinear
for tid in cuda_threads(0, 128, unit=cuda_thread):
  buf[tid] = gmem[tid]
# __syncthreads
Fence(cuda_in_order,cuda_in_order)
for tid in cuda_threads(0, 128, unit=cuda_thread):
  accum: f32 # Each thread sums up buf[...]
  accum = 0
  for i in seq(0, 128):
    accum += buf[i]
```

Split barriers use a shared barrier-typed variable z, declared as z: barrier @ π_b, where π_b selects the completion mechanism (commit group, mbarrier, or cluster sync). Programmers express a split barrier statement pair with Arrive(τ_s^{pre}) >> z and Await(z, τ_s^{post}). The statement pairing is determined by the shared barrier variable z. A matching pair orders memory operations prior to Arrive (filtered by τ_s^{pre}) with those after Await (filtered by τ_s^{post}). The right-hand inset illustrates replacing the earlier Fence example with an mbarrier-based split barrier.

```
for tid in cuda_threads(0, 128,
                        unit=cuda_thread):
  ...
bar: barrier @ CudaMbarrier
Arrive(cuda_in_order) >> bar
# ... insert work here
Await(bar, cuda_in_order)
for tid in cuda_threads(0, 128,
                        unit=cuda_thread):
  ...
```

2.2 Exo-GPU Examples with Erroneous Synchronization

To motivate our language design, we build up a series of increasingly subtle, hard-to-identify inter- and intra-thread synchronization bugs; all would be rejected by synchronization checking (Section 5). Variables prefixed with some_* are allocated outside the code snippet, some_in_order_op represents non-async work, and example_tma_multicast and example_wgmma are simplified placeholders for actual TMA (tile copy) and wgmma (tensor core) instructions.

The first example illustrates an inter-thread synchronization bug involving a cluster with 2 CTAs:

```
B: f32[2, 256, 32] @ CudaSmemLinear # Distributed mem: B[0,:,:] on CTA 0; B[1,:,:] on CTA 1
for cta in cuda_threads(0, 2, unit=cuda_cta_in_cluster):
    some_in_order_op(B[cta, :, :])
    Fence(cuda_in_order, cuda_in_order) # __syncthreads
example_tma_multicast(B[:,:,:], some_gmem[:,:])  # Copy some_gmem[:,:] to B[0,:,:] and B[1,:,:]
```

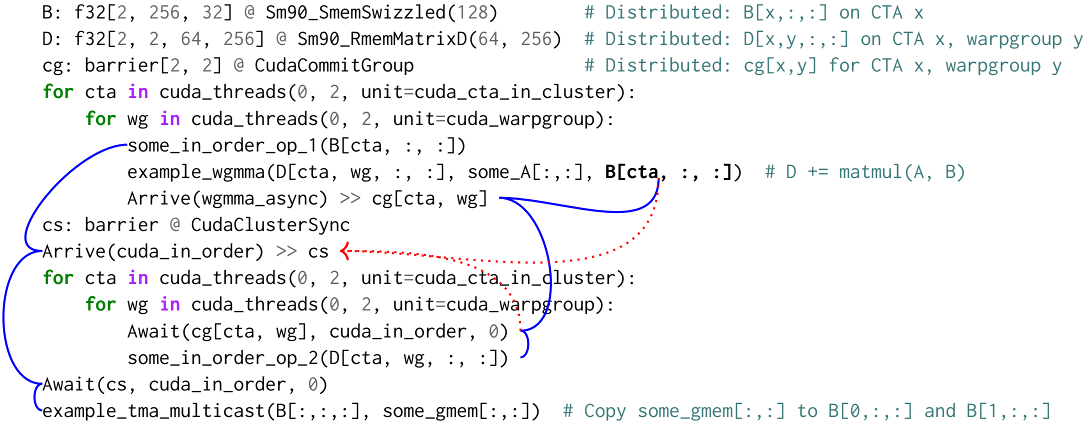
The example_tma_multicast instruction is implemented by threads in the 2 CTAs cooperating to fill each others' shared memory; however, this cooperation does not entail implicit synchronization between the two CTAs. Since the Fence statement, as placed in CTA-scope, only synchronizes threads within a CTA, it is possible that threads in CTA 0 proceed to the multicast instruction, filling the SMEM of CTA 1 (B[1,:,:]), while threads in CTA 1 are still reading from it while executing some_in_order_op. Although the TMA instruction is async, this is not relevant to the bug illustrated here, as the TMA appears after the in-order operations upon which it races with.

The next example illustrates an intra-thread synchronization bug:

```
D: f32[2, 2, 64, 256] @ Sm90_RmemMatrixD(64, 256)  # Distributed: D[x,y,:,:] on CTA x, warpgroup y
for cta in cuda_threads(0, 2, unit=cuda_cta_in_cluster):
    for wg in cuda_threads(0, 2, unit=cuda_warpgroup):
        example_wgmma(D[cta, wg, :, :], some_A[:, :], some_B[:, :])  # D += matmul(A, B)
        some_in_order_op(D[cta, wg, :, :])
```

This code parallelizes across CTAs and warpgroups, with each warpgroup accessing only its own shard D[cta, wg, :, :]. Nevertheless, the code is flawed, as the async example_wgmma instruction isn't waited-for before some_in_order_op attempts to access its output.

A more subtle bug arises when async instructions interact with inter-thread synchronization:



```
B: f32[2, 256, 32] @ Sm90_SmemSwizzled(128)        # Distributed: B[x,:,:] on CTA x
D: f32[2, 2, 64, 256] @ Sm90_RmemMatrixD(64, 256)  # Distributed: D[x,y,:,:] on CTA x, warpgroup y
cg: barrier[2, 2] @ CudaCommitGroup                # Distributed: cg[x,y] for CTA x, warpgroup y
for cta in cuda_threads(0, 2, unit=cuda_cta_in_cluster):
    for wg in cuda_threads(0, 2, unit=cuda_warpgroup):
        some_in_order_op_1(B[cta, :, :])
        example_wgmma(D[cta, wg, :, :], some_A[:,:], B[cta, :, :])  # D += matmul(A, B)
        Arrive(wgmma_async) >> cg[cta, wg]
cs: barrier @ CudaClusterSync
Arrive(cuda_in_order) >> cs
for cta in cuda_threads(0, 2, unit=cuda_cta_in_cluster):
    for wg in cuda_threads(0, 2, unit=cuda_warpgroup):
        Await(cg[cta, wg], cuda_in_order, 0)
        some_in_order_op_2(D[cta, wg, :, :])
Await(cs, cuda_in_order, 0)
example_tma_multicast(B[:,:,:], some_gmem[:,:])  # Copy some_gmem[:,:] to B[0,:,:] and B[1,:,:]
```

The statements connected in blue on the left illustrate safe synchronization in a situation similar to our first example: the Arrive/Await pair using cs (cluster sync) ensures all accesses to B by some_in_order_op_1 finish before any thread in the cluster overwrites B using TMA. The statements connected in blue on the right illustrate safe synchronization in a situation similar to our second example: the Arrive/Await pair using cg (commit group) ensures each warpgroup waits for its example_wgmma to finish before accessing D[cta, wg...] in some_in_order_op_2.

However, the bolded access to **B** by example_wgmma is unsafe. Dotted red arrows illustrate synchronization that does *not* happen. Because the Arrive on cs uses the timeline cuda_in_order, the completion of the Arrive does not depend on the prior async wgmma instructions to retire. Meanwhile, the Await on cg (commit group) occurs too late for it to interact transitively with the prior cs (cluster sync) Arrive. Therefore, while all *non-async* accesses to B will retire before any thread proceeds to the TMA multicast, this isn't the case for the async wgmma instructions, whose B operand could be overwritten unexpectedly by another CTA multicasting to its input memory. Subtle bugs like this can be difficult to spot in optimized CUDA code.

2.3 Optimization Process and Equivalence Checking

Exo-GPU programmers can either directly write optimized object code or apply a sequence of scheduling transformations to simple initial code. Both approaches grant full control over performance-critical details. However, the scheduling approach offers significant practical advantages for performance engineering: optimization strategies become reusable functions across similar kernels, complex transformations decompose into sequences of verified primitives that enable rapid iteration with correctness guarantees, and the compiler automatically handles complex index calculations for tiling strategies—especially valuable for GPU optimization. Below, we describe Exo-GPU's scheduling-based optimization process.

v : Var	::=	$x \in \mathbb{X}$	data variable
	|	$y \in \mathbb{Y}$	control variable
	|	$z \in \mathbb{B}$	barrier variable
	|	$r \in \mathbb{W}$	warp variable
e : DExpr	::=	d	data value
	|	$x \mid x\texttt{[}w^*\texttt{]}$	reads
	|	$e_1{+}e_2 \mid e_1{-}e_2 \mid e_1{*}e_2 \mid e_1/e_2$	compound expr
e_z : ZExpr	::=	$z \mid z\texttt{[}w^*\texttt{]}$	reads
e_w : WExpr	::=	$r = (n, n, n)$	warp config
n	$\in$	$\mathbb{Z}$	integer
t	$\in$	Bool	boolean
d	$\in$	$\mathbb{D}$	data values

c : CExpr	::=	n	integer
	|	y	reads
	|	$c_1{+}c_2 \mid c_1{-}c_2 \mid c{*}n$	affine arith
	|	c / $n \mid c$ mod n	quasi affine
b : BExpr	::=	t	boolean
	|	b_1 and $b_2 \mid b_1$ or b_2	logical ops
	|	$c_1 == c_2 \mid c_1{<}c_2 \mid c_1{\leq}c_2$	relational ops
w : WinCoord	::=	c	point access
	|	$c_1..c_2$	interval access

Fig. 1. The syntax of variables and expressions of GPU IR. We use $\cdot^*$ to mean 0 or more.

The scheduling flow in Exo-GPU begins with a simple initial procedure p_1. The programmer issues a series of *rewrite operations* R_1 ... R_{N-1} to create transformed procedures $p_2 ... p_N$.

$$\begin{array}{lccccccccc} \text{Rewrites:} & p_1 & \xrightarrow{R_1} & p_2 & \dots & \xrightarrow{R_{N-1}} & p_N & \text{(Exo-GPU checks)} & p_N \\ \text{Behavior:} & \mathsf{seq}\,[\![p_1]\!] & \equiv & \mathsf{seq}\,[\![p_2]\!] & \dots & \equiv & \mathsf{seq}\,[\![p_N]\!] & \equiv & \mathsf{par}\,[\![p_N]\!] \end{array}$$

Exo's scheduling rewrite verifies functional equivalence of seq $[\![p_1]\!]$ ...seq $[\![p_N]\!]$ under sequential semantics (denoted seq $[\![...]\!]$). Exo-GPU adds new scheduling operations for parallel and synchronization constructs, but these non-semantics-altering annotations are ignored under seq $[\![...]\!]$ checks. Then Exo-GPU performs additional verification on the final program p_N. Specifically, synchronization checking ensures seq $[\![p_N]\!] \equiv$ par $[\![p_N]\!]$, where par $[\![...]\!]$ denotes parallel semantics. Transitively, this completes the *chain of equivalence* from seq $[\![p_1]\!] \equiv ... \equiv$ seq $[\![p_N]\!] \equiv$ par $[\![p_N]\!]$, guaranteeing that p_1's behavior is preserved throughout all transformations.

Equivalence checking relies on three categories of reasoning:

Exo Rewrites: Exo's existing rewrites implement dependency analysis-based equivalence checking, which guards against *dataflow logic bugs*. For example, `reorder_stmts`, which reorders two statements $s_1; s_2$, is only permitted when the effects of the statements commute.

Instruction Substitution: Explicit instruction selection rewrite (`replace`) lets programmers substitute a code block with an accelerator instruction call. Exo-GPU models instructions via their *behavior* (sequential semantics), per-parameter *memory type* (physical memory and layout), expected collective unit τ_u, and async annotations. Exo's unification verifies that the replaced code block's behavior matches the instruction, while Exo-GPU's static analysis ensures appropriate threads execute it (Section 4.1); together preventing *instruction logic errors*.

Exo-GPU Synchronization Check: Since parallel loops and synchronization statements (`Fence`, `Arrive`, and `Await`) are ignored during sequential checks, synchronization checking must ensure that their usage is valid and does not alter program behavior or introduce race conditions. We verify this via interpretation on the Abstract Machine (Section 5), which protects the program against both *inter- and intra-thread synchronization bugs* for selected concrete problem sizes.

3 GPU IR

GPU IR is the frontend language of Exo-GPU, and examples in Section 2 are written in GPU IR.

3.1 Variables and Expressions

Figure 1 defines the expression syntax of GPU IR. GPU IR explicitly distinguishes between control, data, barrier, and warp expressions and is a static control program. We denote the sets of data, control, barrier, and warp variables by $\mathbb{X}$, $\mathbb{Y}$, $\mathbb{B}$, $\mathbb{W}$, respectively. $x \in \mathbb{X}$ is a data variable, $y \in \mathbb{Y}$

is a control variable, $z \in \mathbb{B}$ is a barrier variable, and $r \in \mathbb{W}$ is a warp variable. The truth values Bool $\triangleq$ $\{\texttt{true}, \texttt{false}\}$. are ranged over by a metavariable t, and n is the metavariable for $\mathbb{Z} \triangleq \{\cdots, -1, 0, 1, 2, \cdots\}$. The metavariable $d \in \mathbb{D}$ ranges over data values, including 8-, 32-, and 64-bit integers as well as single- and double-precision floating-points.

Control expressions CExpr encompass integer, scalar read, and (quasi-) affine arithmetic operations. Boolean expressions BExpr define basic logical and relational operations on control expressions. Data expressions DExpr are similarly defined, but operations need not be affine. Barrier Expressions ZExpr are only allowed to read, but not composed. Window coordinates w^* (we use $.^*$ to denote a tuple) used in data and barrier read accesses are specified as a tuple of point-wise or interval control expressions, expressing a windowed access to multi-dimensional arrays. Scalar read is syntactic sugar for an array access of $w^* = \langle\,\rangle$ (empty tuple).

3.2 Types

All types are fully defined in Appendix A. This section provides a general introduction to the different types in GPU IR. Figure 10 defines data and barrier types. Data types τ_x specify precisions for data variables $\mathbb{X}$, such as `f32` and `i32`. Barrier types τ_z distinguish a non-explicitly guarded barrier from an explicitly guarded barrier (`barrier(z)`); the explicitly guarded barrier case is required only for mbarriers. A *collective type* ($\delta \in \Delta$) defines a tuple consisting of a domain and a box, each representing d- dimensional natural numbers $((\mathbb{N}^d, \mathbb{N}^d))$. Together, these define a d-level thread hieararchy within a thread cluster, and a selection of a number of elements on each level. An argument to `cuda_threads` loop τ_u is parameterized by a collective type δ, which is defined in Figure 11. Figures 12–14 classify three categories of memory spaces: π_d for data memories (such as `CudaRmem` for CUDA registers), π_z for barrier completion mechanisms (such as `mbarrier` or `commit_group`), and π_w for special window aliasing (such as `CUtensorMap`). Similar to τ_u, π_d is parameterized by δ.

A qualitative timeline τ_q annotates each *runtime* buffer access. We define distinct τ_q to distinguish between buffer accesses that require different synchronization mechanisms or memory fences to resolve hazards. For example, we distinguish non-async copies from `cp.async` copies (which require `cp.async.wait_all` or similar to wait for), and we distinguish register and SMEM operands of `wgmma` instructions (which require `wgmma.fence` and `fence.proxy.async`, respectively). Section 5 will formally define how a runtime buffer access instance $x[i_1, \ldots, i_n]$ tagged with q_1 states that, at the program point and thread, the access to x occurs under q_1. Moreover, the sync timelines SyncTL mentioned in Section 2.1, which parameterize synchronization statements, are actually defined as compositions of qualitative timelines (QualTL), as shown in Figure 16.

3.3 Programs

Exo-GPU supports two types of procedures, p : Proc, which models CPU functions (possibly containing CUDA kernel launches), and g : Instr, which models hardware instructions for either CPU or CUDA. Both procedure types share three common components: a statement body that defines sequential behavior, a tuple of control parameters y^*, and a tuple of data parameters x^*. Each data parameter in Proc carries two annotations: τ_x, specifying the required data precision, and π_d, specifying the required memory type for the passed argument.

Instruction parameters (τ_a) also carry both τ_x and π_d annotations, as well as hardware-specific information, namely: out-of-order, convergence, and qualitative timeline information, which all affect only abstract machine semantics for checking (Section 5). The τ_a also contains a distributed collective units tuple (τ_u^*) specifying the CUDA thread hierarchy at which each array dimension is sharded (checked by distributed memory analysis, Section 4.1). The instruction (g) contains annotations for t (a CPU/CUDA flag), a collective unit τ_u specifying the convergent threads required, and π_z and τ_u^*.

s : Stmt	::=	s_1 ; s_2	sequencing
	$\mid$	`for` y `in seq(`c_{lo}`,`c_{hi}`) do` s	sequential loop
	$\mid$	`for` y `in cuda_tasks(`c_{lo}`,`c_{hi}`) do` s	cuda tasks
	$\mid$	`for` y `in cuda_threads(`c_{lo}`,`c_{hi}`,unit=`τ_u`) do` s	cuda threads
	$\mid$	`with CudaDeviceFunction(`n, e_{w}^{*}`) do s`	CUDA device function block
	$\mid$	`with CudaWarps(`n, n, r`) do s`	Warp Specialization Block
	$\mid$	`if` b `then` s	guard
	$\mid$	$p(c^{*}, e^{*})$	non-instruction subprocedure call
	$\mid$	$g(c^{*}, e^{*})$ `>>` e_{z}	instruction subprocedure call
	$\mid$	x`[`c^{*}`] =` e	write
	$\mid$	x`[`c^{*}`] +=` e	reduce
	$\mid$	$x = x[\,w^{*}\,]\,@\,\pi_{\text{w}}$	window statement
	$\mid$	$x : \tau_x[c^{*}]\,@\,\pi_{\text{d}}$	Data alloc
	$\mid$	$z : \tau_z[c^{*}]\,@\,\pi_{\text{z}}$	Barrier alloc
	$\mid$	`free` x	Data free
	$\mid$	`free` z	Barrier free
	$\mid$	`Fence(`τ_s, τ_s`)`	Non-split barriers
	$\mid$	`Arrive(`τ_s, n`)` `>>` e_{z}^{*}	Arrive
	$\mid$	`Await(`e_{z}, τ_s, n`)`	Await
τ_a : Arg	::=	τ_x	data type (precision)
	$\mid$	π_{d}	data memory type
	$\mid$	`out_of_order` ∈ Bool	out-of-order execution flag
	$\mid$	`convergent` ∈ Bool	implicit thread sync flag
	$\mid$	τ_u^{*}	distributed collective units
	$\mid$	`initial_qual_tl` ∈ QualTL	initial qualitative timeline
	$\mid$	`ext_qual_tl` ∈ $\mathcal{P}$(QualTL)	extended qualitative timeline set
	$\mid$	`atomic_qual_tl` ∈ $\mathcal{P}$(QualTL)	atomic qualitative timeline set
p : Proc	::=	`proc` $y^{*}, (x : \tau_x\ @\ \pi_d)^{*}$ `do` s	
g : Instr	::=	`proc` $y^{*}, (x : \tau_a)^{*}, t, \tau_u, \pi_z, \tau_u^{*}$ `do` s	

Fig. 2. The syntax of GPU IR statements.

3.4 Statements

Figure 2 defines statements $s \in$ Stmt. A statement can sequence ($s_1; s_2$), iterate over loops (seq, cuda_tasks, or cuda_threads), and guard (if b then s). Data effects include indexed writes $x[c^{*}] = e$, reductions $x[c^{*}]{+}{=}\,e$, and window creation $x = x[c_{\text{lo}} : c_{\text{hi}}]\ @\ \pi_{\text{w}}$. CudaDeviceFunction and CudaWarps control generation and selection of CUDA threads. Synchronization statements Fence, Arrive, and Await get lowered to CUDA synchronization and proxy fence instructions.

CUDA Kernel Structure: All Exo-GPU code is compiled as CPU code by default. A CudaDeviceFunction block specifies a CPU-side launch of a CUDA kernel. Its body must contain only a single statement: a nest of one or more cuda_tasks loops; cuda_tasks loops must not appear elsewhere. Each iteration of the innermost cuda_tasks loop comprises a device task and is assigned to one CUDA cluster for execution. The *device scope* of a statement is CPU if outside of a CudaDeviceFunction block, and GPU if inside. At CPU scope, all loops must have loop mode seq. The arguments of CudaDeviceFunction(n, e_w^{*}) respectively define the clusterDim and, indirectly, the blockDim of the CUDA clusters launched. Each warp expression $e_w : r = (n, n, n)$ defines a group of warps named r, with the first n parameter being the number of warps. If nonzero, the latter two n parameters specify that the warps will adjust the register count down or up, respectively, using a setmaxnreg.dec or setmaxnreg.inc instruction [18]. We infer that blockDim is 32 times the total number of warps named.

```
with CudaDeviceFunction(...):
    for cta_m in cuda_tasks(...):
        for cta_n in cuda_tasks(...):
            # Device task starts here
            A_smem: f32[128, 32] @ CudaSmemLinear
            # ...
```

```
my_warp_config = [
  CudaWarpConfig("producer", 1, 40, 0),
  CudaWarpConfig("unused",   3, 40, 0),
  CudaWarpConfig("consumer", 8, 0, 232)]
with CudaDeviceFunction(clusterDim=2,
    warp_config=my_warp_config):
```

Fig. 3. Launching clusters of 2 CTAs each, with 1 "producer", 3 "unused", and 8 "consumer" warps per CTA (total `blockDim=384`). The consumer warps have 232 registers per thread, and the others only 40.

Thread Control: Having described the `cuda_tasks` loops (inter-cluster parallelism), we now describe `cuda_threads` loops and `with CudaWarps` (intra-cluster parallelism). A `cuda_threads` loop takes its the executing threads in flight and sub-divides it, assigning one subdivision to execute each loop iteration (possibly with left-over, inactivated threads). The `unit` parameter adjusts the number of threads per iteration. The `CudaWarps(`n, n, r`)` block restricts its body statement to execute only with threads residing in the warp variable named by r (which is defined in the `CudaDeviceFunction`); the numeric arguments further restrict the warps selected.

Proc & Instruction Calls: Calls to p : Proc and g : Instr are syntactically similar, distinguished only by the callable's type. Exo-GPU's static analysis enforces that calls to p : Proc only appear at CPU scope, while calls to g : Instr appear at CPU or CUDA scope as appropriate for g. For CUDA instructions, calls to g must only appear at τ_u-scope, where τ_u is the collective unit specified for g. Additionally, each data parameter must have the correct precision τ_x and memory type π_d, and the trailing barrier (if required) must have the correct π_z.

Synchronization Statements: The user specifies synchronization between threads and/or between async accelerator instructions by inserting synchronization statements. They take *synchronization timeline* (SyncTL) parameters, defined as compositions of qualitative timelines and a transitivity (trnstv?) flag (Figure 16). The transitivity flag controls interaction between synchronization statements, detailed in Section 5.

Each τ_s : SyncTL contains two QualTL sets: the full timeline set (indicated by "full" in (Figure 16), and a temporal timeline set (indicated by "full" or "temp."). A `Fence(`$\tau_s^{\text{pre}}, \tau_s^{\text{post}}$`)`, or paired `Arrive(`$\tau_s^{\text{pre}}$`,_)` and `Await(_,` τ_s^{post}`,_)` statements, resolves hazards between two accesses to the same array element when: (1) the first access precedes synchronization and uses a QualTL from τ_s^{pre}'s full timeline set, and (2) the second access follows synchronization and uses a QualTL from τ_s^{post}'s full timeline set (for reads) or temporal timeline set (for writes). In hardware terms, the distinction between the full and temporal timeline sets is due to memory fences. If the prior value of a buffer element is *overwritten*, without being read, we require only that the overwrite is temporally ordered after prior accesses to that buffer element, and memory fences may be elided.

Alloc & Free: An allocation statement begins the lifetime of a new data or barrier variable. For data allocations, the data memory π_d determines the physical backing memory (registers (RMEM), shared memory (SMEM), or global memory (GMEM)) and the mapping between multidimensional tensor coordinates and 1D memory offsets (Figure 12). It also carries an implicit collective type δ, specifying which level of the thread hierarchy at which the physical memory type is allocated (e.g. CTA for shared memory allocations); this is consumed by distributed memory analysis (Section 4.1). For barrier allocations, the BarrierComp parameter π_z controls the completion mechanism used for CUDA code translated from `Arrive` and `Await` statements (Figure 13). The explicitly-guarded barrier type is relevant only for certain configurations of `mbarrier` usage. A free statement ends the lifetime of a data or barrier variable. Currently, we forbid the user from inserting free statements manually; these are added automatically in a separate compiler pass.

Window Statement: A window statement x_{win} `=` x_{data}`[`w^*`] @` π_w defines an alias to x_{data}. A window into x_{win} can be passed as an instruction argument which requires special window type π_w. Currently, we only use this feature to construct `CUtensorMap` objects [18] required for TMA instructions; we parameterize the `CUtensorMap` type with its swizzle mode and box shape.

4 COLLECTIVE ANALYSIS

Collective analysis is a static, forward dataflow analysis over a GPU IR procedure. Its results are consumed by (1) a memory analysis, (2) code generation, and (3) an abstract-machine interpreter. The analysis annotates every GPU lexical scope with a collective tiling $\omega \in \Omega$. By default, statements inherit their parent's tiling unchanged; a new tiling is derived only at: device entry (`CudaDeviceFunction`), thread loops (`cuda_threads`), and warp-specialization blocks (`CudaWarps`). The purpose of this analysis is to statically reason about the mapping between work and the threads *within* clusters, so here we ignore `cuda_tasks` loops, which distribute work across clusters. We uniquely identify a thread within a cluster by its *natural thread index*; a set of such indices comprises a *thread collective* $\mu \in \mathbb{T}$, where $\mathbb{T} \triangleq \mathcal{P}(\mathbb{N})$.

Definition 4.1 (Natural thread index)**.** The *natural thread index* for a CUDA thread is `cluster_ctarank * blockDim.x + threadIdx.x`. (Exo-GPU only parallelizes on the x dimension).

Definition 4.2 (Collective tiling)**.** A *collective tiling* $\omega \in \Omega$ for an M-dimensional domain is an M-tuple of *dimension descriptors*

$$\omega = \langle \mathcal{D}_0, \ldots, \mathcal{D}_{M-1} \rangle \qquad \text{with} \qquad \mathcal{D}_m = (D_m, \mathcal{O}_m),$$

where $D_m \in \mathbb{N}$ is the extent of dimension m and $\mathcal{O}_m \in \mathcal{O}^*$ is an ordered list of (possibly empty) *dimension operators* $\mathcal{O} : \mathbb{Y} \times \mathbb{N}^3$. We write the thread pitch of dimension m as $P_m \triangleq \prod_{i=m+1}^{M-1} D_i$ where $P_{M-1} = 1$.

The dimensions of a collective tiling ω capture how a cluster's threads are partitioned into hierarchical collectives (cluster, block, warp). We support *reshape* to introduce ad-hoc additional hierarchy (e.g. pairs of warps). From there, each dimension's dimension operators summarize the control variables (identified by $y \in \mathbb{Y}$) that iterate on that dimension. The thread pitch of the dimension describes the distance, in units of natural thread indices, between adjacent elements (e.g. a CTA dimension would have thread pitch `blockDim`).

During forward dataflow analysis, new collective tilings are derived for the three special statements mentioned above, while all other statements inherit their parent's tiling. It produces a child collective tiling through the following transformation:

$$\mathsf{derive}_\omega : \ (\omega^{\mathrm{env}}, \delta^{\mathrm{stmt}}, y, \mathsf{lo}, \mathsf{hi}, \mathsf{tileCount}) \ \to \ \omega'$$

where ω^{env} is the parent scope's collective tiling and δ^{stmt} is the collective type from the statement. The (optional) iterator y and bounds $[\mathsf{lo}, \mathsf{hi})$ describe a regular slice of linear thread space; $\mathsf{tileCount} \in \mathbb{N}$ is the number of tiles for that slice.

The `cuda_threads` loop and `CudaWarps` block use derive_ω to create a new collective tiling ω based on the one annotating the parent scope. The `CudaDeviceFunction` block derives ω based on launch parameters: ω is 1-D if `clusterDim`= 1 (domain D_0=`blockDim`), otherwise 2-D with (D_0, D_1)=(`clusterDim`, `blockDim`); all operator lists start empty.

The implementation of derive_ω proceeds in two steps:

(1) *Shape alignment.* Make ω^{env} and δ^{stmt} compatible by reshaping via hierarchical splits so that their thread pitches P_m align.
(2) *Single-dimension refinement.* After alignment, at most one dimension m is newly tiled; the derivation appends exactly one operator to $\mathcal{O}_m$.

Once a tiling ω is derived, it is compiled into a *thread mapping*: $f_\omega : \Sigma \to \mathbb{T}$, where $\Sigma \triangleq (\mathbb{Y} \to \mathbb{Z})$ which maps a control environment $\sigma \in \Sigma$ to a runtime thread collective $\mu \in \mathbb{T}$. Equivalently, let $\mathcal{F} \triangleq \{ f \mid f : \Sigma \to \mathbb{T} \}$ and $f_\omega \in \mathcal{F}$. During conversion, each Abstract Machine statement $s^\#$ is annotated with its thread mapping $f_{s^\#} \in \mathcal{F}$. In Section 5.3 we write $f_{s^\#}$ for the thread mapping associated with statement $s^\#$.

4.1 Distributed Memory Analysis

Distributed memory analysis is a static, program-wide consistency check that ensures every use of CUDA-scoped data and barrier variables is compatible with their memory definitions. Each memory type π_d specifies a collective type δ_{π_d}. The analysis validates that allocation and use agree on which threads "own" which memory slices, indexing respects that ownership, and all uses induce the same mapping from logical indices to CUDA thread identifiers. Analysis runs in four steps:

(1) **Desugaring instruction call.** To expose the implicit collective requirement encoded by each hardware instruction to the collective analysis in **step 2**, we rewrite instruction calls $g(\cdots)$ into explicit loops. Each arguments to the instruction call are wrapped around with `cuda_threads`, with their collective units specified by $g.\tau_u^*$ in the instruction specification.

(2) **Collective analysis.** Run collective analysis and annotate each scope with $\omega \in \Omega$.

(3) **Triple collection.** For each data or barrier variable, record the *access triple* of all of use sites. We use $\mathcal{T}_v$ to denote a set of those triples for a variable v; these sets for all the data and barrier variables are the sole inputs to **step 4**. Each triple (n^*, ω, c^*) denotes: the subdivided dimensions n^*, the use-site tiling ω, and observed index coordinates c^*. For each use site:
   - **Data.** Compute n^* from the memory's collective type δ_{π_d} and the use-site tiling ω.
   - **Barriers.** Treat all dimensions as subdivided (every axis participates in distribution).

(4) **Triple analysis.** For each CUDA-scoped data variable v, we require: (i) consistency of the allocation w.r.t. the memory's δ_{π_d}, (ii) consistency of data or barrier accesses across sites (by checking all triples in $\mathcal{T}_v$), and (iii) all triples in $\mathcal{T}_v$ must derive one and only one thread pitch tuple that summarizes how incrementing logical indices advances natural thread IDs.

As a result, distributed memory analysis rejects programs that (i) mismatch allocation and accesses with their memory declaration, (ii) inconsistently index the same data or barrier across sites, or (iii) rely on missing/extra CUDA-thread iterators. Passing guarantees that the program's memory behavior is coherent with its collective structure and declared memory types.

4.2 Code Generation to CUDA Code

Lowering `cuda_threads` Loops: The correspondence between parallel loops and threads has thus far been expressed in terms of thread mappings of type $\Sigma \rightarrow \mathbb{T}$, which infer a set of active threads from the control environment σ. However, the generated CUDA C++ implements the inverse of this mapping, inferring the value of a control variable y from the thread index instead. Each Exo-GPU loop of the form `for y in cuda_threads(lo, hi, unit=`τ_u`) do s` lowers to CUDA C++ as shown in the inset right. Here, f is a function of CUDA's built-in `blockIdx` and `threadIdx` variables, and g is a boolean condition on y that disables threads not assigned to execute any iteration of the body. Where possible, f is 0 and g is 1, to facilitate uniform branching.

```
if (int y = f(blockIdx, threadIdx); g(y)) {
    lower(s);}
```

Warp Specialization: When a device function uses multiple warp variables $r_1, ..., r_W \in \mathbb{W}$, we generate W-many CUDA C++ code paths from a single Exo-GPU device function, each specialized for (and executed by) the threads assigned to one warp variable. Each path uses PTX's `setmaxnreg` instruction to vary per-thread register counts, and omits all code paths guarded by a `with CudaWarps` block holding code not intended for the current warp. This allows the PTX assembler to statically verify that register-heavy instructions (e.g. `wgmma`) don't appear on low-register code paths.

Persistent Kernels and `cuda_tasks` Loops: The generated CUDA C++ device functions implement a persistent kernel design that launches exactly the number of thread block clusters needed to saturate the device. Following the approach of CUTLASS grouped kernel schedulers [16], we implement a C++ task generator object that generates a series of device task coordinates for thread block clusters to execute. The current implementation orders tasks lexicographically and assigns them round-robin to thread block clusters for execution. While we leave it as future work to

$\tau_v : \mathrm{VL}$	::=	$\texttt{fully\_ordered} \mid \texttt{temporally\_ordered}$ $\mid \texttt{unordered} \mid \texttt{atomic\_only} \mid \texttt{invisible}$	visibility levels
$e^{\#} : \mathrm{Expr}^{\#}$	::=	$e^{\#}_{d} \mid e^{\#}_{z}$ where $e^{\#}_{d} \triangleq x[\,w^*\,]$ $e^{\#}_{z} \triangleq z[\,w^*\,]$	barrier and data accesses
$s^{\#} : \mathrm{Stmt}^{\#}$	::=	$s^{\#}_1 ; s^{\#}_2$	sequencing
		$\mid$ $\texttt{for}\ y\ \texttt{in loop}(c_{\mathrm{lo}}, c_{\mathrm{hi}})\ \texttt{do}\ s^{\#}$	loop
		$\mid$ $\texttt{if}\ b\ \texttt{then}\ s^{\#}$	guard
		$\mid$ IncTaskID	increment the task ID
		$\mid$ $\texttt{Alloc}(e^{\#})$	Updates SyncEnv for Alloc
		$\mid$ $\texttt{RecordRead}(\tau_v, t, e^{\#}, \tau_q, \tau^*_q, e^{\#}_{\mathrm{z}})$	Updates SyncEnv with Read
		$\mid$ $\texttt{RecordMutate}(\tau_v, t, e^{\#}, \tau_q, \tau^*_q, e^{\#}_{\mathrm{z}})$	Updates SyncEnv with Mutate
		$\mid$ $\texttt{ClearReads}(e^{\#})$	Remove state added by RecordRead
		$\mid$ $\texttt{ClearMutates}(e^{\#})$	Remove state added by RecordMutate
		$\mid$ $\texttt{Fence}(t, \tau^*_q, \tau^*_q, \tau^*_q)$	Non-split barriers
		$\mid$ $\texttt{Arrive}(t, \tau^*_q, e^{\#*}_{\mathrm{z}})$	Arrive
		$\mid$ $\texttt{Await}(e^{\#}_{\mathrm{z}}, \tau^*_q, \tau^*_q, n)$	Await
		$\mid$ $\texttt{CheckReads}(\tau_v, t, e^{\#}, \tau^*_q)$	Checks visibility of Reads in SyncEnv
		$\mid$ $\texttt{CheckMutates}(\tau_v, t, e^{\#}, \tau^*_q)$	Checks visibility of Mutates in SyncEnv
		$\mid$ $\texttt{CheckBarrier}(e^{\#}_{\mathrm{z}})$	Checks the consistency of arrive and await
$p^{\#} : \mathrm{Proc}^{\#}$	::=	proc y^* do $s^{\#}$	arguments are control-only

Fig. 4. Syntax of Abstract Machine IR (AMIR). Variables, types, and expressions (c, b, w) use the same definition as GPU IR. Visibility levels are denoted by the *type* $\tau_v \in \mathrm{VL}$ and range over the five constructors listed above.

provide a mechanism for programmers to customize this ordering, such customization would be unconstrained by frontend's quasi-affine indexing restrictions since cuda_tasks loops impose no semantic ordering, thereby enabling techniques like Morton swizzling.

Shared Memory Allocations: Device functions generated by Exo-GPU use only dynamic shared memory for SMEM allocation. The Exo-GPU compiler deduces the required SMEM allocation size and statically assigns offsets into this allocation for each SMEM-backed variable. Variables with non-overlapping lifetimes may share the same memory locations; to ensure safe aliasing, we prohibit any thread in the cluster from passing an SMEM free statement until all reads and writes to the freed variable have retired, as enforced by the Abstract Machine (Section 5). This is necessary because freed SMEM may be immediately reused by any thread in the cluster. We note that the compiler's aliasing may be suboptimal, and the cluster-wide safety requirement may be overly conservative (e.g., for CTA-local variables). Providing programmers explicit control over SMEM lifetime and aliasing is left as future work.

5 ABSTRACT MACHINE

To guarantee sequential–parallel equivalence, Exo-GPU performs synchronization checking via Abstract Machine interpretation. This section introduces the AMIR grammar (Section 5.1), the translation from GPU IR (Section 5.2), and the AMIR execution semantics (Section 5.3). We translate GPU IR into AMIR solely for validation: the frontend language remains simple, while synchronization reasoning is factored into explicit AMIR operations that are orthogonal to value computation.

5.1 Abstract Machine Grammar

Figure 4 defines AMIR grammar. It follows GPU IR for control c, booleans b, window coordinates w, and all variable/type definitions (Figures 1, 2). AMIR defines only windowed access expressions, written $e^{\#}_{d} \triangleq x[w^*]$ (data access) and $e^{\#}_{z} \triangleq z[w^*]$ (barrier access). AMIR introduces visibility levels $\tau_v \in \mathrm{VL}$, which are totally ordered: invisible ≺ atomic_only ≺ unordered ≺ temporally_ordered ≺ fully_ordered.

GPU IR	**Abstract Machine IR**
$s_1; s_2$	$T_s(s_1);\ T_s(s_2)$
$\texttt{for}\ y\ \texttt{in seq}(c_{\text{lo}}, c_{\text{hi}})\ \texttt{do}\ s$	$\texttt{for}\ y\ \texttt{in loop}(c_{\text{lo}}, c_{\text{hi}})\ \texttt{do}\ T_s(s)$
$\texttt{for}\ y\ \texttt{in cuda\_tasks}(c_{\text{lo}}, c_{\text{hi}})\ \texttt{do}\ s$	$\texttt{for}\ y\ \texttt{in loop}(c_{\text{lo}}, c_{\text{hi}})\ \texttt{do (IncTaskID;}\ T_s(s))$
$\texttt{for}\ y\ \texttt{in cuda\_threads}(c_{\text{lo}}, c_{\text{hi}}, \tau_u)\ \texttt{do}\ s$	$\texttt{for}\ y\ \texttt{in loop}(c_{\text{lo}}, c_{\text{hi}})\ \texttt{do}\ T_s(s)$
$x[c^*]\ \diamond\ e\quad \textit{where}\ \diamond \in \{=, +=\}$	$\fatsemi_{e_1^\# \in T_{\text{ex}}(e)}\ \big(\texttt{CheckMutates}(\tau_v^{\text{pre}}, \texttt{false}, e_1^\#, \{T_{\text{qin}}(e_1^\#)\})\texttt{)};$ $\texttt{RecordRead}(\tau_v^{\text{post}}, \texttt{false}, e_1^\#, T_{\text{qin}}(e_1^\#), \emptyset, \emptyset)\big)$ $\fatsemi_{e_2^\# \in T_{\text{ex}}(x[c^*])}\ \big(\texttt{CheckReads}(\tau_v^{\text{pre}}, \texttt{false}, e_2^\#, \{T_{\text{qin}}(e_2^\#)\})\texttt{)};$ $\texttt{CheckMutates}(\tau_v^{\text{pre}}, \texttt{false}, e_2^\#, \{T_{\text{qin}}(e_2^\#)\})\texttt{)};$ $\texttt{ClearReads}(e_2^\#);$ $\texttt{ClearMutates}(e_2^\#);$ $\texttt{RecordMutate}(\tau_v^{\text{post}}, \texttt{false}, e_2^\#, T_{\text{qin}}(e_2^\#), \emptyset, \emptyset)\texttt{)};\big)$, where $\tau_v^{\text{post}} = \texttt{fully-ordered and}$ $\tau_v^{\text{pre}} = \texttt{fully-ordered}$ if +=, else $\texttt{temporally-ordered}$
$\texttt{if}\ b\ \texttt{then}\ s$	$\texttt{if}\ b\ \texttt{then}\ T_s(s)$
$\texttt{with CudaDeviceFunction}(n,\ e_{\text{w}}^*)\ \texttt{do}\ s$	$\texttt{Fence}(\text{true}, \text{Qcs} \cup \{\texttt{cpu\_in\_order\_qual}\}, \text{Qcs}, \text{Tcs});$ $T_s(s);$ $\texttt{Fence}(\text{true}, \text{Qcs}, \text{Qcs}, \text{Tcs});$, where $\text{Qcs} \triangleq \text{T}_{\text{qfull}}(\texttt{cuda\_stream\_sync})$, $\text{Tcs} \triangleq \text{T}_{\text{qtmp}}(\texttt{cuda\_stream\_sync})$
$\texttt{with CudaWarps}(n,\ n,\ r)\ \texttt{do}\ s$	$\texttt{if true then}\ T_s(s)$
$g(c^*, e^*) \gg e_{\text{z}}$	$\fatsemi_{e^\# \in T_{\text{ex}}(e_z)}\ \texttt{RecordRead}(\texttt{fully-ordered}, \texttt{false}, e^\#, \emptyset, \emptyset, T_e(e_{\text{z}}))$ $\fatsemi_{i=0}^{\text{len}(e^*)}\ T_{\text{arg}}(g.p_i,\ e_i,\ e_{\text{z}})$
$x : \tau_x[c^*]\ @\ \pi_{\text{d}}$	$\texttt{Alloc}(x[c^*])$
$z : \tau_z[c^*]\ @\ \pi_{\text{z}}$	$\texttt{Alloc}(z[c^*])$
$\texttt{free}\ x$	$\fatsemi_{e^\# \in T_{\text{smem}}(x[:^*])}\ \big(\texttt{CheckReads}(\tau_v^{\text{free}}, \texttt{false}, e^\#, \{T_{\text{qin}}(e^\#)\})\texttt{)};$ $\texttt{CheckMutates}(\tau_v^{\text{free}}, \texttt{false}, e^\#, \{T_{\text{qin}}(e^\#)\})\texttt{)};\big)$
$\texttt{free}\ z$	$\texttt{CheckBarriers}(\ T_e(z[:^*])\ );$ $\fatsemi_{e^\# \in T_{\text{smem}}(z[:^*])}\ \big(\texttt{CheckReads}(\tau_v^{\text{free}}, \texttt{false}, e^\#, \{T_{\text{qin}}(e^\#)\})\texttt{)};$ $\texttt{CheckMutates}(\tau_v^{\text{free}}, \texttt{false}, e^\#, \{T_{\text{qin}}(e^\#)\})\texttt{)};\big)$
$\texttt{Fence}(\tau_s^{\text{pre}}, \tau_s^{\text{post}})$	$\texttt{Fence}(\text{T}_{\text{tr}}(\tau_s^{\text{pre}}),\ \text{T}_{\text{qfull}}(\tau_s^{\text{pre}}),\ \text{T}_{\text{qfull}}(\tau_s^{\text{post}}),\ \text{T}_{\text{qtmp}}(\tau_s^{\text{post}}))$
$\texttt{Arrive}(\tau_s, n) \gg e_{\text{z}}^*$	$\fatsemi_{e^\# \in T_{\text{ex}}(e_{\text{z}}^*)}\ \texttt{RecordRead}(\texttt{fully-ordered}, \texttt{false}, e^\#, \text{T}_{\text{qarr}}(\tau_s), \emptyset, \emptyset)$ $\texttt{Arrive}(\text{T}_{\text{tr}}(\tau_s),\ \text{T}_{\text{qfull}}(\tau_s),\ T_e(e_{\text{z}}^*));$
$\texttt{Await}(e_{\text{z}}, \tau_s, n)$	$\fatsemi_{e^\# \in T_{\text{ex}}(e_z)}\ \texttt{RecordRead}(\texttt{fully-ordered}, \texttt{false}, e^\#, T_{\text{qin}}(e^\#), \emptyset, \emptyset)$ $\texttt{Await}(\text{T}_{\text{e}}(e_{\text{z}}),\ \text{T}_{\text{qfull}}(\tau_s),\ \text{T}_{\text{qtmp}}(\tau_s),\ n);$

Fig. 5. T_s : Stmt → Stmt$^\#$ (statement conversion). All *conversion* functions use the $T\_\cdot$ prefix (Figures 18-20). $\tau_v^{\text{free}} = \texttt{temporally-ordered}$

AMIR has no hardware-instruction procedures (g in GPU IR) and no subprocedure calls. Procedures $p^\#$ take control-only arguments. The statement (Figure 4) consists of: (i) structural statements–sequencing, loop, and guard (ii) recording statements that update the synchronization environment SyncEnv–`Alloc`, `RecordRead`, `RecordMutate`, `Fence`, `Arrive`, and `Await`; and (iii) checking statements that query SyncEnv and fail if the required visibility obligations are not met; otherwise they are no-ops: `CheckBarrier`, `CheckReads`, and `CheckMutates`.

5.2 Conversion from GPU IR to Abstract Machine IR

We describe how GPU IR expressions (Section 5.2.1), statements (Section 5.2.2), and instruction arguments (Section 5.2.3) are converted. Before conversion, we apply two trivial preprocessing steps to GPU IR: (i) Inline all window statements, and (ii) Inline all subprocedure calls that occur outside any `CudaDeviceFunction` block. Throughout this section (and in figures) we use the following notation. For a finite, ordered variables $R = \langle r_0, \dots, r_{k-1} \rangle$ and a statement template $S(\cdot)$, $;_{i=0}^{k} S(x) \quad \triangleq \quad S(r_0) \ ; \ S(r_1) \ ; \ \cdots \ ; \ S(r_{k-1})$. If $R = \emptyset$, the result is no-op. The intent is purely left-to-right sequential composition at AMIR conversion time. When unambiguous, we write $f(A) = \{ f(a) \mid a \in A \}$ for the pointwise image of a set.

5.2.1 Expression Conversion. Figure 17 defines $T_e : \text{DExpr} \cup \text{ZExpr} \longrightarrow \mathcal{P}(\text{Expr}^\#)$, which extracts the windowed AMIR accesses $e^\# \in \{x[w^*], z[w^*]\}$: scalars d contribute nothing ($\emptyset$); a bare name x or z yields the degenerate accesses $x[\langle\rangle]$ or $z[\langle\rangle]$; explicit indexings $x[w^*]$ and $z[w^*]$ are preserved; and for arithmetic e_1 op e_2 the sets union, $T_e(e_1) \cup T_e(e_2)$. T_e is set-valued (order and duplicates are irrelevant), leaves the value expression unchanged, and merely supplies the converted $\text{Expr}^\#$ used by `RecordRead`/`RecordMutate` and subsequent visibility checks.

5.2.2 Statement Conversion. Figure 5 gives a structural translation $T_s : \text{Stmt} \longrightarrow \text{Stmt}^\#$ from GPU IR to AMIR. Helper functions referenced by Figure 5 are defined in Figure 18. Sequencing and conditionals are preserved, while all loops are normalized to the abstract `loop` form. Warp/block scopes are dropped and replaced with a guard whose condition is `true`. Assignments $x[c^*] \diamond e$ ($\diamond \in \{=, +=\}$) are lowered to explicit read/write effects. We expand source and destination with T_e and remove sync-exempt expressions via T_{ex}. *Sync-exempt memories* are those whose accesses can be assumed safe without checking: `CudaGridConstant`, `CudaClusterSync`, and `CudaCommitGroup`. *Shared memories* are the following: `CudaBasicSmem`, `CudaSmemAtomicity16B`, `CudaSmemLinear`, `Sm90_SmemSwizzled(B)`, or `CudaMbarrier`.

Hardware instruction calls ignore control arguments and pair the instruction's parameter definition and the arguments via T_{arg}. The entry and exit of `CudaDeviceFunction` is bracketed with fences with `cuda_stream_sync`, capturing both the CPU-to-GPU fence implied by the kernel launch, and the serialization of the CUDA kernel with respect to prior and subsequent CUDA kernels and API calls (Note that Exo-GPU currently uses only one CUDA stream). During the conversion, we record the scope of each statement using a global environment $\text{Scope} : \text{Stmt}^\# \longrightarrow \{\text{CPU}, \text{CUDA}\}$. Statements outside the `CudaDeviceFunction` block are CPU-scoped, while those inside are GPU-scoped. The scope mapping is used in Definition 5.2. `Fence`, `Arrive`, and `Await` translate a sync timeline (τ_s) into a qualitative timeline set (τ_q^*). T_{tr} queries whether a sync timeline is transitive; T_{qfull} and T_{qtmp} materialize the full and temporal QualTL sets, respectively. `Arrive` selects its arrival qualitative timeline via T_{qarr} to sync-check the barrier itself, thereby preventing use-after-free bugs.

5.2.3 Instruction Argument Conversion. Figure 19 expands the conversion $T_{\text{arg}}(g.p_i, e_i, e_z)$ from a GPU IR instruction call argument to the AMIR statements. The auxiliary projections and predicates referenced in the figure—T_{ext}, T_{atom}, T_{init}, T_{ooo}, and T_{cvg}—are defined in Figure 20. For each non-sync exempt expression $e^\# \in T_{\text{ex}}(e_i)$, the conversion emits a sequence of checks and effect-recording operations which depends on two properties of the parameter $g.p_i$: its *access mode* (read-only vs. write-only/read–write) and whether it is *atomic*. The access mode of $g.p_i$ is determined by a simple intraprocedural static analysis over instruction body (behavior) $g.s$: if $g.p_i$ is only read (and never written or reduced), it is classified as read-only; if it is written but never read or reduced, it is write-only; otherwise it is read–write. The atomic case is selected exactly when $T_{\text{atom}}(g.p_i) \neq \emptyset$.

5.2.4 Program Conversion. The GPU IR 's program `proc` $y^*, (x : \tau_x)^*$; `do` s is translated into an AMIR program `proc` y^* ; `do` $s^\#$ where only control arguments are retained. Non-control arguments

$$\frac{}{\langle n, \sigma\rangle \Downarrow_{\mathbb{Z}} n}\ \textsc{C-Int} \qquad \frac{\sigma(y) = n}{\langle y, \sigma\rangle \Downarrow_{\mathbb{Z}} n}\ \textsc{C-Read} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow_{\mathbb{Z}} n_1 \qquad \langle c_2, \sigma\rangle \Downarrow_{\mathbb{Z}} n_2 \qquad \mathrm{op} \in \{+, -, *, /\}}{\langle c_1 \,\mathrm{op}\, c_2, \sigma\rangle \Downarrow_{\mathbb{Z}} (n_1 \,\mathrm{op}\, n_2)}\ \textsc{C-Aff}$$

$$\frac{}{\langle t, \sigma\rangle \Downarrow_{\mathsf{Bool}} t}\ \textsc{B-Const} \qquad \frac{\langle b_1, \sigma\rangle \Downarrow_{\mathsf{Bool}} t_1 \qquad \langle b_2, \sigma\rangle \Downarrow_{\mathsf{Bool}} t_2 \qquad \mathrm{op} \in \{\mathsf{and}, \mathsf{or}\}}{\langle b_1 \,\mathrm{op}\, b_2, \sigma\rangle \Downarrow_{\mathsf{Bool}} \widehat{\mathrm{op}}(t_1, t_2)}\ \textsc{B-Log}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow_{\mathbb{Z}} n_1 \qquad \langle c_2, \sigma\rangle \Downarrow_{\mathbb{Z}} n_2 \qquad \mathrm{op} \in \{==, <, \leq\}}{\langle c_1 \,\mathrm{op}\, c_2, \sigma\rangle \Downarrow_{\mathsf{Bool}} (n_1 \,\mathrm{op}\, n_2)}\ \textsc{B-Rel} \qquad \frac{\langle c, \sigma\rangle \Downarrow_{\mathbb{Z}} i}{\langle c, \sigma\rangle \Downarrow_{\mathcal{I}} \{\, i \,\}}\ \textsc{W-Point}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow_{\mathbb{Z}} i_1 \qquad \langle c_2, \sigma\rangle \Downarrow_{\mathbb{Z}} i_2}{\langle c_1..c_2, \sigma\rangle \Downarrow_{\mathcal{I}} \{\, i \in \mathbb{Z} \mid i_1 \leq i \leq i_2 \,\}}\ \textsc{W-Range} \qquad \frac{\forall j \in \{1..n\}.\ \langle w_j, \sigma\rangle \Downarrow_{\mathcal{I}} W_j}{\langle w_1, \ldots, w_n, \sigma\rangle \Downarrow_{\mathcal{I}^n} W_1 \times \cdots \times W_n}\ \textsc{W-Tuple}$$

$$\frac{a \in \mathbb{X} \cup \mathbb{B} \qquad \langle w_1, \ldots, w_n, \sigma\rangle \Downarrow_{\mathcal{I}^n} \mathbf{W}}{\langle a[w_1, \ldots, w_n], \sigma\rangle \Downarrow_{\mathsf{Acc}} \{\, \{a \mapsto \vec{i}\} \mid \vec{i} \in \mathbf{W} \,\}}\ \textsc{E\#-Acc}$$

Fig. 6. The big-step operational semantics for expressions.

$(x : \tau_x)^*$ are ignored during the conversion. Control variables are preserved, since GPU IR and AMIR share the same control variables set $\mathbb{Y}$. T_s is applied to the program body to convert s to $s^\#$.

5.3 Abstract Machine Semantics

Definition 5.1 (Control–Value Environment)**.** Let $\mathbb{Z}$ be the set of control value and $\mathbb{Y}$ the set of control variables. $\sigma \in \Sigma \triangleq \mathbb{Y} \to \mathbb{Z}$.

Definition 5.2 (Global Threads Collective)**.** Let $\mathbb{I}$ be the set of task IDs and $\mathbb{T}$ the set of all (local) thread IDs (thread collective). The set of global thread IDs is $\mathbb{G} \triangleq \mathbb{I} \times \mathbb{T}$. For each statement $s^\#$, thread mapping $f_{s^\#} : \Sigma \to \mathbb{T}$ returns the set of local threads that execute $s^\#$ under σ ($f_{s^\#}$ defined in Section 4). The scope mapping scope : Stmt$^\#$ $\to$ {CPU, GPU} is initialized during the statement conversion to indicate whether each statement belongs to the CPU or GPU scope. Given σ and $\imath \in \mathbb{I}$, define the corresponding global threads

$$g_{s^\#}(\sigma, \imath) \ \triangleq\ \begin{cases} \{\, (\imath, t) \mid t \in f_{s^\#}(\sigma) \,\} & \text{if } \mathrm{scope}(s) = \mathrm{GPU} \\ \mathbb{G} & \text{otherwise} \end{cases}$$

Definition 5.3 (Synchronization Environment)**.** Let $\mathbb{B}$ be the set of barrier names and $\mathbb{X}$ the set of data variables. Define

$$\mathbb{P} : \mathbb{B} \to \mathbb{Z}^n \to \mathcal{P}(\mathbb{N}) \quad \text{and} \quad \mathbb{S} : \mathbb{G} \to \mathrm{QualTL} \to \mathrm{VL},$$

where $\mathbb{P}(b, c^*)$ is the set of observed arrive counts at barrier access $b[c^*]$, and $\mathbb{S}(g, \tau_q)$ is the visibility level for the thread with global ID $g \in \mathbb{G}$ on a qualitative timeline τ_q (Figure 4). Let

$$\mathbb{R} \ \triangleq\ \mathrm{QualTL} \times \mathbb{S} \times \mathbb{P}$$

be the set of *visibility records*, written (q, ʃ, p). The synchronization environment is the map

$$\rho :\ (\mathbb{X} \cup \mathbb{B}) \to \mathbb{Z}^n \ \to\ \mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathbb{R}) \times \mathbb{N}^2,$$

sending a name and index to a tuple of (read visibility record (r): $\mathbb{R}$, mutate visibility record (m): $\mathbb{R}$, barrier count (b): n^2).

The semantics follows a standard big-step style where judgments take the form $\langle e^\#, \sigma\rangle \Downarrow v$ for expressions and $\langle s^\#, \imath, \sigma, \rho\rangle \Downarrow \imath', \sigma', \rho'$ for statements, indicating that expression $e^\#$ evaluates to value v in state σ, and statement $s^\#$ transforms state σ to σ', ρ to ρ', and $\imath$ to $\imath'$, respectively.

$$\frac{\langle e^{\#}, \sigma\rangle \Downarrow_{\text{Acc}} W_{e^{\#}} \qquad \begin{array}{l} \exists(\_, \int, \_) \in \rho(W_{e^{\#}}).r\ \big((t = \text{false} \wedge \exists g \in g_{s^{\#}}(\sigma, \imath), \forall \tau_q \in \tau_q^{*} \mid \int(g, \tau_q) \not\succeq \tau_v) \\ \qquad\qquad\qquad\qquad \vee\ (t = \text{true} \wedge \forall g \in g_{s^{\#}}(\sigma, \imath), \forall \tau_q \in \tau_q^{*} \mid \int(g, \tau_q) \not\succeq \tau_v)\big) \Downarrow_{\text{Bool}} \text{true} \end{array}}{\langle \text{CheckReads}(\tau_v, t, e^{\#}, \tau_q^{*}), \imath, \sigma, \rho\rangle \Downarrow \text{error}}\ \textsc{CheckReads}$$

$$\frac{\langle e^{\#}, \sigma\rangle \Downarrow_{\text{Acc}} W_{e^{\#}} \qquad \begin{array}{l} \exists(\_, \int, \_) \in \rho(W_{e^{\#}}).m\ \big((t = \text{false} \wedge \exists g \in g_{s^{\#}}(\sigma, \imath), \forall \tau_q \in \tau_q^{*} \mid \int(g, \tau_q) \not\succeq \tau_v) \\ \qquad\qquad\qquad\qquad \vee\ (t = \text{true} \wedge \forall g \in g_{s^{\#}}(\sigma, \imath), \forall \tau_q \in \tau_q^{*} \mid \int(g, \tau_q) \not\succeq \tau_v)\big) \Downarrow_{\text{Bool}} \text{true} \end{array}}{\langle \text{CheckMutates}(\tau_v, t, e^{\#}, \tau_q^{*}), \imath, \sigma, \rho\rangle \Downarrow \text{error}}\ \textsc{CheckMutates}$$

$$\frac{\langle e_z^{\#}, \sigma\rangle \Downarrow_{\text{Acc}} W_{e_z^{\#}} \qquad (\exists(a, w) \in \rho(W_{e_z^{\#}}).b \mid a \neq w) \Downarrow_{\text{Bool}} \text{true}}{\langle \text{CheckBarrier}(e_z^{\#}),\ \imath,\ \sigma,\ \rho\rangle \Downarrow \text{error}}\ \textsc{CheckBarrier}$$

Fig. 7. Big-step rules for access visibility and barrier consistency. Successful checks leave $(\imath, \sigma, \rho)$ unchanged; any violation reduces the configuration to error state.

5.3.1 Expression semantics. Figure 6 gives big-step judgements $\langle e, \sigma\rangle \Downarrow_\tau v$. Integer and boolean literals evaluate to themselves (C-Int, B-Const); variables read from the store (C-Read). Arithmetic evaluates both operands to integers and then applies op $\in \{+, -, *, /\}$ (C-Aff). Boolean connectives apply the usual truth functions for `and`/`or` (B-Log), and comparisons op $\in \{==, <, \leq\}$ compare integer results to yield booleans (B-Rel). Windows denote index sets $\mathcal{I}$: a point yields $\{i\}$ (W-Point); a range $c_1..c_2$ yields the inclusive set $\{\, i \in \mathbb{Z} \mid i_1 \leq i \leq i_2 \,\}$ (W-Range); and an n-tuple of windows denotes the Cartesian product $W_1 \times \cdots \times W_n$ (W-Tuple). The only nonstandard part is E#-Acc: given $a \in \mathbb{X} \cup \mathbb{B}$ and a window tuple $\mathbf{W}$, it *unpacks* the window into the set of concrete accesses $\{\, \{a \mapsto \vec{i}\} \mid \vec{i} \in \mathbf{W} \,\}$ of type $\mathbb{X} \cup \mathbb{B} \to \mathbb{Z}^*$.

5.3.2 Statement semantics. Figure 21 gives big-step operational semantics for a loop, sequencing, an explicit task–ID increment, and a guard. For `for` y in `loop`$(c_{\text{lo}}, c_{\text{hi}})$ `do` $s^{\#}$, the bounds evaluate in σ to integers m and n; the loop ranges over the *half-open* interval $[m, n)$: when $m < n$ (LoopStep) the body executes once with $y \mapsto m$, producing $(\imath', \sigma', \rho')$, and the loop recurs on $(m{+}1) \ldots n$; when $m \geq n$ (LoopDone) it terminates with no effect. Sequencing (Seq) threads the triple $(\imath, \sigma, \rho)$ from the first statement into the second. The guard (IF-True/IF-False) evaluates b in σ and either executes $s^{\#}$ or behaves as no-op. The only non-standard rule, IncTaskID, increments the task identifier ($\imath' = \imath + 1$) while leaving σ and ρ unchanged.

Figure 22 shows the rest of statement semantics. Alloc evaluates each extent c_i to an integer n_i (under $\Downarrow_{\mathbb{Z}}$), forms the index set $I = \{(v_1, \ldots, v_k) \mid 0 \leq v_i < n_i\}$, and initializes the per-access visibility record for x by updating ρ so that, for every $\mathbf{n} \in I$, $\rho(x, \mathbf{n}) = (\emptyset, \emptyset, (0, 0))$; the thread id $\imath$ and store σ are unchanged. ClearReads first resolves the accessed locations via $\langle e^{\#}, \sigma\rangle \Downarrow_{\text{Acc}} W_{e^{\#}}$ and, for each $(x, \mathbf{n}) \in W_{e^{\#}}$, replaces $\rho(x, \mathbf{n}) = (r, m, b) \in \mathbb{R}$ with $(\emptyset, m, b)$. ClearMutates is analogous but replaces it with $(r, \emptyset, b)$. RecordRead and RecordMutate first evaluate the accessor $e^{\#}$ and $e_z^{\#}$ to access sets $W_{e^{\#}}$ and $W_{e_z^{\#}}$ (via $\Downarrow_{\text{Acc}}$). They then update the visibility map ρ at each key $(x, \mathbf{n}) \in W_{e^{\#}}$ using the new record constructor $\mathcal{R}$ defined in Appendix B.

`Fence`, `Arrive`, and `Await` update ρ solely via lift(ρ, λ) (defined in Appendix B), which applies λ pointwise to both the read and mutate visibility sets. In the following, $\mathcal{W}$ decides when augmentation is permitted, $\mathcal{A}$ raises visibility to `fully_ordered` or `temporally_ordered`. For `Fence`$(t, \tau_q^{\text{pre}*}, \tau_q^{\text{full}*}, \tau_q^{\text{temp}*})$, $\lambda(r) = \mathcal{A}\Big(r,\ g^{\text{exec}*},\ \tau_q^{\text{full}*},\ \tau_q^{\text{temp}*}\Big)$ iff the witness $\mathcal{W}\big(t,\ g^{\text{exec}*},\ \tau_q^{\text{pre}*},\ r\big)$ holds, otherwise $\lambda(r) = r$; thus visibility is raised by joining $f \vee f_{\text{aug}}$. `Arrive`$(t, \tau_q^{\text{pre}*}, e_z^{\#*})$ and `Await`$(e_z^{\#}, \tau_q^{\text{full}*}, \tau_q^{\text{temp}*}, n)$ are defined similarly in Appendix B, updating the access visibility appropriately.

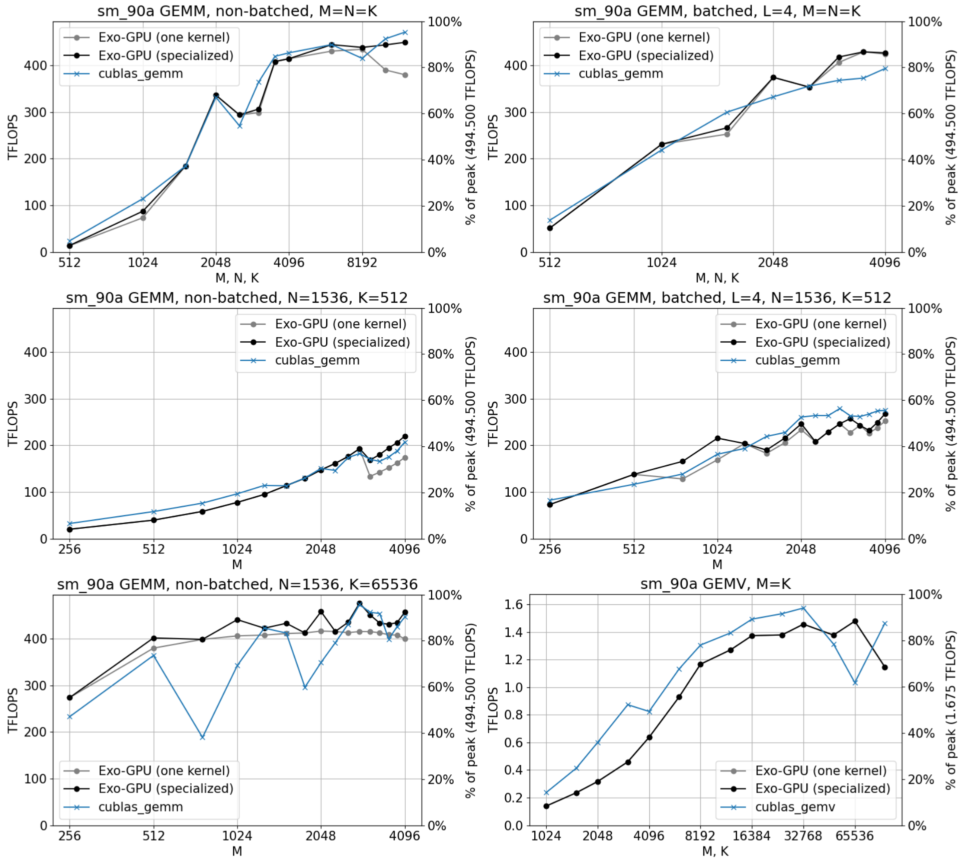


Fig. 8. Performance by varying problem size on a sm_90a GPU.

5.3.3 Checks. Figure 7 gives three pure checks that either signal error or leave (ι, σ, ρ) unchanged. CheckReads evaluates $e^\#$ to accesses $W_{e^\#}$ and raises error when some $(\_, \int, \_) \in \rho(W_{e^\#}).r$ fails to supply visibility at least τ_v for all timelines $\tau_q \in \tau_q^*$. CheckMutates is analogous but ranges over mutate visibility record $\rho(W_{e^\#}).m$. CheckBarrier evaluates $e_z^\#$ and signals error iff some $(a, w) \in \rho(W_{e_z^\#}).b$ has $a \neq w$, i.e., the recorded *arrive* and *await* counts disagree.

5.3.4 Program semantics. Finally, abstract machine executes the program `proc` y^* ; `do` $s^\#$. The synchronization check is exposed to the user, who invokes the abstract machine interpreter with concrete integer values corresponding to the control variables y^*. The control-value environment $\sigma : \mathbb{Y} \rightarrow \mathbb{Z}$ is initialized based on the user's input. The statement $s^\#$ is then evaluated according to the semantics defined previously. Note that the task id ι is initialized to 0, and the synchronization ρ is initially empty.

6 PERFORMANCE RESULTS

We implemented tf32-precision GEMM kernels for the sm_90a architecture, using `wgmma` and TMA instructions. We also implemented a full-precision fp32 GEMV kernel using warp shuffles and scalar math [12]; this did not use the accelerator instructions. We compared the Exo-GPU kernels to cuBLAS's `cublasSgemm`, `cublasSgemmStridedBatched`, and `cublasSgemv` functions, all with the math mode set to CUBLAS_TF32_TENSOR_OP_MATH. Our sm_90a-compatible test hardware contained an NVIDIA H100 80GB SXM5 GPU and an AMD EPYC 9454 48-Core Processor, and we used nvcc `Build cuda_12.3.r12.3/compiler.33492891_0` to compile the CUDA C++ sources generated by Exo-GPU. For GEMM, a compute-bound workload, theoretical peak is 494.5 TFLOPS [17]. For GEMV, a memory-bound workload, we derive the theoretical peak from the memory bandwidth: $3.35\frac{\text{TB}}{s}\frac{2\text{ FLOP}}{4\text{ B}} = 1.675\frac{\text{TFLOP}}{s}$, with one multiply and one add done for each 4 byte tf32 element loaded from the matrix (the bandwidth from loading the vector is amortized).

We tested each problem size and hardware combination over 105 iterations: 5 warm-up iterations, and 100 timed iterations, with the pcg3d hash algorithm [9] used to generate "random" input matrices. Each iteration tested, in a randomized order, the cuBLAS kernel and all applicable Exo-GPU kernel variants for that problem size. In particular, we tested non-split-k and split-k variants of the GEMM kernel, with the split-k kernels using `cp.reduce.async.bulk` (TMA) and split factors of 1, 2, 4, 8, and 16. For each (problem size, hardware) combination, we collected 100 time samples per tested kernel and reported the mean of the inter-quartile range (middle 50 samples). For each plot in Figure 8, we report the best-performing Exo-GPU kernel for each problem size (plotted as "Exo-GPU (specialized)") and the single best-performing Exo-GPU kernel ("Exo-GPU (one kernel)").

For large, square matrix GEMM problem sizes, we see very similar performance between Exo-GPU and CUBLAS; this is expected, as Exo-GPU is able to express all optimizations required for a peak-performance sm_90a gemm: warp specialization, cluster & TMA multicast support, `wgmma` support, and split-barrier synchronization using commit group, mbarrier, and cluster sync mechanisms. For more unusual GEMM problem sizes, such as small matrices and `K=65536` workloads, we see more variation between Exo-GPU and CUBLAS, possibly due to different problem size thresholds for observing wave quantization effects. For GEMV, Exo-GPU and CUBLAS follow a similar performance trend, with CUBLAS having an almost-consistent performance advantage.

Abstract Machine performance. We report the performance of the synchronization checking alone, which is implemented as an interpreter for the abstract machine (Section 5). We benchmarked using only one thread of a laptop with an "`11th Gen Intel(R) Core(TM) i7-11850H @ 2.50GHz`" CPU. In Figure 9, we report times for concrete square matrix problem sizes for the GEMV kernel and the sm_90a GEMM kernel described earlier (both the split-k and non-split-k versions). The batch size `L` is 1. We use one untimed warmup run for each problem size, and report the average of five timed runs, as reported by the `timeit` module of Python 3.10.12.

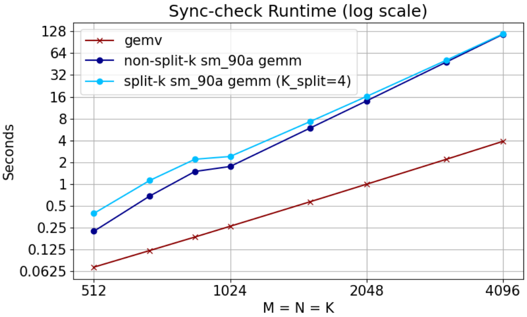


Fig. 9. Sync-check runtime by problem size.

We observe that runtimes are linear in the number of memory operations interpreted. The number of memory operations is cubic in the problem size for GEMM, and quadratic for GEMV. We see this pattern followed cleanly by the measured trend (e.g. 2/16/128 seconds for GEMM for 1024/2048/4096 problem size), indicating no unexpected overhead for large problem sizes. The bump for small GEMM problem sizes is due to imperfect division by the per-cluster tile size. Even if the expected problem size for a deployed kernel is large, during development of an Exo-GPU kernel, it generally suffices to test only with a small problem size (e.g. 768) for immediate feedback on bugs, with the option of testing the finalized kernel with the true, larger problem size for additional confidence.

7 RELATED WORK

7.1 GPU Kernel Authoring Languages

While CUDA offers fine-grained control, it remains low-level and error-prone. To address this, various approaches have emerged at different abstraction levels.

Higher-level abstractions: Tile-level DSLs like Triton [24] and Pallas [1] abstract away low-level details by enabling programmers to express computations at the block level, with compilers automatically handling the thread-level mapping. However, these abstractions can hide performance-critical details. When Triton performance fell far behind state-of-the-art for Blackwell GPUs, they had to invent Gluon, a lower-level language that gives users direct control over tile layouts, memory allocation, data movement, and asynchrony, essentially abandoning the higher-level automation to regain the fine-grained control necessary for peak performance. Rather than starting with automation that may later prove inadequate, Exo-GPU deliberately maintains a thin abstraction layer over CUDA. Performance-critical constructs, including control flow and data copies across the memory hierarchy, are made explicit statements in Exo-GPU's imperative object code.

Template-based libraries: CUTLASS [15] and ThunderKittens [22] are C++ template-based libraries that raise the abstraction of GPU programming while preserving performance. Instead of hiding CUDA behind rigid abstractions, they provide composable, architecture-aware primitives that integrate with raw CUDA code. This approach lets programmers work at the tile level when convenient, yet seamlessly drop to thread-level operations when fine control is needed. CuTe [14], introduced in CUTLASS 3.0, offers flexible abstraction for defining tensor layouts and transformations. While these libraries are effective engineering toolkits, they do not provide safety guarantees or whole-program analyses for asynchronous instructions and synchronization statements.

Achieving both peak performance and safety remains a fundamental challenge. Existing approaches either provide limited safety guarantees or lack support for modern GPU features. For instance, Descend [11] introduces Rust-like borrow checking to ensure race freedom but does not support key GPU features such as tensor cores and split barriers. The aforementioned tile- and template-based languages are practical tools, yet do not offer safety guarantees. This motivated the collective analysis (Section 4) and synchronization checking via abstract machine (Section 5).

7.2 User-schedulable Languages

User-schedulable languages (USLs) describe optimization as rewriting programs into new programs that compute the same result, but execute much faster. Rather than relying on opaque compiler heuristics, programmers directly control these rewrites by specifying scheduling operations. Halide [19, 20] popularized the idea of USLs, and subsequent Halide-inspired languages explored different target applications and abstraction for a given domain [2, 3, 5–7, 10, 23, 25, 26, 28]. For instance, TVM offers accelerator abstraction through `tensorize`, TACO separates tensor definitions from sparse formats, and Taichi abstracts memory layout and management with `fields`.

Several systems have extended user-scheduling to support modern GPU features. TVM's TensorIR [4] provides intrinsics for manipulating low-level constructs like split barriers, while Cypress [27] lets programmers describe computations as tasks and map them to different resources. Exo-GPU differs from these approaches in two key ways. First, Exo-GPU exposes low-level control to the programmer; there is little hidden control flow in GPU IR. In this sense, Exo-GPU follows a WYSIWYG (what you see is what you get) philosophy: the structure of the program directly reflects its execution. This contrasts with Cypress, which hides warp specialization as an internal compiler optimization, and with TensorIR, which focuses on automatically generating schedules with lowering-based intrinsics. Second, Exo-GPU guarantees sequential-parallel equivalence through a synchronization checker that detects data races, hazards, and other concurrency bugs.

7.3 Exo vs. Exo-GPU

Exo-GPU fundamentally extends existing the Exo language by introducing parallel semantics. While Exo [7, 8] is a high-performance USL that achieves peak performance on many vector machines and accelerators, it was designed to exploit only the *implicit parallelism* exposed by hardware. Exo's scheduling reasoning operates exclusively under a sequential execution model. This approach suffices for vector machines and matrix accelerators because CPU frontends maintain instruction ordering; even with out-of-order execution and reorder buffers, the hardware guarantees that parallel execution produces results equivalent to sequential execution. Therefore, the compiler needs only to verify program equivalence when interpreting object code sequentially. Even within this sequential framework, Exo's optimizations could exploit implicit hardware parallelism through techniques like software pipelining [7, 8].

In contrast, GPUs expose *explicit data- and instruction-parallelism* to software. Unlike traditional CPUs and accelerators that maintain the illusion of sequential execution through their frontend, modern GPUs shift this responsibility entirely to software. For Exo, this shift requires reasoning about parallel execution at the object code level and establishing equivalence not just within programming models, but across sequential and parallel paradigms. We address this fundamental challenge through two key contributions. First, we extend Exo's object code and scheduling language to expose user-controlled parallel loops and synchronization features. Second, we provide safety guarantees by establishing sequential-parallel equivalence through the abstract machine. This completes the equivalence chain from the simple, initial Exo program through to its optimized, parallel implementation in Exo-GPU, ensuring correctness at every transformation step.

8 LIMITATIONS AND FUTURE WORK

To optimize Hopper GEMMs (Figure 8), we had to apply two rewrite steps that were out-of-scope of Exo's sequential safety checks. These patterns arise when modeling TMA's asynchronous copy instruction, which supports default behavior for out-of-bounds reads and writes. Inset right figure shows a minimal example that illustrates the problem. Even though removing the first guard is safe since the `tmp[6]` and `tmp[7]` are not being read by `C` in the second loop thanks to the guard, Exo's dependency-based checks treat these writes as semantically relevant and cannot prove they are observationally redundant. We marked these transformations as *unsafe* in our schedules, despite being semantically sound. To address this issue, we need to extend Exo's sequential scheduling rewrites with value-sensitivity.

```
tmp : f32[8] @ tmp_mem
for i in seq(0, 8):
  if i < 6: # removing is safe
    tmp[i] = 0
for i in seq(0, 8):
  if i < 6:
    C[i] = tmp[i]
```

Our ability to reason about parallelism is incomplete in several ways. Our programming model and output CUDA synchronization code is generated based on our understanding of PTX as documented by NVIDIA in natural language [18]. We do not show that our generated code is correct with respect to any formal CUDA semantics. We currently do not support Blackwell Tensor Cores (`tcgen05`), nor do we support platforms not providing CUDA C++ and inline PTX support. Finally, our current synchronization checking approach – translation to abstract machine IR followed by software interpretation – supports only concrete problem sizes and limits compiler performance. This approach may be improved with static analysis of the abstract machine program.

Nevertheless, we believe that our approach—exposing detailed control to programmers while verifying its safety—offers a promising path forward for GPU programming. By treating parallelism and synchronization as analyzable annotations on sequential code, Exo-GPU provides both the transparency and static guarantees that high-performance GPU kernels demand.

REFERENCES

[1] The JAX authors. 2024. Pallas: a JAX kernel language. https://docs.jax.dev/en/latest/pallas/index.html

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12* (Salt Lake City, UT, USA). IEEE, Piscataway, NJ, USA, 66. https://doi.org/10.1109/SC.2012.71

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

[4] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3575693.3576933

[5] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodík, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *PACT '20: International Conference on Parallel Architectures and Compilation Techniques, Virtual Event, GA, USA, October 3-7, 2020*, Vivek Sarkar and Hyesoon Kim (Eds.). ACM, 71–82. https://doi.org/10.1145/3410463.3414632

[6] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.* 38, 6 (2019), 201:1–201:16. https://doi.org/10.1145/3355089.3356506

[7] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for Productive Programming of Hardware Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[8] Yuka Ikarashi, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. 2025. Exo 2: Growing a Scheduling Language. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 426–444. https://doi.org/10.1145/3669940.3707218

[9] Mark Jarzynski and Marc Olano. 2020. A PCG3D Family of Hash Functions. https://github.com/markjarzynski/PCG3D

[10] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (oct 2017), 1–29. https://doi.org/10.1145/3133901

[11] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. 2024. Descend: A Safe GPU Systems Programming Language. *Proc. ACM Program. Lang.* 8, PLDI, Article 181 (June 2024), 24 pages. https://doi.org/10.1145/3656411

[12] Bruce Lee. 2024. CUDA HGEMV. https://github.com/Bruce-Lee-LY/cuda_hgemv

[13] NVIDIA. 2017. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[14] NVIDIA. 2025. CuTe. https://docs.nvidia.com/cutlass/media/docs/cpp/cute

[15] NVIDIA. 2025. CUTLASS 4.2.1. https://github.com/NVIDIA/cutlass

[16] NVIDIA. 2025. CUTLASS grouped kernel schedules. https://docs.nvidia.com/cutlass/media/docs/cpp/grouped_scheduler.html

[17] NVIDIA. 2025. H100 GPU. https://www.nvidia.com/en-us/data-center/h100/

[18] NVIDIA. 2025. Parallel Thread Execution ISA Version 9.0. https://docs.nvidia.com/cuda/parallel-thread-execution/

[19] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12. https://doi.org/10.1145/2185520.2185528

[20] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2018. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (2018), 106–115. https://doi.org/10.1145/3150211

[21] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. arXiv:2407.08608 [cs.LG] https://arxiv.org/abs/2407.08608

[22] 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

[23] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization*

(CGO). IEEE, Piscataway, NJ, USA, 74–85. https://doi.org/10.1109/CGO.2017.7863730

[24] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[25] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730 [cs.PL]

[26] Anand Venkat, Tharindu Rusira, Raj Barik, Mary Hall, and Leonard Truong. 2019. SWIRL: High-performance many-core CPU code generation for deep neural networks. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1275–1289. https://doi.org/10.1177/1094342019866247

[27] Rohan Yadav, Michael Garland, Alex Aiken, and Michael Bauer. 2025. Task-Based Tensor Computations on Modern GPUs. *Proc. ACM Program. Lang.* 9, PLDI, Article 163 (June 2025), 25 pages. https://doi.org/10.1145/3729262

[28] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt: a high-performance graph DSL. *PACMPL* 2, OOPSLA (2018), 121:1–121:30. https://doi.org/10.1145/3276491

A GPU IR TYPE DEFINITIONS

τ_x : DataType	::=	f32	32-bit floating point
	|	f64	64-bit floating point
	|	f16	16-bit floating point
	|	i32	32-bit integer
	|	ui16	16-bit unsigned integer
	|	i8	8-bit integer
	|	ui8	8-bit unsigned integer
τ_z : BarrierType	::=	barrier	non-explicitly guarded barrier
	|	barrier(z)	explicitly guarded barrier

Fig. 10. List of variable types

			domain	*box*
τ_u : CollUnit	::=	standalone_thread	(1,)	(1,)
	|	n_1 * cuda_thread	(blockDim,)	(n_1,)
	|	cuda_quadpair	(blockDim/16, 16)	(2, 4)
	|	n_1 * cuda_warp	(blockDim,)	(n_1 * 32,)
	|	n_1 * cuda_warpgroup	(blockDim,)	(n_1 * 128,)
	|	n_1 * cuda_threads_strided(n_2, n_3)	(blockDim/n_3, n_3)	(n_1, n_2)
	|	n_1 * cuda_warp_in_cluster	(clusterDim, blockDim)	(n_1, 32)
	|	n_1 * cuda_cta_in_cluster	(clusterDim * blockDim,)	(n_1 * blockDim,)
	|	cuda_cluster	(clusterDim * blockDim,)	(clusterDim * blockDim,)
	|	n_1 * cuda_cta_in_cluster_strided(n_3)	(ClusterDim/n_3, n_3, blockDim)	(n_1, 1, blockDim)
	|	n_1 * cuda_warp_in_cluster_strided(n_3)	(clusterDim/n_3, n_3, blockDim)	(n_1, 1, 32)
	|	cuda_agnostic_sub_cta	(clusterDim, blockDim)	(1, _)
	|	cuda_agnostic_intact_cta	(clusterDim, blockDim)	(_, blockDim)

Fig. 11. Collective units defined in the frontend language. These are parameterized by a collective type δ, which consists of the *domain* and *box* shown here.

π_d : DataMemory	::=	CudaBasicDeviceVisible	base type for CUDA-visible allocations
	|	CudaBasicSmem	base type for CUDA SMEM allocations
	|	CudaDeviceVisibleAtomicity16B	base type for 16B-atomicity layout CUDA-visible allocations
	|	CudaDeviceVisibleLinear	base type for linear-order CUDA-visible allocations
	|	CudaGridConstant	CUDA grid constant memory
	|	CudaGmemLinear	CUDA global memory, linear-order
	|	CudaSmemAtomicity16B	Any CUDA shared memory with 16B-atomicity layout
	|	CudaSmemLinear	CUDA shared memory with linear-order layout
	|	CudaRmem	Per-thread CUDA registers
	|	Sm80_RmemMatrixA(M, K)	Matrix tile for sm_80+ warp MMA A operand
	|	Sm80_RmemMatrixB(N, K)	Matrix tile for sm_80+ warp MMA B operand
	|	Sm80_RmemMatrixD(M, N)	Matrix tile for sm_80+ MMA acumulator operands
	|	Sm90_SmemSwizzled(B)	B-byte swizzled shared memory, in TMA/wgmma-accepted format
	|	Sm90_RmemMatrixA(M)	Register tile for wgmma A operand (not implemented)
	|	Sm90_RmemMatrixD(M, N)	Register tile for wgmma D operand

Fig. 12. CUDA Memory Types – Data Allocations (pre-existing Exo Memory not listed)

π_z : BarrierComp	::=	`CudaDeviceBarrier`	base type for CUDA-allocated barriers
	|	`CudaMbarrier`	CUDA `mbarrier`
	|	`CudaCommitGroup`	CUDA `commit_group` mechanism
	|	`CudaClusterSync`	`barrier.cluster` sync

Fig. 13. CUDA Barrier (completion) mechanisms

π_w : SpecialWindow	::=	`Sm90_tensorMap(swizzle, *smem_box)`	`CUtensorMap` parameterized by swizzle mode and box size

Fig. 14. CUDA Special Window Types (for Window Statement)

τ_q : QualTL	::=	`cpu_in_order_qual`	Ordinary CPU reads/writes (cpu)
	|	`cpu_cuda_stream_qual`	Stream-ordered CUDA API calls (strm)
	|	`cuda_in_order_rmem_qual`	non-async CUDA instrs' accesses to registers (cuda1)
	|	`cuda_in_order_ram_qual`	non-async CUDA instrs' accesses to SMEM/GMEM (cuda2)
	|	`Sm80_cp_async_qual`	non-bulk `cp.async` memory accesses (Sm80)
	|	`tma_to_smem_async_qual`	`cp.async.bulk` GMEM reads, SMEM writes (tmaS)
	|	`tma_to_gmem_async_qual`	`cp.async.bulk` SMEM reads, GMEM writes/reduces (tmaG)
	|	`wgmma_async_rmem_a_qual`	`wgmma.mma_async` reads from A register parameter (wgA)
	|	`wgmma_async_rmem_d_qual`	`wgmma.mma_async` accesses to D (accumulator) (wgD)
	|	`wgmma_async_smem_qual`	`wgmma.mma_async` access to SMEM, A or B (wgS)
	|	`wgmma_zero_qual`	special case used to model the wgmma scale-d parameter (wg0)

Fig. 15. List of qualitative timelines

τ_s : SyncTL	trnstv?	cpu	strm	cuda1	cuda2	Sm80	tmaS	tmaG	wgA	wgD	wgS	wg0
`empty_sync_tl`												
`cpu_in_order`	y	full										
`cuda_stream_sync`	y		full	full	full	full	full	full	full	full	full	
`cuda_in_order`	y		temp.	full	full	temp.	temp.	temp.	temp.	temp.	temp.	temp.
`cuda_temporal`			temp.	temp.	temp.	temp.	temp.	temp.	temp.	temp.	temp.	temp.
`Sm80_cp_async`						full						
`Sm80_generic`			temp.	full	full	full	temp.	temp.	temp.	temp.	temp.	temp.
`tma_to_smem_async`							full					
`tma_to_gmem_async`								full				
`wgmma_async_smem`											full	
`wgmma_fence_1`				full					full	full		
`wgmma_fence_2`									full	full		
`wgmma_async`									full	full	full	
`cuda_async_proxy`							full	full			full	
`cuda_async_proxy_wgmma`							full	full	full	full	full	
`cuda_generic_and_async`			temp.	full	full	full	full	full	temp.	temp.	full	temp.

Fig. 16. SyncTL are defined as composition of QualTL.

B ABSTRACT MACHINE CONVERSION AND SEMANTICS DEFINITIONS

A new record Constructor $\mathcal{R}$ is used to define `RecordRead` and `RecordMutate` in Figure 22. RecordRead and RecordMutate first evaluate the accessor $e^{\#}$ and $e_z^{\#}$ to access sets $W_{e^{\#}}$ and $W_{e_z^{\#}}$ (via $\Downarrow_{\text{Acc}}$). They then update the visibility map ρ at each key $(x, \mathbf{n}) \in W_{e^{\#}}$ using the new record constructor $\mathcal{R}$ defined below: RecordRead adds it to the *read* visibility set, setting $\rho'(x, \mathbf{n}).r = \rho(x, \mathbf{n}).r \cup \mathcal{R}(\cdots)$ while leaving $\rho(x, \mathbf{n}).m$ and $\rho(x, \mathbf{n}).b$ unchanged; RecordMutate adds it to the *mutate* visibility set, setting $\rho'(x, \mathbf{n}).m = \rho(x, \mathbf{n}).m \cup \mathcal{R}(\cdots)$ while preserving $\rho(x, \mathbf{n}).r$ and $\rho(x, \mathbf{n}).b$. In both cases, neither the control $\imath$ nor the store σ changes.

Definition B.1 (New Record Constructor $\mathcal{R}$). Define $\mathcal{R}$ as follows:

$$\mathcal{R} :\ \mathrm{VL} \times \mathrm{Bool} \times \mathrm{QualTL} \times \mathcal{P}(\mathrm{QualTL}) \times \mathrm{Expr}^{\#} \times \rho \times \mathbb{G} \ \longrightarrow\ \mathcal{P}(\mathbb{R})$$

$$\mathcal{R}(\tau_v, t, \tau_q, \tau_q^*, E^{\#}, \rho, G) \ = \ \begin{cases} \{\, (\tau_q,\ S_G,\ P_{E^{\#},\rho}) \,\}, & t = \mathsf{true}, \\ \{\, (\tau_q,\ S_{\{g\}},\ P_{E^{\#},\rho}) \mid g \in G \,\}, & t = \mathsf{false}, \end{cases}$$

where, for any $H \subseteq \mathbb{G}$, $S_H \in \mathbb{S}$ and $P_{E^{\#},\rho} \in \mathbb{P}$ are given pointwise by

$$S_H(g', \tau_q') \ = \ \max_{\mathrm{VL}}\left(\begin{cases} \mathsf{atomic\_only}, & \tau_q' \in \tau_q^*, \\ \mathsf{invisible}, & \text{otherwise} \end{cases}, \begin{cases} \tau_v, & g' \in H \ \wedge \ \tau_q' = \tau_q, \\ \mathsf{invisible}, & \text{otherwise} \end{cases} \right),$$

$$P_{E^{\#},\rho}(z, \mathbf{n}) \ = \ \begin{cases} \{a\}, & (z, \mathbf{n}) \in E^{\#} \text{ where } (a, \_) = \rho(z, \mathbf{n}).b, \\ \emptyset, & \text{otherwise.} \end{cases}$$

`Fence`, `Arrive`, and `Await` (Figure 22) update ρ solely via $\mathrm{lift}(\rho, \lambda)$ (defined below), which applies λ pointwise to both the read and mutate visibility sets. For `Fence`$(t, \tau_q^{\mathrm{pre}*}, \tau_q^{\mathrm{full}*}, \tau_q^{\mathrm{temp}*})$, $\lambda(r) = \mathcal{A}\left(r,\ g^{\mathrm{exec}*},\ \tau_q^{\mathrm{full}*},\ \tau_q^{\mathrm{temp}*}\right)$ iff the witness $\mathcal{W}\left(t,\ g^{\mathrm{exec}*},\ \tau_q^{\mathrm{pre}*},\ r\right)$ holds, otherwise $\lambda(r) = r$; thus visibility is raised by joining $f \vee f_{\mathrm{aug}}$. For `Arrive`$(t, \tau_q^{\mathrm{pre}*}, e_z^{\#*})$, the accessed sets yield a unique home barrier $\{(z, \mathbf{n})\} = \bigcap_i W_{e^{\#}_{zi}}$; guarded by $\mathcal{W}$, λ adds the existing arrive count a from $\rho(z, \mathbf{n})$ into $r.p$ at every $(z, \mathbf{n}) \in \mathbf{W}$, and home barrier's arrive count is incremented by $(a' + 1, w')$. For `Await`$(e_z^{\#}, \tau_q^{\mathrm{full}*}, \tau_q^{\mathrm{temp}*}, n)$, with $(a, w) = \rho(z, \mathbf{n}).b$, compute Max and New as in the rule; set $\lambda(r) = \mathcal{A}\left(r,\ g_{s^{\#}}(\rho),\ \tau_q^{\mathrm{full}*},\ \tau_q^{\mathrm{temp}*}\right)$ iff some observed arrive count $i \in r.p(z, \mathbf{n})$ satisfies $i \leq \mathrm{Max}$, else $\lambda(r) = r$; after lift, update the wait count to (a', New). In short, $\mathcal{W}$ decides when augmentation is permitted, $\mathcal{A}$ raises visibility to `fully_ordered` or `temporally_ordered`.

Definition B.2 (Lift). For $\lambda : \mathbb{R} \to \mathbb{R}$, define $\mathrm{lift}(\rho, \lambda)$ pointwise by

$$\mathrm{lift}(\rho, \lambda)(x, \mathbf{v}) \ = \ \big(\{\, \lambda(r) \mid r \in R \,\},\ \{\, \lambda(r) \mid r \in M \,\},\ (a,\ w)\big) \quad \text{where } (R, M, (a, w)) = \rho(x, \mathbf{v}).$$

Definition B.3 (Synchronization helpers). Let $r = (\tau_q^{\mathrm{orig}}, f, p) \in \mathbb{R}$ with $f \in \mathbb{S}$. For $f_1, f_2 \in \mathbb{S}$, define the pointwise join $(f_1 \vee f_2)(g, \tau_q) \ = \ \max_{\mathrm{VL}}\{f_1(g, \tau_q),\ f_2(g, \tau_q)\}$.

(i) Witness $\mathcal{W}$. For $t \in \mathrm{Bool}$, $g^{\mathrm{exec}*} \subseteq \mathbb{G}$, and $\tau_q^{\mathrm{pre}*} \subseteq \mathrm{QualTL}$, set

$$\mathcal{W}(t, g^{\mathrm{exec}*}, \tau_q^{\mathrm{pre}*}, r) = \begin{cases} \exists g \in g^{\mathrm{exec}*}\ \exists \tau_q \in \tau_q^{\mathrm{pre}*}\ :\ f(g, \tau_q) \geq \mathsf{unordered}, & t = \mathsf{true}, \\ \left(\tau_q^{\mathrm{orig}} \in \tau_q^{\mathrm{pre}*}\right)\ \wedge\ \exists g \in g^{\mathrm{exec}*}\ :\ f(g, \tau_q^{\mathrm{orig}}) \geq \mathsf{unordered}, & t = \mathsf{false}. \end{cases}$$

(ii) Augment $\mathcal{A}$. Given $g^{\mathrm{exec}*} \subseteq \mathbb{G}$ and $\tau_q^{\mathrm{full}*}, \tau_q^{\mathrm{temp}*} \subseteq \mathrm{QualTL}$, define

$$\mathcal{A}(r, g^{\mathrm{exec}*}, \tau_q^{\mathrm{full}*}, \tau_q^{\mathrm{temp}*}) \ = \ \left(\tau_q^{\mathrm{orig}},\ f \vee f_{\mathrm{aug}},\ p\right),$$

$$f_{\mathrm{aug}}(g, \tau_q) = \begin{cases} \mathsf{fully\_ordered}, & g \in g^{\mathrm{exec}*} \ \wedge\ \tau_q \in \tau_q^{\mathrm{full}*}, \\ \mathsf{temporally\_ordered}, & g \in g^{\mathrm{exec}*} \ \wedge\ \tau_q \in \tau_q^{\mathrm{temp}*} \setminus \tau_q^{\mathrm{full}*}, \\ \mathsf{invisible}, & \text{otherwise.} \end{cases}$$

Data expressions $e \in \text{DExpr}$	$\mathcal{P}(\text{Expr}^{\#})$
d	$\emptyset$
x	$\{\, x[\langle\rangle] \,\}$
$x[w^{*}]$	$\{\, x[w^{*}] \,\}$
$e_1 \text{ op } e_2 \quad (\text{op} \in \{+, -, *, /\})$	$T_e(e_1) \cup T_e(e_2)$
Barrier expressions $e_z \in \text{ZExpr}$	
z	$\{\, z[\langle\rangle] \,\}$
$z[w^{*}]$	$\{\, z[w^{*}] \,\}$

Fig. 17. Expression conversion $T_e : \text{DExpr} \cup \text{ZExpr} \to \mathcal{P}(\text{Expr}^{\#})$ collects the distinct reads in an expression and returns them as a set of AMIR reads ($\text{Expr}^{\#}$).

Type	**Definition**
$T_{\text{smem}} : \text{Expr} \to \mathcal{P}(\text{Expr}^{\#})$	$T_{\text{smem}}(e) \triangleq \{\, e^{\#} \in T_e(e) \mid \text{isSmem}(e^{\#}) \,\}.$
$T_{\text{ex}} : \text{Expr} \to \mathcal{P}(\text{Expr}^{\#})$	$T_{\text{ex}}(e) \triangleq \{\, e^{\#} \in T_e(e) \mid \neg\, \text{isEx}(e^{\#}) \,\}.$
$\text{isEx} : \text{Expr}^{\#} \to \text{Bool}$	$\text{isEx}(e^{\#}) \triangleq$ `true` iff $e^{\#}$ is annotated as sync-exempt memory; `false` otherwise.
$\text{isSmem} : \text{Expr}^{\#} \to \text{Bool}$	$\text{isSmem}(e^{\#}) \triangleq$ `true` iff $e^{\#}$ is annotated as shared memory; `false` otherwise.
$T_{\text{tr}} : \text{SyncTL} \to \text{Bool}$	$T_{\text{tr}}(\tau_s) \triangleq$ `true` iff τ_s is transitive; `false` otherwise.
$T_{\text{qfull}} : \text{SyncTL} \to \mathcal{P}(\text{QualTL})$	$T_{\text{qfull}}(\tau_s) \triangleq \{\, \tau'_q \in \text{QualTL} \mid \tau'_q$ annotated as "full" in $\tau_s \,\}.$
$T_{\text{qtmp}} : \text{SyncTL} \to \mathcal{P}(\text{QualTL})$	$T_{\text{qtmp}}(\tau_s) \triangleq \{\, \tau'_q \in \text{QualTL} \mid \tau'_q$ annotated as "temporal" or "full" in $\tau_s \,\}.$
$T_{\text{qarr}} : \text{SyncTL} \to \text{QualTL}$	$T_{\text{qarr}}(\tau_s) \triangleq \begin{cases} \texttt{Sm80\_cp\_async\_qual}, & \text{if } \texttt{Sm80\_cp\_async\_qual} \in T_{\text{qfull}}(\tau_s), \\ \texttt{cuda\_in\_order\_qual}, & \text{otherwise.} \end{cases}$
$T_{\text{qin}} : \text{Expr}^{\#} \to \text{QualTL}$	$T_{\text{qin}}(e^{\#}) \triangleq \begin{cases} \texttt{cpu\_in\_order\_qual}, & \text{if } e^{\#} \text{ is host/CPU scoped,} \\ \texttt{cuda\_in\_order\_rmem\_qual}, & \text{if } e^{\#} \text{ is register memory,} \\ \texttt{cuda\_in\_order\_ram\_qual}, & \text{otherwise.} \end{cases}$

Fig. 18. Helper functions used by Fig. 5 statement conversion. All sets are finite.

Case for $g.p_i$	**Abstract Machine IR**
Non-atomic *read-only*	$\mathop{;}\limits_{e^\# \in T_{\text{ex}}(e_i)} \Big(\texttt{CheckMutates(full\_ordered}, T_{\text{cvg}}(g.p_i), e^\#,\ T_{\text{ext}}(g.p_i)\texttt{);}$ $\texttt{RecordRead(}\tau_v^{\text{post}}, T_{\text{cvg}}(g.p_i), e^\#,\ T_{\text{init}}(g.p_i),\ \emptyset, T_e(e_z)\texttt{))}\Big)$
Non-atomic *non-read-only*	$\mathop{;}\limits_{e^\# \in T_{\text{ex}}(e_i)} \Big(\texttt{CheckMutates(}\tau_v^{\text{pre}}, T_{\text{cvg}}(g.p_i), e^\#,\ T_{\text{ext}}(g.p_i)\texttt{);}$ $\texttt{CheckReads(temporal\_ordered}, T_{\text{cvg}}(g.p_i), e^\#,\ T_{\text{ext}}(g.p_i)\texttt{);}$ $\texttt{ClearReads(}e^\#\texttt{);}$ $\texttt{ClearMutates(}e^\#\texttt{);}$ $\texttt{RecordMutate(}\tau_v^{\text{post}}, T_{\text{cvg}}(g.p_i), e^\#,\ T_{\text{init}}(g.p_i),\ \emptyset, T_e(e_z)\texttt{))}\Big)$ where $\tau_v^{\text{pre}} = \texttt{temporal\_ordered}$ if $g.p_i$ is write-only, else $\texttt{full\_ordered}$.
Atomic	$\mathop{;}\limits_{e^\# \in T_{\text{ex}}(e_i)} \Big(\texttt{CheckMutates(atomic\_only}, T_{\text{cvg}}(g.p_i), e^\#,\ T_{\text{ext}}(g.p_i)\texttt{);}$ $\texttt{CheckReads(temporal\_ordered}, T_{\text{cvg}}(g.p_i), e^\#,\ T_{\text{ext}}(g.p_i)\texttt{);}$ $\texttt{ClearReads(}e^\#\texttt{);}$ *(note lack of ClearMutates)* $\texttt{RecordMutate(}\tau_v^{\text{post}}, T_{\text{cvg}}(g.p_i), e^\#,\ T_{\text{init}}(g.p_i), T_{\text{atom}}(g.p_i), T_e(e_z)\texttt{))}\Big)$

Fig. 19. Argument conversion $T_{\text{arg}}(g.p_i, e_i, e_z)$. $\tau_v^{\text{post}} = \texttt{unordered}$ if $T_{\text{ooo}}(g.p_i)$, else $\texttt{full\_ordered}$.

Type	**Definition**
$T_{\text{ext}} : \text{Expr} \to \mathcal{P}(\text{QualTL})$	$T_{\text{ext}}(e) \triangleq \{\, \tau_q \in \text{QualTL} \mid \tau_q \in e.\texttt{ext\_qual\_tl} \,\}$.
$T_{\text{atom}} : \text{Expr} \to \mathcal{P}(\text{QualTL})$	$T_{\text{atom}}(e) \triangleq \{\, \tau_q \in \text{QualTL} \mid \tau_q \in e.\texttt{atomic\_qual\_tl} \,\}$.
$T_{\text{init}} : \text{Expr} \to \text{QualTL}$	$T_{\text{init}}(e) \triangleq e.\texttt{initial\_qual\_tl}$.
$T_{\text{ooo}} : \text{Expr} \to \text{Bool}$	$T_{\text{ooo}}(e) \triangleq e.\texttt{out\_of\_order}$
$T_{\text{cvg}} : \text{Expr} \to \text{Bool}$	$T_{\text{cvg}}(e) \triangleq e.\texttt{convergent}$

Fig. 20. Helper functions used by Fig. 19 argument conversion.

$$\frac{\begin{array}{c}\langle c_{\text{lo}}, \sigma\rangle \Downarrow_{\mathbb{Z}} m \quad \langle c_{\text{hi}}, \sigma\rangle \Downarrow_{\mathbb{Z}} n \quad m < n \quad \langle s, \iota, \sigma[y \mapsto m], \rho\rangle \Downarrow \iota', \sigma', \rho' \\ \langle \text{for } y \text{ in } (m{+}1) \ldots n \text{ do } s^\#, \iota', \sigma', \rho'\rangle \Downarrow \iota'', \sigma'', \rho''\end{array}}{\langle \text{for } y \text{ in } \texttt{loop}(c_{\text{lo}}, c_{\text{hi}}) \text{ do } s^\#, \iota, \sigma, \rho\rangle \Downarrow \iota'', \sigma'', \rho''}\ \textsc{LoopStep}$$

$$\frac{\langle c_{\text{lo}}, \sigma\rangle \Downarrow_{\mathbb{Z}} m \quad \langle c_{\text{hi}}, \sigma\rangle \Downarrow_{\mathbb{Z}} n \quad m \geq n}{\langle \text{for } y \text{ in } \texttt{loop}(c_{\text{lo}}, c_{\text{hi}}) \text{ do } s^\#, \iota, \sigma, \rho\rangle \Downarrow \iota, \sigma, \rho}\ \textsc{LoopDone}$$

$$\frac{\langle s_1^\#, \iota, \sigma, \rho\rangle \Downarrow \iota', \sigma', \rho' \qquad \langle s_2^\#, \iota', \sigma', \rho'\rangle \Downarrow \iota'', \sigma'', \rho''}{\langle s_1^\#;\ s_2^\#, \iota, \sigma, \rho\rangle \Downarrow \iota'', \sigma'', \rho''}\ \textsc{Seq} \qquad \frac{\iota' = \iota + 1}{\langle \text{InctaskID}, \iota, \sigma, \rho\rangle \Downarrow \iota', \sigma, \rho}\ \textsc{IncTaskID}$$

$$\frac{\langle b, \sigma\rangle \Downarrow_{\text{Bool}} \text{true} \quad \langle s^\#, \iota, \sigma, \rho\rangle \Downarrow \iota', \sigma', \rho'}{\langle \text{if } b \text{ then } s^\#, \iota, \sigma, \rho\rangle \Downarrow \iota', \sigma', \rho'}\ \text{IF-}\textsc{True} \qquad \frac{\langle b, \sigma\rangle \Downarrow_{\text{Bool}} \text{false}}{\langle \text{if } b \text{ then } s^\#, \iota, \sigma, \rho\rangle \Downarrow \iota, \sigma, \rho}\ \text{IF-}\textsc{False}$$

Fig. 21. Big-step semantics for a loop, sequencing, an explicit task–ID increment, and a guard.

$$\frac{\begin{array}{c} c^{*} = (c_1, \ldots, c_k) \quad \left\langle c_i, \sigma \right\rangle \Downarrow_{\mathbb{Z}} n_i \ \text{ for } i = 1..k \\ I \ = \ \{\, (v_1, \ldots, v_k) \mid 0 \leq v_i < n_i \,\} \qquad \rho' \ = \ \rho\big[(x, \mathbf{n}) \mapsto (\emptyset, \emptyset, (0,0))\big]_{\mathbf{n} \in I} \end{array}}{\langle \mathrm{Alloc}(x[c^{*}]),\ \imath,\ \sigma,\ \rho \rangle \Downarrow \imath, \sigma, \rho'} \ \textsc{Alloc}$$

$$\frac{\langle e^{\#}, \sigma \rangle \Downarrow_{\mathrm{Acc}} W_{e^{\#}} \qquad \rho' \ = \ \rho\big[(x, \mathbf{n}) \mapsto (\, \emptyset,\ \rho(x, \mathbf{n}).m,\ \rho(x, \mathbf{n}).b \,)\big]_{\ (x,\mathbf{n}) \in W_{e^{\#}}}}{\big\langle \mathrm{ClearReads}(e^{\#}),\ \imath,\ \sigma,\ \rho \big\rangle \Downarrow \ \imath,\ \sigma,\ \rho'} \ \textsc{ClearReads}$$

$$\frac{\langle e^{\#}, \sigma \rangle \Downarrow_{\mathrm{Acc}} W_{e^{\#}} \qquad \rho' \ = \ \rho\big[(x, \mathbf{n}) \mapsto (\, \rho(x, \mathbf{n}).r, \emptyset, \rho(x, \mathbf{n}).b \,)\big]_{\ (x,\mathbf{n}) \in W_{e^{\#}}}}{\big\langle \mathrm{ClearMutates}(e^{\#}),\ \imath,\ \sigma,\ \rho \big\rangle \Downarrow \ \imath,\ \sigma,\ \rho'} \ \textsc{ClearMutates}$$

$$\frac{\begin{array}{c} \langle e^{\#}, \sigma \rangle \Downarrow_{\mathrm{Acc}} W_{e^{\#}} \quad \langle e^{\#}_{z}, \sigma \rangle \Downarrow_{\mathrm{Acc}} W_{e^{\#}_{z}} \\ \rho' \ = \ \rho\big[(x, \mathbf{n}) \mapsto (\, \rho(x, \mathbf{n}).r \cup \mathcal{R}(\tau_v,\ t,\ \tau_q,\ \tau^{*}_{q},\ W_{e^{\#}_{z}}, \rho, g_{s^{\#}}(\sigma, \imath)),\ \rho(x, \mathbf{n}).m,\ \rho(x, \mathbf{n}).b \,)\big]_{\ (x,\mathbf{n}) \in W_{e^{\#}}} \end{array}}{\langle \mathrm{RecordRead}(\tau_v,\ t,\ e^{\#},\ \tau_q,\ \tau^{*}_{q},\ e^{\#}_{z}),\ \imath,\ \sigma,\ \rho \rangle \Downarrow \imath, \sigma, \rho'} \ \textsc{RecordRead}$$

$$\frac{\begin{array}{c} \langle e^{\#}, \sigma \rangle \Downarrow_{\mathrm{Acc}} W_{e^{\#}} \quad \langle e^{\#}_{z}, \sigma \rangle \Downarrow_{\mathrm{Acc}} W_{e^{\#}_{z}} \\ \rho' \ = \ \rho\big[(x, \mathbf{n}) \mapsto (\, \rho(x, \mathbf{n}).r,\ \rho(x, \mathbf{n}).m \cup \mathcal{R}(\tau_v,\ t,\ \tau_q,\ \tau^{*}_{q},\ W_{e^{\#}_{z}}, \rho, g_{s^{\#}}(\sigma, \imath)),\ \rho(x, \mathbf{n}).b \,)\big]_{\ (x,\mathbf{n}) \in W_{e^{\#}}} \end{array}}{\langle \mathrm{RecordMutate}(\tau_v,\ t,\ e^{\#},\ \tau_q,\ \tau^{*}_{q},\ e^{\#}_{z}),\ \imath,\ \sigma,\ \rho \rangle \Downarrow \imath, \sigma, \rho'} \ \textsc{RecordMutate}$$

$$\frac{\lambda \ = \ \big(r \mapsto \mathcal{A}(r, g_{s^{\#}}(\sigma, \imath), \tau^{\mathrm{full}*}_{q}, \tau^{\mathrm{temp}*}_{q}) \text{ if } \mathcal{W}(t, g_{s^{\#}}(\sigma, \imath), \tau^{\mathrm{pre}*}_{q}, r) \text{ else } r\big)}{\big\langle \mathtt{Fence}(t, \tau^{\mathrm{pre}*}_{q}, \tau^{\mathrm{full}*}_{q}, \tau^{\mathrm{temp}*}_{q}),\ \imath,\ \sigma,\ \rho \big\rangle \Downarrow \ \imath,\ \sigma,\ \mathrm{lift}(\rho, \lambda)} \ \textsc{Fence}$$

$$\frac{\begin{array}{c} \langle \langle e^{\#}_{z1}, \cdots, e^{\#}_{zk} \rangle, \sigma \rangle \Downarrow_{\mathrm{Acc}} W_{e^{\#}_{z1}}, \cdots, W_{e^{\#}_{zk}} \quad \{(z, \mathbf{n})\} = \bigcap_{i=1}^{k} W_{e^{\#}_{zi}} \quad \mathbf{W} = \bigcup_{i=1}^{k} W_{e^{\#}_{zi}} \\ \lambda \ = \ \big(r \mapsto \big(r.q,\ r.\int,\ r.p[(z, \mathbf{n}) \mapsto r.p(z, \mathbf{n}) \cup \{a\}]^{(a,\_)=\rho(z,\mathbf{n})}_{z,\mathbf{n} \in \mathbf{W}}\big) \ \text{if } \mathcal{W}(t, g_{s^{\#}}(\sigma, \imath), \tau^{\mathrm{pre}*}_{q}, r) \ \text{ else } \ r\,\big) \\ \rho' = \mathrm{lift}(\rho, \lambda) \qquad (R', M', (a', w')) \ = \ \rho'(z, \mathbf{n}) \qquad \rho'' \ = \ \rho'[(z, \mathbf{n}) \mapsto (R', M', (a' + 1, w'))] \end{array}}{\big\langle \mathtt{Arrive}(t, \tau^{\mathrm{pre}*}_{q}, e^{\#*}_{z}),\ \imath,\ \sigma,\ \rho \big\rangle \Downarrow \ \imath,\ \sigma,\ \rho''} \ \textsc{Arrive}$$

$$\frac{\begin{array}{c} \langle e^{\#}_{z}, \sigma \rangle \Downarrow_{\mathrm{Acc}} \{(z, \mathbf{n})\} \quad (a, w) = \rho(z, \mathbf{n}).b \qquad \mathrm{Max} \ \triangleq \ \text{if } n \geq 0 \text{ then } a - (n+1) \text{ else } w + n + 1 \\ \mathrm{New} \ \triangleq \ \text{if } n \geq 0 \text{ then } \ \max\big(w,\ \mathrm{Max} + 1\big) \text{ else } w + 1 \\ \lambda \ = \ \Big(r \mapsto \mathcal{A}\big(r,\ g_{s^{\#}}(\rho),\ \tau^{\mathrm{full}*}_{q},\ \tau^{\mathrm{temp}*}_{q}\big) \text{ if } \exists i.\ i \leq \mathrm{Max} \wedge i \in r.p(z, \mathbf{n}) \ \text{ else } \ r\Big) \\ \rho' = \mathrm{lift}(\rho, \lambda) \qquad (R', M', (a', \_)) \ = \ \rho'(z, \mathbf{n}) \qquad \rho'' \ = \ \rho'[(z, \mathbf{n}) \mapsto (R', M', (a', \mathrm{New}))] \end{array}}{\big\langle \mathtt{Await}(e^{\#}_{z},\ \tau^{\mathrm{full}*}_{q},\ \tau^{\mathrm{temp}*}_{q},\ n),\ \imath,\ \sigma,\ \rho \big\rangle \Downarrow \ \imath,\ \sigma,\ \rho''} \ \textsc{Await}$$

Fig. 22. The big-step operational semantics for Abstract Machine statements with sync environment updates.