Modular GPU Programming with Typed Perspectives

ANONYMOUS AUTHOR(S)

To achieve peak performance on modern GPUs, one must balance two frames of mind: issuing instructions to individual threads to control their behavior, while simultaneously tracking the convergence of many threads acting in concert to perform *collective operations* like Tensor Core instructions. The tension between these two mindsets makes modular programming error prone. Functions that encapsulate collective operations, despite being called per-thread, must be executed cooperatively by groups of threads.

In this work, we introduce Prism, a new GPU language that restores modularity while still giving programmers the low-level control over collective operations necessary for high performance. Our core idea is *typed perspectives*, which materialize, at the type level, the granularity at which the programmer is controlling the behavior of threads. We describe the design of Prism, implement a compiler for it, and lay its theoretical foundations in a core calculus called Bundl. We implement state-of-the-art GPU kernels in Prism and find that it offers programmers the safety guarantees needed to confidently write modular code without sacrificing performance.

1 Introduction

CUDA [39] is a low-level, imperative programming language for NVIDIA GPUs. These GPUs are organized into a hierarchy of compute resources. *Threads* are the basic unit of sequential execution, *blocks* are groups of threads that can cooperate through shared scratchpad memory, and the *grid* is the full collection of blocks launched for a computation. A GPU program executes in parallel across this hierarchy, but is written *from the perspective of a single thread.*

While operations are specified per-thread, some are only valid when executed *collectively* by a group of threads. The **`__syncthreads()`** intrinsic, for instance, synchronizes all threads within a block, and it will cause a deadlock if executed by only some of those threads. There are many such collective intrinsics, including Tensor Core instructions [41–43], warp shuffle operations [22], and other kinds of synchronization primitives [21]. These operations require programmers to carefully marshal compute resources to coordinate which threads execute which lines of code. As a result, such collective operations break the illusion of threads executing independently.

The conceptual clash between regular statements (executed by a single thread) and collective operations (executed by a group of threads) impacts not only hardware intrinsics, but also *user-defined* collective operations, like functions. While functions are invoked individually by threads, they may encapsulate collective behavior, creating a contradiction between the per-thread syntax of their invocation and the cooperative semantics of their execution.

This contradiction puts modularity at odds with correctness and is apparent even in widely used CUDA libraries that package common functionality via function interfaces [13, 38, 51]. Consider the following snippet of documentation taken directly from CUB [38], a library of parallel primitives. It describes the **`BlockReduce`** function [11], in which all threads in a block collaboratively apply a reduction operation, such as a maximum or prefix sum, over an array:

> Computes a <u>block-wide</u> reduction for thread$_0$ using the specified binary reduction functor.
> - The return value is undefined in threads other than thread$_0$.
> - A subsequent **`__syncthreads()`** threadblock barrier should be invoked after calling this method if the collective's temporary storage (e.g., **`temp_storage`**) is to be reused or repurposed.

The snippet above attempts to convey several assumptions implicit in **`BlockReduce`**'s implementation. First, the function is only well defined when invoked by all threads in a block. Second, because it accesses memory shared among threads, any subsequent reuse of that storage requires synchronization to ensure all threads have completed their accesses. In effect, CUB attempts to retrofit CUDA with information about the compute and memory requirements of collective operations. By prefixing functions with identifiers such as **`Block`**, CUB creates ad-hoc "namespaces" for different functions that assume similar invariants. However, without a type system to statically enforce these invariants, correct usage of this function—and collective operations in general—depends on carefully reading and interpreting the documentation.

In this work, we ask: can we provide low-level control over collective operations while statically guaranteeing that they execute with the necessary compute resources? By reifying configurations of compute resources as type-level *perspectives*—so named because they describe the view of GPU resources against which each statement is written—we find that we can. Our key insight is that GPU programmers naturally map computations onto different compute resources, and, unlike CUDA, which obscures this mapping, we can track it with a type system. This tracking enforces that collective operations are executed with the necessary resources, while still allowing users to access low-level intrinsics. Further, it allows users to define and compose their own collective operations by specifying the perspective with which their code must be run.

Our approach departs from previous work, which attempts to resolve the mismatch between per-thread syntax and collective execution by restricting access to low-level operations. Tile-based languages like Triton [54], Helion [44], and Tilus [26], limit the user's perspective to the block level, which prevents them from writing the highest-performance kernels. A variety of functional languages, [7, 27, 32] provide compile-time guarantees through their type systems but do not expose low-level control over hardware. Some other efforts, like Descend [35], aim to provide a low-level, memory-safe GPU systems programming language, but lack support for modern GPU features (like Tensor Cores or asynchrony) entirely. As a result, despite its flaws, CUDA remains the de-facto standard for writing high-performance kernels on modern GPUs [23].

We introduce Prism, a new low-level GPU programming language that enforces that operations are only executed with the correct view of hardware resources, enabling fearless composition. Inspired by dependency calculi [1], Prism statically tracks the configurations of compute and memory resources with type-level perspectives. We also develop Bundl, a core calculus underpinning Prism, provide formal rules for manipulating the perspectives of both code and data, and prove *type-and-perspective safety*. This ensures that Bundl is sound, and that code always has the right perspective to execute operations at run time. A parallel goal, alongside safe composition, is performance. We incorporate modern GPU features such as Tensor Cores and asynchronous data movement into Prism, and demonstrate that Prism can achieve the same performance as hand-written, highly optimized code on an H100 and a 4070 Ti Super. Our contributions are:

(1) Prism, a low-level GPU language that tracks the logical grouping of compute and memory resources with type-level perspectives, empowering users to write modular code (Section 3);
(2) Bundl, a formal model of Prism that tracks perspectives in its type system and operational semantics, along with a soundness theorem guaranteeing that programs execute operations only when they have been statically proven to possess the necessary perspective (Section 4); and
(3) An implementation of Prism (Section 5) that demonstrates that it can support modern GPU features like Tensor Cores and asynchronous data movement, achieving performance comparable to hand-optimized CUDA implementations (Section 6).

Section 7 discusses related work, and Section 8 discusses the limitations of our approach and outlines future work.

2 Background & Motivation

Before diving into the design of Prism, we begin with an overview of the GPU's compute and memory hierarchies and outline the challenges posed by reasoning about them collectively. We also discuss how Prism can help solve these problems.

```
 1  int tid = threadIdx.x;
 2  if (tid >= 0 && tid < 32){
 3    float A[4], B[2];
 4    float C[4] = { 0 };
 5   // Populate A with unique values.
 6    for (int i = 0; i < 4; i++)
 7       A[i] = tid * 4 + 1;
 8    // Populate B with unique values.
 9    for (int i = 0; i < 2; i++)
10       B[i] = tid * 4 + 1;
11    // Issue a warp-level Tensor Core
12    // operation: D = A * B + C
13    // (eliding some typecasts).
14   asm("mma.sync.aligned.m16n8k8..."
15     "{%0, %1, %2, %3},    "  /*D*/
16     "{%4, %5, %6, %7},    "  /*A*/
17     "{%8, %9},            "  /*B*/
18     "{%10, %11, %12, %13};" /*C*/
19     : "=r"(C[0]),"=r"(C[1]), ...
20     :  "r"(A[0]), "r"(A[1]), ...
21       "r"(B[0]), "r"(B[1]),
22       "r"(C[0]), "r"(C[1]), ...);}
```

Fig. 1. Warp-level Tensor Core instruction in CUDA.

```
tid : int @ thread[1] = id();
with group(thread[32]):
  A : float[4] @ thread[1]
  B : float[2] @ thread[1]
  C : float[4] @ thread[1]

  # Populate A with unique values
  for i in range(0, 4, 1):
    A[i] = tid * 4 + 1
    C[i] = 0
  # Populate B with unique values
  for i in range(0, 2, 1):
    B[i] = tid * 4 + 1

  # Issue Tensor Core op.
   intrinsic.mma(
    A[0], A[1], A[2], A[3],
    B[0], B[1],
    C[0], C[1], C[2], C[3],
    out=[A[0], A[1], A[2], A[3]])
```

Fig. 2. Warp-level Tensor Core instruction in Prism.

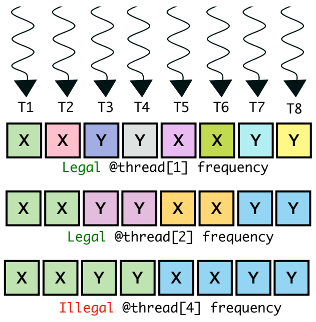
T1
T2
T3
T4
T5
T6
T7
T8
X X Y Y X X Y Y
Legal @thread[1] frequency
X X Y Y X X Y Y
Legal @thread[2] frequency
X X Y Y X X Y Y
Illegal @thread[4] frequency

Fig. 3. The **@** syntax represents the frequency at which a variable varies across threads **T1-T8**. Each **@** denotes a coloring of the variable; threads that "view" the variable with the same color must observe the same value.

2.1 Compute Hierarchy

In CUDA, programmers launch computations that run on thousands of threads. These threads are organized into a *compute hierarchy* that defines how work is distributed and scheduled on the GPU. At the top of this hierarchy is the *grid*, representing all threads launched as part of a single computation. The grid is divided into *blocks*, each containing a user-specified number of threads. *Threads* are the finest unit of execution and the machine's basic unit of sequential control.

Users describe a computation by writing a single program which is replicated across all threads. This program controls the behavior of individual threads by reading built-in identifiers like **threadIdx.x**—to determine a thread's position within a block—and **blockIdx.x**—to determine the block's position within the grid—at runtime. By making control flow decisions based on these two variables, users can assign each thread to its share of the full computation.

A natural and tempting way to interpret these built-in variables is to think of them as the indices of implicit "parallel-for loops" surrounding the program, where each iteration executes simultaneously. While this view is sufficient to understand CUDA's programming model when threads do independent work, it quickly breaks down in the presence of collective operations.

Unlike most instructions, which are executed by a single thread, collective operations must be executed collaboratively by a group of threads. For example, on line 14 of Figure 1, we perform a Tensor Core operation [43], which is only meaningful when invoked by a collection of 32 threads—a *warp*—acting together. For the call, each thread sets up its portion of the operands, and the operation is performed *once* for the entire warp, with the results scattered across participating threads. In this example, the first 32 threads in a block are tasked with this operation; programmers mentally group these 32 threads into a collective unit, and then reflect that grouping in the program via the **if**-statement on line 2. If the condition on line 2 were instead **tid >= 0 && tid < 30**, making fewer than 32 threads reach line 14, the result would be undefined. To make matters worse, the restriction is not only on the number of threads executing the operation, but also on their alignment. For threads to form a warp, the starting ID of the group must be aligned to a multiple of 32. So, if the condition on line 2 were instead **tid >= 1 && tid < 33**, the result would still be invalid, even though 32 threads would execute it.

Collective operations make reasoning about CUDA programs challenging because they force programmers to track the *convergence behavior* of threads. Specifically, programmers must reason both about how many threads reach each in the program and how those threads are arranged, accounting for alignment. This difficulty is further amplified by two factors. First, programs may have multiple points of convergence, requiring programmers to mentally track the relative ID of a thread within a logical group as that group evolves over the course of a program's execution. Second, threads may be participating in multiple levels of convergence within the same program. In our example, we only considered the warp-level Tensor Core operation, but there are other collective operations that require convergence at different granularities like *warpgroup*-level Tensor Core operations—which must be issued collectively by four warps—or the block-level `__syncthreads()` primitive—which must be executed by all threads within a block.

Reasoning about collective operations is already error prone within the context of a single function, but becomes even more difficult when reasoning across functions. A callee may assume a certain configuration of threads and may structure its computation around that assumption. However, such assumptions are not visible in the callee's function interface, which only exposes the input and output types. To invoke the function correctly, users must read its documentation—or worse, its implementation—to understand its assumptions, breaking modular reasoning.

In Prism, we materialize the programmer's mental grouping of the compute hierarchy explicitly in the program's source. Consider the example in Figure 2, which shows an equivalent rewrite of the CUDA program in Figure 1 using Prism. The call to the `mma` is valid only because it is executed within a `group` of 32 threads, made explicit on line 2. This intrinstic exposes its invariant via its function signature, which we will discuss in detail in Section 3.6. Since Prism's `group` construct enforces both the size and alignment of participating threads, the validity of the `mma` operation is guaranteed by construction.

However, it is not sufficient to only consider the compute hierarchy when reasoning about convergence; we must consider the memory hierarchy as well. Whenever threads see different data, they can use it to induce divergent behavior. We've already seen an example of this: the `tid` variable in Figure 1. In general, the possibility of divergence lurks anywhere threads branch on data.

2.2 Data and the Memory Hierarchy

Data on the GPU is organized into a *memory hierarchy*, mirroring the compute hierarchy. All threads in a grid can read and write from *global memory*, where operands reside at the start of a computation and where results are eventually written. Each block has access to a limited amount of *shared memory*, a programmer-managed scratchpad typically used to stage repeatedly-used data. Finally, each thread maintains its own state in *registers* and *local memory*. These are private to each thread, while shared and global memory are accessible by multiple threads.

Thread-Local Data. Registers and local memory are owned by individual threads, so variables with the same name can have different values on different threads at run time. In general, CUDA offers no support for dealing with the divergence that arises as a consequence.

A key challenge for Prism is managing this divergence, ensuring that all threads organized by other language constructs like `group` remain logically unified, even in the face of data-dependent control flow. To do this, Prism tracks the *frequency* with which values vary in space. The `@` syntax attached to each declaration denotes this frequency. For example, `thread[1]` variables can vary for every thread, while `block[1]` variables can vary for every block, but not for threads within that block. Intuitively, the `@` construct "colors" a variable across threads, and any threads that share a color are required to agree on that variable's value. Figure 3 illustrates valid and invalid colorings of a variable.

Using this information, Prism enforces rules for reading from and writing to variables. So, for example, if we introduce a condition based on `tid` *within* the `group`'s scope in Figure 2, Prism will reject that program; code that diverges at low frequency cannot read from variables that change at higher frequency. We explain the rules of Prism programs in detail in Section 3.

Thread-Shared Data. Shared and global memory, on the other hand, is not replicated per-thread; instead, all threads in a block have the same view into shared memory, while all threads on the grid have the same view into global memory.

CUDA does not explicitly model shared and global memory spaces, nor does it track whether allocations in a given memory space exceed device limits. Prism, on the other hand, distinguishes these spaces and restricts shared memory to static allocations, throwing an error if an allocation exceeds device limits. We will discuss these aspects in detail in Section 3.4. For now, however, we focus on how views of memory converge and diverge in tandem with the compute hierarchy.

As an example, when launching a kernel on the GPU, a global memory pointer initially belongs to the grid because every thread sees the same pointer value at the start of the computation. To write to that memory, each thread computes an offset from the original base address. In doing so, the pointer effectively diverges across threads so each can write independently. After these writes, the view on the memory must reconverge, ensuring that all threads have completed their updates before it can return to its original logical owner.

```
 1  int x = threadIdx.x;
 2  if (x >= 0 && x< 32){
 3    float A[4], B[2];
 4    float C[4] = { 0 };
 5
 6    // k is a_mem's stride.
 7    A[0] = a_mem[(x/4)*k+(x%4)]);
 8    A[1] = a_mem[(x/4)*k+(x%4)+4]);
 9    A[2] = a_mem[(x/4+8)*k+(x%4)]);
10    A[3] = a_mem[(x/4+8)*k+(x%4)+4]);
11
12    // n is b_mem's stride.
13    B[0] = b_mem[(x%4)*n+(x/4)]);
14    B[1] = b_mem[(x%4+4)*n+(x/4)]);
15
16    asm("mma.sync...");
17
18    // Write back into c_mem.
19    // n is c_mem's stride.
20    c_mem[(x/4)*n+2*(x%4)] = C[0];
21    c_mem[(x/4)*n+2*(x%4)+1] = C[1];
22    c_mem[(x/4+8)*n+2*(x%4)] = C[2];
23    c_mem[(x/4+8)*n+2*(x%4)+1]=C[3];
24
25    __syncwarp();}
```

Fig. 4. Invoking a warp-level Tensor Core instruction in CUDA after loading data from memory.

Let us reconsider the Tensor Core example from Figure 1, this time initializing the operands of the Tensor Core operation from pointers into global memory. In Figure 4, **A** and **B** are now populated from **a_mem** and **b_mem**. After the Tensor Core operation completes, the result is written back to **c_mem**, also in global memory. This variable **c_mem** is logically accessed from two different levels of the compute hierarchy. At the beginning, each thread in a group of 32 sees the same value for **c_mem**. Then, they each locate an offset within **c_mem** that they write to (lines 20-23). To restore **c_mem** back to its 32 thread-level ownership, all 32 threads must synchronize (line 25) to ensure all per-thread writes have completed. This is similar to the requirement we saw documented in CUB in Section 1.

Similarly to the compute hierarchy, CUDA programmers are responsible for tracking the logical owner of a view of memory as it evolves over the course of a program, and for ensuring that appropriate synchronization occurs.

```
 1  @prism("device")
 2  @requires(thread[32])
 3  def simple_mma(
 4      a: ptr(const(float)) @ thread[32],
 5      b: ptr(const(float)) @ thread[32],
 6      c: ptr(float) @ thread[32]):
 7    x : int @ thread[1] = id()
 8    with group(thread[32]):
 9      A : float[4] @ thread[1]
10      B : float[2] @ thread[1]
11      C : float[4] @ thread[1]
12      # Reads do not need to be lowered.
13      A[0] = a_mem[(x/4)*k+(x%4)]);
14      A[1] = a_mem[(x/4)*k+(x%4)+4]);
15      A[2] = a_mem[(x/4+8)*k+(x%4)]);
16      A[3] = a_mem[(x/4+8)*k+(x%4)+4]);
17      B[0] = b_mem[(x%4)*n+(x/4)]);
18      B[1] = b_mem[(x%4+4)*n+(x/4)]);
19      # Skipping initialize C to 0...
20      intrinsic.mma(
21        A[0], A[1], A[2], A[3],
22        B[0], B[1],
23        C[0], C[1], C[2], C[3],
24        out=[C[0], C[1], C[2], C[3]])
25      # Must be at thread[1] to write.
26      idx = \
27        lambda ro, co: (x/4+ro)*n+2*(x%4)+co
28      with
     ↪   partition(c_mem,p=thread[32],f=idx)
     ↪   as c_thrd:
29        c_thrd[0, 0] = C[0]
30        c_thrd[0, 1] = C[1]
31        c_thrd[8, 0] = C[2]
32        c_thrd[8, 1)] = C[3]
33      # --- Sync point ---
```

Fig. 5. Invoking a warp-level Tensor Core instruction in Prism after loading data from memory.

Prism, by contrast, makes the evolution of memory's view explicit in the syntax and automatically inserts the synchronization required to restore that view to its original owner. In Figure 5, we show how memory is lowered through the compute hierarchy for the same Tensor Core operation from Figure 4. The lowering of **c_mem** is required in this program as Prism only permits writes from the view of a single thread. To lower **c_mem**, we use Prism's **partition** operation on line 28. The operation takes a pointer to partition, **c_mem**, and lowers it to a single thread, assigning it a new name, **c_thrd**; Prism will not allow use of the old variable **c_mem** within the **partition**'s scope. The **partition** also takes an indexing function, and each time **c_thrd** is accessed, this function is implicitly applied. Once the **partition**'s scope ends, Prism inserts a synchronization barrier before the next use of the original variable, so that all per-thread writes have completed. In this way, at the end of the **partition**, the original variable represents a convergent view of the data once again.

```
 1  @prism("global")
 2  # Top-level perspective bounds and shared memory usage.
 3  @requires(grid[1], block[1], thread[32], smem=1280)
 4  def mmaTF32NaiveKernel(A: ptr(const(float)) @ grid[1],
 5                         B: ptr(const(float)) @ grid[1],
 6                         C: ptr(float) @ grid[1],
 7                         M : int @ grid[1],
 8                         N : int @ grid[1],
 9                         K : int @ grid[1]):
10
11    # Starts out with grid[1] perspective.
12    with group(grid[1]):
13      # @ grid[1] inferred from current perspective
14      # Each block computes an 16 x 8 tile
15      num_blocks_n : const(int) = (N + 8 - 1) / 8
16
17      # id() function returns the block id
18      # inferred from @ block[1].
19      blk_row : const(int) @ block[1] = (id()/num_blocks_n)*16
20      blk_col : const(int) @ block[1] = (id()%num_blocks_n)*8
21
22      # Give each block an offset into C
23      offset = lambda x: blk_row * N + blk_col + x
24      with partition(C, p=block[1], f=offset) as C_blk:
25        with group(block[1]):
26          # SHMEM declarations are only allowed
27          # with a block[1] perspective.
28          A_smem : shared(float[16 * 8]) @ block[1]
29          B_smem : shared(float[8 * 8])  @ block[1]
30          C_smem : shared(float[16 * 8]) @ block[1]
31          # Now, id() returns the thread id
32          idx = lambda x: x * 4
33          # To write to C_smem, drop to thread[1] perspective
34          with partition(C_smem,p=thread[1],f=idx) as C_th:
35             for i in range(0, 4, 1):
36                with group(thread[1]):
37                  C_th[i] = 0
38        for i in range(0, K_tiles, 1):
39
40          # --- Sync point --- (backedge from for loop)
41          for j in range(0, 4, 1):
42            global_row : int @ thread[1] = blk_row + row
43            global_col: int @ thread[1] = i * 8 + col
44            with partition(A_smem, p=thread[1], f=..) as A_th:
45              with group(thread[1]):
46                A_thrd[0] = A[global_row * K + global_col]
47
48          # --- Sync point --- (backedge from for loop)
49          for j in range(0, 2, 1):
50            # Similar to write into A_smem ...
51            with partition(B_smem,p=thread[1],f=...) as B_th:
52              with group(thread[1]):
53                B_th[0] = B[global_row_b * N + global_col_b]
54
55            # Give each warp an offset into C_smem.
56            # --- Sync point --- (backedge from for loop)
57            with claim(C_smem, p=thread[32]) as Cs_warp:
58                match split(thread): # Masks off other threads
59                  case 32:
60                    # Call function that performs a
61                    # Tensor Core instruction.
62                    simple_mma(A_smem, B_smem, Cs_warp)
63            # --- Sync point ---
64
65        for j in range(0, 4, 1):
66          flat_idx_c : int @ thread[1] = id() * 4 + j
67          row_c : int @ thread[1] = flat_idx_c / MMA_K
68          col_c : int @ thread[1] = flat_idx_c % MMA_K
69          idx = lambda x: row_c * N + col_c + x
70          with partition(C_blk, p=thread[1], f=idx) as C_th:
71            with group(thread[1]):
72              C_th[0]  = C_smem[row_c * MMA_N + col_c]
73
74  return
```

Fig. 6. Naive tensor float 32 matrix multiplication in Prism (full program can be found in Appendix B.1).

3 The Prism Language

Prism is an imperative, low-level language designed at a level of abstraction comparable to that of CUDA. Unlike CUDA, however, Prism's syntax materializes the mapping of computations onto the compute and memory hierarchy explicitly in the program source. Using this information, Prism enforces that programs only execute collective operations with sufficient resources.

To guide our discussion about Prism's design, we use the program in Figure 6 as a running example. It computes a matrix multiplication between two float arrays, **A** and **B**, to produce an output matrix **C**. In this program, each block computes an independent 16×8 tile of the output. To do so, blocks first locate the tile index assigned to them (line 19-20). Next, threads in each block load corresponding rows and columns from **A** and **B** (line 41-53) into shared memory. Finally, the program invokes a warp-level, Tensor Core instruction to compute the output (line 62), requiring threads in a warp to converge. We encapsulate this Tensor Core instruction in a function, demonstrating how function composition works in Prism. This function is the same as the one in Figure 5.

3.1 Levels

Prism models the machine's compute hierarchy through *levels*. There are three levels in Prism—**grid**, **block** and **thread**—which are organized as expected: a **grid** consists of multiple **block**s, each of which consist of multiple **thread**s. Levels are ordered, with **thread** < **block** < **grid**.

There are two key differences between Prism's levels and those in CUDA. First, Prism does not model CUDA's three-dimensional grid or block structure. Second, there are two commonly-used "levels", warp and warpgroup, absent from our hierarchy. On the hardware, the units of each level are arranged in a linear order, and the three-dimensional structures of CUDA are simply interpretations of this ordering, not distinct hardware resources. Similarly, warps and warpgroups are organizational constructs defined in terms of existing levels. Namely, a warp is a group of 32 threads whose first thread ID is aligned to 32. A warpgroup, which was introduced with the release of the Hopper architecture [8], consists of 4 consecutive warps.

Rather than baking these interpretations into Prism by adding new dimensions and levels, we let users express multi-dimensional structures and define groupings of custom sizes.

3.2 Perspectives

Perspectives are the central concept of Prism, representing the view of the hierarchy from which the programmer is defining the machine's behavior. Perspectives allow Prism to determine which compute resources the programmer is controlling, whether they are available in the program's context, and if those resources are sufficient for a given operation.

A perspective is a level—**`grid`**, **`block`**, or **`thread`**—paired with a static constant **`n`**, specifying the number of units at that level. For example, **`thread[2]`** denotes a perspective of two threads, **`block[4]`** denotes a perspective of four blocks, and so on. Perspectives also carry *alignment information*: a perspective of size **`n`** is aligned to **`n`**. In this way, a warp is simply a desugaring of **`thread[32]`**, and a warpgroup is a desugaring of **`thread[128]`**.

Finally, perspectives are partially ordered. We say that $\texttt{level}_2[\texttt{n}_2]$ is *broader* than $\texttt{level}_1[\texttt{n}_1]$, or that $\texttt{level}_1[\texttt{n}_1]$ is *narrower* than $\texttt{level}_2[\texttt{n}_2]$, if either:

(1) $\texttt{level}_1 < \texttt{level}_2$; or,
(2) $\texttt{level}_1 = \texttt{level}_2$ and $\texttt{n}_1$ divides $\texttt{n}_2$.

3.3 Perspectives on Code

Code is associated with a set of perspectives, called a *perspective bound*, which corresponds to the compute resources whose behavior it defines. At every point in the program, the perspective bound for that point indicates which layer of the hierarchy is being programmed, and how that layer can be destructed into narrower perspectives. For example, a line of code with a **`{block[1], thread[4]}`** perspective bound tells Prism that the current line of code is being programmed at the **`block[1]`** perspective and that the **`block[1]`** perspective can be destructed into a some number of **`thread[4]`** perspectives. As a shorthand, when we refer to a code's perspective, we mean the broadest perspective available in its perspective bound (in this example, **`block[1]`**).

Functions begin with a top-level perspective bound. In Figure 6, the perspective bound is defined on line 2, using the notation $\texttt{@requires(grid[n}_1\texttt{],block[n}_2\texttt{],thread[n}_3\texttt{])}$. Programmers can then shape the program's current perspective using two constructs: **`group`** and **`split`**.

Group. The **`group`** construct lets programmers shift from a broader perspective to some number of narrower perspectives contained in it. Operationally, the **`group`** construct does this by replicating code written from the narrower perspective across the broader one. In effect, **`group`** forks many copies of a narrower perspective.

Let's consider this in context of our example. In Figure 6, execution begins at **`grid[1]`** on line 12. At that point, a programmer controls the whole grid's behavior. To produce different output tiles, the programmer shifts their perspective to **`block[1]`** on line 25. The code within the **`group(block[1])`** defines the behavior of a single block, and is replicated across all blocks in the grid.

```
1  # Example 1
2  with group(thread[2]):
3      # Illegal because block > thread.
4      with group(block[1]):
5          pass
6  # Example 2
7  with group(block[6]):
8      # Illegal because 6 % 5 != 0.
9      with group([block[5]]):
10         pass
```

Fig. 7. Illegal uses of **`group`**.

Not all uses of **`group`** are valid: the examples in Figure 7 have no meaning on the hardware. On line 4, the program tries to broaden its perspective to **`block[1]`** from **`thread[2]`**, which is illegal. On the other hand, in the second example, the program tries to narrow its perspective from **`block[6]`** to **`block[5]`**. While 5 is indeed less than 6, Prism cannot evenly replicate **`block[5]`** inside a **`block[6]`** perspective, and so rejects this program. Recall, from Section 3.2, that whether one perspective is narrower than another is dependent on divisibility, not just size.

To eliminate such cases, Prism only allows an invocation of **group(level[n])** if the current perspective bound contains a perspective broader than **level[n]**. Once **group(level[n])** is invoked, it modifies the current perspective bound in two ways. First, it removes all perspectives broader than **level[n]** from it. Second, it sets the broadest perspective within the **group** to be **level[n]**.

Split. Unlike **group**, which is used for replication into equally-sized, narrower perspectives, **split** is used for sharding the current perspective unequally. For example, Figure 8 shows a **split** from **thread[4]** into one branch with **thread[2]** perspective and two with **thread[1]**. The three arms of the **split** execute independently in parallel[1], as a form of unordered composition. When **split(level)** is invoked, the perspectives of its branches diverge. At the end of the **split**, they reconverge, and continue execution with the original perspective.

```
with group(thread[4]):
  match split(thread):
    case 2:
      ...
    case 1:
      ...
    case 1:
      ...
```

Fig. 8. Example **split**.

Use of **split** is necessary to write warp-specialized [4] code, a programming pattern used in high-performance kernels. Another important use of **split**—masking off threads—can be found in our running example. Line 58 in Figure 6 shows a **split** that requests the first warp in the block, narrowing from **block[1]** into a single **thread[32]**. This warp is later used to execute a Tensor Core operation.

```
with group(thread[4]):
  match split(thread):
    case 4:
      ...
    case 1:
      ...
```

Fig. 9. Illegal split exceeds the available perspectives.

Because **split** corresponds to unordered composition, it must provide each of its branches their requested perspectives simultaneously. Prism thus checks that the sum of the perspectives requested by all branches of the **split** can be satisfied. For example, the program in Figure 9 does not type check. Finally, because perspectives enforce alignment, every branch of the **split** must also be aligned; not all **split**s whose sizes are at most the available units are valid. Figure 10 shows an example violating this constraint: the second branch of the split is not aligned to 2.

```
with group(thread[3]):
  match split(thread):
    case 1:
      ...
    case 2:
      ...
```

Fig. 10. Illegal split violates alignment.

Once **split** is invoked, for each branch that requests **n** units, all perspectives broader than **level[n]** are removed from its perspective bound, and the available units for **level** are set to **n**.

3.4 Perspectives on Data

Section 3.2 described how programmers can control different layers of the hierarchy by changing their perspectives on code through **group** and **split**. To ensure these operations remain meaningful, Prism must ensure that threads inside a perspective remain logically grouped, even when they encounter **for**, **while**, and **if** statements. As we saw in Section 2.2, making this guarantee requires Prism to track how data varies across threads.

3.4.1 Thread-Local Data. In Prism, each local variable has a perspective, which indicates the frequency at which its values change in space. This frequency remains constant for the duration of a program, and it tells Prism that a **level[n]** variable is always indistinguishable to threads within that perspective. For example, **blk_row @ block[1]** on line 19 in Figure 6 has the same value across all threads in a block.

Programmers specify the perspective that each variable lives at in its declaration. A variable **v** of type **int** is declared at **thread[1]** perspective using the syntax **v : int @ thread[1]**.[2] To enforce this frequency invariant, Prism restricts reads from and writes to variables based on their perspectives. The rule can be summarized as follows: Prism allows programs to "read up" from broader perspectives and "write down" to narrower ones.

[1]Despite the use of the **match** syntax, *all* branches of the **split** execute.

[2]If not explicitly annotated, Prism infers a variable's perspective to be the perspective of the code where it was declared.

Read Up. Variables can only be read if their perspective is at least as broad as the current code perspective. Figure 11 gives an example of an illegal read that would violate this constraint. While **`__syncthreads()`** should always be safe in **`block[1]`**, branching on the variable **`flag`**—which may take different values across threads in a block—can cause only some of those threads to reach the **`__syncthreads()`**, violating its collective invariant.

```
flag : bool @ thread[1] = ...
with group(block[1]):
  if (flag)
    __syncthreads();
```

Fig. 11. Illegal read of **`thread[1]`** variable.

Write Down. Writes are dually constrained. Only values that live at broader perspectives can be written into variables that live at narrower ones. For example, a **`block[1]`** variable can be used to write into a **`thread[1]`** variable, but not vice versa. An example of an illegal write is shown in Figure 12, where writing from a **`thread[1]`** variable into a **`block[1]`** one would cause a deadlock.

```
x : bool @ thread[1] = ...
y : bool @ block[1] = ...
with group(block[1]):
  y = x
  if(y):
    __syncthreads();
```

Fig. 12. Illegal write of **`thread[1]`** variable into a **`block[1]`** variable.

Together, the "read up" and "write down" rules ensure that information only flows from broader perspectives to narrower ones. In Figure 6, we can see the "read up" rule in action on lines 42 and 43. Meanwhile, lines 19 and 20 are instances of the "write down" rule.

3.4.2 Thread-Shared Data. Unlike thread-local data, which is literally replicated across threads and is backed by distinct physical storage, thread-shared data consists of pointers into shared and global memory that are visible to some collection of threads. As a result, the perspective such data inhabits can evolve as the program executes.

In Prism, there are three mechanisms for obtaining thread-shared data. The first is to directly allocate data residing in shared memory.[3] The second and third are to obtain offsets into existing data by using the **`partition`** or **`claim`** operations. These constructs mirror **`group`** and **`split`**, and are used to temporarily narrow the perspective associated with a pointer.

Prism simplifies shared memory allocations by requiring them to have a static size. The **`@requires`** annotation specifies the amount of shared memory a function expects to have. Prism uses this information to ensure that a function's allocations do not exceed its declared limit, and checks whether there is enough shared memory available at its call sites. The function in Figure 6 declares the amount of shared memory it will require on 2. We use standard techniques [34] to statically bound memory usage with Prism's type system.

Additionally, since shared memory is only visible to threads within the same block, such allocations are only permitted at **`block[1]`**. An example of such an allocation is shown on lines 28-30 of Figure 6. During compilation, Prism will automatically handle the necessary pointer arithmetic to assign each allocation an appropriate offset within the shared memory space.

Partition. The **`partition`** operation plays the same role for memory that **`group`** plays for code. It is used to refine the perspective of a pointer by computing offsets for each narrowed perspective. Concretely, the **`partition`** operation takes a pointer variable **`x`**, an indexing function **`f`**, and produces a new variable **`y`** at a narrower perspective **`level[n]`** to be used inside the **`partition`**. After the partition ends, the threads participating in the **`partition`** must synchronize to restore the original pointer **`x`** to its perspective.[4]

Inside the scope of the partition, the original variable **`x`** cannot be used, and the pointer can only be accessed using **`y`**. Each use of **`y`** applies the indexing function **`f`** to compute the true offset of the

[3]Prism assumes that all global memory allocations have been made before launching the CUDA kernel, as is typically the case with CUDA programs.

[4]Strictly speaking, this synchronization is only required at the next point the memory is to be *re-used*. Prism has a mechanism for optimizing the placement of these synchronization points, which is describe in Section 5.

access. For example, the use of **partition** on line 34 in Figure 6 takes a pointer **C_smem** at **block[1]**, gives it a new name **C_th**, and distributes it across **thread[1]** perspectives by transforming each occurrence of **C_th[i]** in the body into **C_smem[i*4]**.

At this point, it is necessary to consider how different indexing functions affect the possibility of data races. If the indexing function is injective, each narrower perspective receives a distinct offset into the underlying array, and the resulting partition is free of data races. When the indexing function is not injective, multiple threads may race on the same location, introducing a potential data race.

Preventing data races is not one of Prism's goals. In Prism, data races *can* occur within a **partition**; however, since the data is eventually synchronized before it is reused, the last writer wins. Prior work, in particular Descend [35], describes a type system for data-race free GPU programs, and we believe a similar approach can be combined with Prism's. Prism instead focuses on the interaction between the compute and memory hierarchies and on reasoning about them simultaneously to ensure that operations are executed only with sufficient compute resources, a guarantee that Descend cannot provide.

Out-of-bounds accesses are undefined behavior.

Claim. The **claim** operation lets programmers narrow a pointer's perspective by giving it to only one collection of threads with that narrower perspective. For example, line 57 of Figure 6 shows an example of a **claim**, where the pointer **C_smem @ block[1]** is only available to a single warp in the block and narrowed to **Cs_warp @ thread[32]** to call a Tensor Core operation.

A **claim** takes the original pointer **x**, the target perspective to narrow it to, and a new name **y** to assign to the narrowed memory. In all **split**s within the **claim**, the new pointer **y** is accessible only within one branch; sibling branches are not permitted to read or write from this memory. As with **partition**, the original variable **x** is not accessible within the **claim**.

3.5 The id() Function

As opposed to exposing users to special hardware variables like **blockIdx.x**, and **threadIdx.x**, Prism provides an **id()** function instead. The **id** function returns the relative index of a narrow perspective within a broader perspective. That is, the interpretation of the **id** function depends on both the perspective of the variable it is being written to and the perspective of the code invoking it. In Figure 6, we use a call to **id** with **grid[1]** perspective on lines 19 and 20 to locate the tile that each block is in charge of computing. On line 32, however, a write of **id** into a **thread[1]** variable at **block[1]** perspective will return the relative ID of the thread within the block, not in the grid.

3.6 Collective Operations

There are two types of collective operations in Prism.

Function Calls. Each function carries a *perspective signature*, which consists of its top-level perspective bound, shared memory usage, and its arguments' perspectives. An example of such a signature can be seen in lines 3-9 in Figure 6. At call sites, Prism ensures that the callee's perspective signature can be satisfied by the caller. Since the perspective bound describes a minimum requirement, functions can be called with a broader code perspective than necessary, accounting for alignment. When checking function arguments at call sites, Prism distinguishes between primitive data types and pointers. For primitive data types, like **int** or **bool**, arguments can have broader perspectives than specified in the function signature. On the other hand, pointers that live at broader perspectives can only be passed if the pointer is marked as **const**, indicating that it will only be used for reading. Otherwise, we require that the pointer's perspective exactly match that of the function's perspective signature.

Intrinsics. The compiler provides a pre-defined set of collective intrinsics –like the Tensor Core instruction discussed in Section 2.1—each of which declares its perspective signature. Programmers may also add to this set using **unsafe**. From within **unsafe** code, programmers can inline assembly instructions and wrap them in a Prism function, specifying its perspective signature. Prism checks these call sites like those of any other function.

3.7 Asynchrony

Asynchronous data movement works similarly to other memory operations. Users can mark storage as asynchronous with the **async** construct. As with **partition** and **claim**, **async** hides the old variable and exposes a fresh one that is only accessible within the **async** statement. Inside, Prism ensures the new variable is only used by async data-movement intrinsics.

Prism includes two such intrinsics: bulk [16] and non-bulk [19] asynchronous data-movement instructions. We show an example of the former in Figure 13. As with the data operations in Section 3.4, Prism inserts the necessary synchronization before the program's next use of the original variable, ensuring that the asynchronous transfer has completed.

```
with async(old_name) as new_name:
  match split(thread):
    case 1:
      cp_async(smem, new_name, 16);
```

Fig. 13. Example of an **cp.async** instruction in Prism.

4 Formalization in Bundl

Having introduced the full Prism language, we now describe Bundl, a core calculus that formalizes its most fundamental aspects by statically tracking perspectives on code and data. We use Bundl to argue that well-typed Prism programs are not only type-safe, but will also never execute operations for which they lack the correct perspective.

In this section, we describe Bundl's type system and operational semantics—in particular how it manages compute and data perspectives—and build up to a formal proof of type-and-perspective safety. We instrument Bundl's operational semantics with runtime perspective enforcement: its rules will get stuck if they encounter an operation for which they have the wrong perspective. This runtime enforcement means that our safety theorem guarantees that dynamically-realized perspectives match the ones inferred by the type system.

4.1 Bundl Type System

The core idea in Bundl is to track, at the type level, the program's perspective on code and data. To achieve this, we borrow techniques from the literature on dependency tracking [1]. In particular, the code perspective is tracked on the typing judgment, which has the form $\Gamma \vdash^{\pi} e : \tau$ for expressions and $\Gamma \vdash^{\pi} s$ for statements. The π over the $\vdash$ is the code perspective on e and s, and is comprised of a level and a size, the same structure as a perspective in Prism.

The typing context also tracks the perspective at which each variable lives; data can only be read from or written to a variable when its perspective is compatible with that of the code interacting with it. This requirement is made manifest in the T-Var rule, found in Figure 14. Observe that the π in the variable rule must match exactly between data and code; principles like "read up" and "write down" are instead encoded directly in the rules for reading and writing, like T-Arr-Access, which views the array being read with broader perspective than the current code perspective.

Figure 14 also shows other key rules, which fall into two main categories: those for managing perspectives on code and those for perspectives on data.

4.1.1 *Managing Perspectives on Code.*

Bundl's **group** statement directly corresponds to Prism's, and is checked by the T-Group rule. Given some statement s that checks with perspective π, the statement **group** q s will check with $q \cdot \pi$, enforcing Prism's divisibility requirement.

$\boxed{\Gamma \vdash^{\pi} e : \tau}$ *(Expression typing)*

$$\frac{x :^{\pi} \tau \in \Gamma}{\Gamma \vdash^{\pi} x : \tau}\ \textsc{T-Var} \qquad \frac{\Gamma \vdash^{\pi'} e_1 : \tau[]^{l} \quad \Gamma \vdash^{\pi} e_2 : \texttt{int} \quad \pi \le \pi'}{\Gamma \vdash^{\pi} e_1[e_2] : \tau}\ \textsc{T-Arr-Access}$$

$\boxed{\Gamma \vdash^{\pi} s}$ *(Statement typing)*

$$\frac{\Gamma \vdash^{(h,n_1)} s_1 \quad \Gamma \vdash^{(h,n_2)} s_2 \quad n_1,n_2\ \textbf{align to}\ n}{\Gamma \vdash^{(h,n)} \texttt{split}(n_1,n_2)\{s_1\}\{s_2\}}\ \textsc{T-Split}$$

$$\text{where } n_1,n_2\ \textbf{align to}\ n ::= (n_1+n_2 \le n)\ \textit{and}\ (n_1 | n)\ \textit{and}\ (n_2 | n)\ \textit{and}\ (n_2 | n_1+n)$$

$$\frac{\Gamma \vdash^{\pi} s}{\Gamma \vdash^{q \cdot \pi} \texttt{group}\ q\ s}\ \textsc{T-Group} \qquad \frac{\Gamma, y :^{\downarrow \pi} \tau[]^{l} \vdash^{\pi} s \quad l \ne \texttt{Local}}{\Gamma, x :^{\pi} \tau[]^{l} \vdash^{\pi} \texttt{lower}\ x\ \texttt{into}\ y\ \texttt{in}\ s}\ \textsc{T-Lower}$$

$$\frac{\Gamma, y :^{(h,n/c)} \tau[]^{l} \vdash^{(h,n)} s \quad c | n \quad l \ne \texttt{Local}}{\Gamma, x :^{(h,n)} \tau[]^{l} \vdash^{(h,n)} \texttt{partition}\ x\ \texttt{into}\ y\ \texttt{by}\ c\ \texttt{in}\ s}\ \textsc{T-Partition} \qquad \frac{\Gamma \vdash^{\downarrow \pi} s}{\Gamma \vdash^{\pi} \texttt{destruct in}\ s}\ \textsc{T-Destruct}$$

$$\frac{\Gamma, y :^{(h,n')} \tau[]^{l} \vdash^{(h,n')} s \quad n' \le n \quad l \ne \texttt{Local}}{\Gamma, x :^{(h,n)} \tau[]^{l} \vdash^{(h,n)} \texttt{claim}\ x\ \texttt{into}\ y\ \texttt{at}\ n'\ \texttt{in}\ s}\ \textsc{T-Claim}$$

Fig. 14. Core typing rules of Bundl. The typing rules presented here are a simplified selection of the full rules, which can be found in Appendix A.2.

The **split** statement, meanwhile, is checked by the T-Split rule, and functions like a binary version of the n-ary **split** construct in Prism. It enforces the same divisibility requirements to ensure that the perspectives on code and data remain properly aligned, and then checks the two sub-statements s_1 and s_2 with the divided, narrower perspectives.

In Bundl, to better model the details of how perspectives shift down the GPU hierarchy, we introduce a third construct called **destruct** that makes explicit exactly where such shifts occur. In the corresponding rule, T-Destruct, the $\downarrow$ operation on πs "destructs" the perspective into many narrower perspectives at a lower level. This operation is defined as $\downarrow(\texttt{Grid}, 1) = (\texttt{Block}, B)$ and $\downarrow(\texttt{Block}, 1) = (\texttt{Thread}, T)$, where B and T are parameters to a particular instantiation of Bundl to describe the number of blocks per grid and threads per block.[5]

4.1.2 Managing Perspectives on Data. The mechanism for managing data perspective mirrors that of code perspective, with each operation for data corresponding to an operation for code.

The **partition** operation is analogous to **group**ing a code perspective. The typing rule for this operation, T-Partition, requires that the data perspective on x, the variable to be partitioned, is the same as the current perspective on code. After partitioning, a fresh variable y is introduced with a new perspective π/c. Within the **partition**, we disallow references to x and continue checking the body with the original π; the **partition** has no effect on the code perspective.

Unlike **partition**, which divides up a piece of data equally among narrower perspectives, the **claim** operation views the claimed data with exactly one narrower perspective. Accordingly, Bundl needs to ensure that only one branch of a **split** operation, with the appropriate π, can refer to the claimed variable. To ensure that this is the case, the T-Claim rule links the data perspective of the variable to the compute perspective of the code claiming it by changing both at the same time. This represents a minor difference from Prism, which uses additional static analysis to ensure that a claimed variable is only accessed in a single **split** branch.

[5]Bundl is abstracted over these B and T values, so instead of tracking perspective bounds the way that Prism does, it only tracks the top-level perspective described in Section 3.2.

$\boxed{L,S,\Sigma,P \rightsquigarrow L',S',\Sigma',P'}$ *(Machine judgment)*

$$\frac{L(t),S(b),\Sigma,t,b,0 \vdash^{(\texttt{Grid},1)} s \rightsquigarrow s' \dashv \eta',\sigma',\Sigma' \qquad P(t,b)=s}{L,S,\Sigma,P \rightsquigarrow L[t\mapsto\eta'],S[b\mapsto\sigma'],\Sigma',P[(t,b)\mapsto s']}\ \textsc{S-Program}$$

$\boxed{\eta,\sigma,\Sigma,t,b,p \vdash^{\pi} s \rightsquigarrow s' \dashv \eta',\sigma',\Sigma'}$ *(Thread judgment)*

$$\frac{p<n_1 \qquad n_1,n_2\ \textbf{align to}\ n \qquad \eta,\sigma,\Sigma,t,b,p,\vdash^{(h,n_1)} s_1 \rightsquigarrow s_1' \dashv \eta',\sigma',\Sigma'}{\eta,\sigma,\Sigma,t,b,p \vdash^{(h,n)} \texttt{split}(n_1,n_2)\{s_1\}\{s_2\} \rightsquigarrow \texttt{split}(n_1,n_2)\{s_1'\}\{s_2\} \dashv \eta',\sigma',\Sigma'}\ \textsc{S-Split-Left}$$

$$\frac{p\geq n_1 \qquad p<n_1+n_2 \qquad n_1,n_2\ \textbf{align to}\ n \qquad \eta,\sigma,\Sigma,t,b,p-n_1,\vdash^{(h,n_2)} s_2 \rightsquigarrow s_2' \dashv \eta',\sigma',\Sigma'}{\eta,\sigma,\Sigma,t,b,p \vdash^{(h,n)} \texttt{split}(n_1,n_2)\{s_1\}\{s_2\} \rightsquigarrow \texttt{split}(n_1,n_2)\{s_1\}\{s_2'\} \dashv \eta',\sigma',\Sigma'}\ \textsc{S-Split-Right}$$

$$\frac{\eta,\sigma,\Sigma,t,b,p\ \textbf{mod}\ n \vdash^{(h,n)} s \rightsquigarrow s' \dashv \eta',\sigma',\Sigma'}{\eta,\sigma,\Sigma,t,b,p \vdash^{(h,q\cdot n)} \texttt{group}\ q\ s \rightsquigarrow \texttt{group}\ q\ s';\dashv \eta',\sigma',\Sigma'}\ \textsc{S-Group}$$

$$\frac{\eta,\sigma,\Sigma,t,b,t\ \textbf{mod}\ T \vdash^{(\texttt{Thread},T)} s \rightsquigarrow s' \dashv \eta',\sigma',\Sigma'}{\eta,\sigma,\Sigma,t,b,0 \vdash^{(\texttt{Block},1)} \texttt{destruct in}\ s \rightsquigarrow \texttt{destruct in}\ s' \dashv \eta',\sigma',\Sigma'}\ \textsc{S-Destruct-Block}$$

$$\frac{}{\eta,\sigma,\Sigma,t,b,p \vdash^{(\texttt{Block},1)} x := \texttt{alloc Shared}\ \tau\ n\ \texttt{in}\ s \rightsquigarrow s \dashv \eta,\sigma[x \mapsto^{\pi} \langle x,n\rangle],\Sigma}\ \textsc{S-Alloc-Shared}$$

Fig. 15. Core semantic rules of Bundl. As with the typing rules, we present only a simplified selection of the full rules, which can be found in Appendix A.3.

Lastly, the T-Lower rule mirrors the T-Destruct rule; it uses the $\downarrow$ operator to move a variable from one level of the hierarchy to another, distributing it equally among all the narrower perspectives at that level in the same manner as T-Partition.

Note that these rules only apply to thread-shared memory, i.e., arrays that do not live in **`Local`**. The provenance of an array type is denoted by the superscript l above it.

4.2 Bundl Semantics

Having explained the key rules of the type system, we can move on to discuss Bundl's operational semantics. To reflect the fact that a GPU program executes in parallel across numerous threads, we model the semantics of Bundl in the style of Turon et al. [57], using a two-level small step judgment. We present the key rules of this semantics in Figure 15.

The top level (i.e., machine-level) judgment has just one rule: S-Program. This rule acts as a "frame" for the lower level (i.e., thread-level) judgment, and steps a collection of thread-ID-indexed local memories (L), a collection of block-ID-indexed shared memories (S), a global memory (Σ), and a thread pool (P) to an updated collection of memories and updated thread pool. The thread pool maps thread and block IDs to code, intuitively representing the program being executed by each thread at the current moment. The S-Program rule non-deterministically chooses a thread ID and block ID and steps it according to the thread-level judgment. This allows the semantics to model the full range of non-deterministic behavior arising from the GPU's thread scheduler.[6]

[6]In reality, the GPU's warp scheduler issues instructions to threads in a warp in lockstep, but modeling every thread as completely independent is both simpler and a conservative overestimate of the nondeterministic behavior of the GPU.

The thread-level judgment has the shape $\eta,\sigma,\Sigma,t,b,p \vdash^{\pi} s \rightsquigarrow s' \dashv \eta',\sigma',\Sigma'$, where

- η denotes thread-local memory,
- σ denotes shared memory,
- Σ denotes global memory,
- t denotes the thread's ID,
- b denotes the ID of the block in which the thread lives, and
- p denotes the relative position of the thread within π (the perspective ID).

Critically, notice that a π also appears on the thread-level judgment just as it does on the typing judgment. This is because the thread semantics *dynamically tracks and enforces perspectives.* The same way evaluation of a program "gets stuck" if a value does not have the right type, the semantics of Bundl also get stuck if code attempts to access data or invoke commands with the wrong perspective. As an example, observe the S-ALLOC-SHARED rule in Figure 15, which requires a (**`Block`**,1) perspective and will fail to step if encountered with a different one. This runtime perspective is present in Bundl, but is erased by Prism during compilation; in Section 4.3 we use it to prove that well-typed programs will always execute with the same perspective that the type system viewed them with.

The semantic rules for perspectives involve manipulating p to track which threads take which code paths when perspectives are **`split`** or **`group`**ed. Notice that in S-PROGRAM, the thread-level judgment always begins with perspective (**`Grid`**,1): all the perspective management rules are congruences, narrowing the perspective of further evaluation as determined by the particular rule used.

These rules take great care to ensure that p always describes the relative position of a compute resource within its perspective; the payoff is that Bundl's semantics can later use this p value to model the way that Prism automatically adjusts indices into data when **`partition`**ing a data perspective. Beyond these key rules for managing perspective on code, we have modeled all other core features of Prism, such as asynchronous operations and thread synchronization, in Bundl. To handle such features we equip the operational semantics with additional structure, including sets of semaphores [25] for synchronization and a stack of effect handlers for modeling deferred asynchronous computations inspired by Ahman and Pretnar [3]. We have elided these details here for simplicity, but interested readers can find them in their full complexity in Appendix A.3.

4.3 Soundness Theorem

Together, the type system and operational semantics allow us to prove the following syntactic soundness theorem, which says that Bundl programs are type safe and do not get stuck trying to execute operations for which they lack the required perspective:

THEOREM 4.1. *(Type-and-Perspective Safety). For any program s such that $\Gamma \vdash^{\pi} s$, either:*

(1) s is **`skip`***, or*
(2) for any well-typed environments η, σ, and Σ, there is an s', η', σ', and Σ' such that $\eta,\sigma,\Sigma,t,b,p \vdash^{\pi} s \rightsquigarrow^{\star} s' \dashv \eta',\sigma',\Sigma'$ and $\Gamma' \vdash^{\pi} s'$, where Γ' is an extension of Γ, and η', σ', and Σ' are well-typed with respect to Γ'.

PROOF. Via the usual progress and preservation lemmas, available in Appendix A.4. □

It is worth noting that this soundness theorem guarantees a syntactic safety property, not a liveness property: it does not guarantee that all threads sharing perspective π that *can* reach a program point typed with π *will* eventually do so. Indeed, in the presence of nontermination, liveness does not hold—some of the threads could **`split`** off and loop forever. While we believe the liveness version of this theorem holds for a terminating fragment of Bundl, it is not provable with syntactic methods; the proof would require semantic techniques that are notoriously challenging and would be a research contribution [5, 28, 57] in and of itself. We plan to tackle this proof for Bundl in future work.

5 Implementation

Prism is implemented as an embedded language in Python. Once a program type checks, Prism lowers it to a CUDA file. All perspective information is erased during this step, and the generated CUDA contains no run-time checks. The file can then be compiled by **nvcc** [40], NVIDIA's closed-source compiler, to produce an executable. Because Prism operates at roughly the same level of abstraction as CUDA, there is a one-to-one mapping between most language constructs and their CUDA counterparts. A notable change is the addition of three parameters to each device function: the thread's relative ID, the block's ID, and an offset for shared memory allocations.

Inserting Synchronization. Most of Prism's implementation is straightforward, but inserting synchronization points is more involved. As described in Section 3.4, once data has been **partition**ed, Prism is responsible for synchronizing the data after the **partition** ends.

To determine where this synchronization must occur, Prism constructs a *data-control-flow graph* from the program. Nodes correspond to **partition**s, and edges capture program-order precedence: a parent **partition** must complete before its child begins. The graph can have backedges introduced through loops. In this graph, each **partition** is categorized as a read or a write **partition** by checking whether the **partition**ed variable ever appears as an lvalue. Synchronization points are inserted according to the following scheme:

(1) If the *parent* **partition** is a write, a synchronization point is inserted before the current one to ensure that it observes the most recent data; or
(2) If the *current* **partition** is a write, a synchronization point is inserted before it to ensure that all preceding reads have completed.

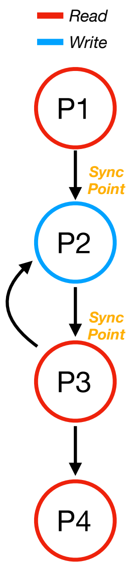

Fig. 16. An example data-control-flow graph with synchronization points.

Figure 16 shows an example graph with synchronization points derived from these two conditions. The inferred synchronization points in Figure 6 have also been marked on lines 40, 48, 56, and 63.

Using this information, Prism emits *wait* operations before **partition**s begin and *arrive* operations after they end, using CUDA's general *split-barrier* [20] primitive to implement them. For special cases, such as synchronizing an entire block or warp, Prism instead uses primitives like **__syncthreads()** or **__syncwarp()**.

Synchronization for asynchronous data movement is handled in exactly the same way and uses the same underlying graph. CUDA allows asynchronous loads to be associated with a split barrier, so Prism binds each asynchronous transfer to the appropriate wait–arrive pair inferred from the graph. Certain features, such as commit group-style synchronization [17, 18], require additional reasoning, and Prism performs further static analysis to insert the necessary synchronization.

It is worth noting that naively inserting synchronization immediately after each **partition** would be correct but prohibitively slow. To avoid this, Prism applies two optimizations: a wait-motion pass pushes waits downward toward the first use of the **partition**ed variable, and an arrive-motion pass pulls arrives upward toward its last use.

6 Evaluation

Having explained the design of Prism, we now evaluate it in the context of three main questions:

RQ1 Can Prism express a variety of composable CUDA programs?
RQ2 Can Prism express programs that use advanced GPU features?
RQ3 Can Prism match the performance of existing, speed-of-light CUDA code?

To perform this evaluation, we use two GPUs. The first is the NVIDIA H100 SXM5, a server-grade chip that supports Tensor Core operations and a dedicated hardware copy engine, the Tensor Memory Accelerator (TMA) [16]. Notably, the H100 introduces a new logical level called the warpgroup, and we show that our programming model can accommodate it. Moreover, because the H100 has historically served as the primary GPU for large-scale AI training, many CUDA kernels are already highly optimized for this hardware and achieve near speed-of-light performance, providing a rigorous baseline for comparison. To ensure our results generalize beyond the H100, we also test programs on a second GPU, the NVIDIA 4070 SuperTi—a consumer-grade chip.

As mentioned in Section 5, when Prism typechecks a program, it produces a CUDA file. We compile the CUDA file with `nvcc` version 12.3 [40] with flags `-O3 -use_fast_math`. We initialize all inputs using a random number generator [31] and report the average runtime sampled over 10 iterations, following a warm-up phase of 5 iterations. All aggregate results reported in this section are geometric means. All programs in the evaluation have been reproduced in Appendix B.

6.1 RQ1: Can Prism Express a Variety of CUDA Programs?

We evaluate Prism's expressivity by writing programs that have fundamentally different patterns of convergence, along with a library modeled after CUB [38] that contains composable pieces.

6.1.1 Programs with Different Convergence Behavior.

Matrix Multiplication *(4070Ti).* We chose matrix multiplication as our first benchmark for two main reasons. First, there exist several implementations that achieve near-peak performance, providing a strong baseline. Second, it allows for a range of increasingly sophisticated implementations that exercise different parts of the language, making it ideal for evaluating expressiveness.

We adapt the codebase from Boehm [6] to implement `float` matrix multiplication, commonly referred to as `sgemm`. Concretely, we compute $C \leftarrow \alpha AB + \beta C$ where A, B and C are matrices and α and β are scalars. As this is a `float` benchmark, it does not need advanced GPU features like asynchrony or Tensor Cores to achieve speed-of-light performance.

We implement five variants of the `sgemm` benchmark: (1) a naive version that follows the traditional single-program multiple-data pattern, written from the perspective of a thread; (2) a version that exploits memory coalescing [10], still expressed at the thread level; (3) a version that builds on (2) by introducing 2D tiling and staging data in shared memory, which requires shifting first to the perspective of a block and then to the perspective of a thread, while also requiring block-level synchronization; (4) a version that applies 2D tiling with vectorized loads, again written from the perspective of a block; and (5) a version that combines the optimizations from (2) through (4) while adding an additional level of tiling from the perspective of a warp.

We can express all five variants cleanly, and the resulting programs are almost the same as their CUDA counterparts, which distinguish perspectives through disciplined style. Prism, on the other hand, enforces this discipline at compile time. We evaluate the performance of these variants in Section 6.3.

Single-Pass Parallel Prefix Scan with Decoupled Look-Back *(4070Ti).* We also implement scan, a widely used parallel primitive, in Prism. We focus on the prefix-sum scan, which computes, for each position in an array, the sum of all elements up to that position. Prefix sum sits in a different corner of the GPU design space from matrix multiplication: it is memory intensive, requires careful attention to the convergence behavior of threads, and traditionally requires multiple passes over data.

We implement the single-pass parallel prefix scan with decoupled look-back, introduced by Merrill and Garland [36], an elegant algorithm that does not require multiple passes over the input data, and involves several distinct points of convergence. Within each block, work is decomposed into fine-grained thread-level and warp-level scans. After producing the local result, blocks publish their partial prefix to global memory. Finally, each block waits until enough information from earlier blocks is available, at which point it accumulates the value and completes its section of the scan.

We implement this full strategy in Prism. Our implementation uses **unsafe** to implement a global-memory spinlock that lets a block check when the previous block's data is ready.

6.1.2 *Case Study: Can Prism help build composable functions?*

While answering the other RQs, we found ourselves developing a small library of functions—similar in spirit to CUB—that we would frequently call. In this section, we qualitatively study how Prism can help programmers design libraries that they can compose with confidence.

As mentioned in Section 1, CUB occupies a unique design space in the GPU library ecosystem. Unlike many other libraries such as cuBLAS [12], cuDNN [14], and cuSPARSE [15], which provide host-side functions, CUB provides a device-side library organized into different levels. It makes these levels apparent by prefixing each of its functions with **Device**, **Block**, and **Thread**. The prefix sum already used variants of these functions, translated into Prism.

```
template<typename T, int BlockDimX,
    int ItemsPerThread, BlockLoadAlgorithm Algorithm,
    int BlockDimY = 1, int BlockDimZ = 1>
class BlockLoad:

// --- Calling the load function by instantiating the class ---

using BlockLoad = cub::BlockLoad<int, 128, 4, BLOCK_LOAD_DIRECT>;
// Allocate shared memory for BlockLoad
__shared__ typename BlockLoad::TempStorage temp_storage;
int thread_data[4]; // Thread local data
BlockLoad(temp_storage).Load(d_data, thread_data);
```

Fig. 17. Using **BlockLoad** in CUB.

Let's turn our attention to a particular CUB function—**BlockLoad**—and examine how it is equivalently expressed in Prism; we will see how Prism's type system reifies CUB's implicit assumptions. In CUB, the load function is implemented as a class, as shown in Figure 17.

CUB exposes a leaky abstraction, where information about the number of threads, block sizes, and other details seeps through:

(1) The CUB documentation needs to specify the number of threads that the function can assume to be available, because within the function, each thread must locate itself in the computation and use its **threadIdx.x** accordingly.
(2) The "item per thread" design can serve two purposes. The first is performance: if loops have constant bounds, they can be unrolled. The second is correctness: the function relies on the assumption that all threads call the function with an equal number of values to load.
(3) The CUB documentation specifies that **thread_data** can be data local to each thread.
(4) Finally, it says that if shared memory is being overwritten, a **__syncthreads()** call must be made to ensure that all reads have completed.

In Prism, however, we are able to describe these requirements through the interface shown in Figure 18, reducing the need to communicate numerous implementation details through documentation:

```
@prism("device")
@requires(block[1], thread[32])
def block_load(input : ptr(const(int)) @ block[1],
               output : ptr(int) @ thread[1],
               items_per_thread : int @ block[1]):
```

Fig. 18. **block_load** signature in Prism.

(1) We do not need to pass in the number of threads at all. Whenever Prism calls a function, it track the relative thread ID, so each function can be written locally as if it were running alone, rather than having to determine where the thread resides on the grid.
(2) We do not need to make **item_per_thread** a template argument for *correctness*. Its frequency is set at the function signature, so Prism will never allow a function to be called with a value living at a narrower perspective.

(3) In our interface, **thread_data** is explicitly set to a thread-local value. Since it is not marked as **const**, Prism conservatively assumes it may be written to, and enforces at compile time that only **thread[1]** values are passed in.
(4) Finally, using our synchronization pass outlined in Section 5, a **__syncthreads()** call will be inserted automatically if **input** is going to be used for writing.

Moreover, our version of CUB's function is built as a composition of two other functions, whose interfaces are shown in Figures 19 and 20. The **block_load** function is implemented by a call to **warp_load**, which in turn calls **thread_load**. Had we mistakenly attempted to call **warp_load** from inside **thread_load**, however, Prism would reject this, rather than silently failing at runtime.

```
@prism("device")
@requires(thread[32])
# block_load calls into warp_load
def warp_load(input : ptr(const(int)) @ thread[32],
              output : ptr(int) @ thread[1],
              items_per_thread : int @ thread[32]):
```

Fig. 19. **warp_load** signature in Prism.

```
@prism("device")
@requires(thread[1])
# warp_load calls into thread_load
def thread_load(input : ptr(const(int)) @ thread[1],
                output : ptr(int) @ thread[1],
                items_per_thread : int @ thread[1]):
```

Fig. 20. **thread_load** signature in Prism.

6.2 RQ2: Can Prism Express Programs that Use Advanced GPU Features?

To answer this question, we write a matrix multiplication for the **bf16** datatype on the H100, also known as **hgemm**. This benchmark is an acid test of our language, as **hgemm** pushes several language features to the extreme. To write an **hgemm** that can hit peak throughput on an H100, we need to write a warp-specialized kernel that uses the TMA–an asynchronous hardware copy engine that can move tiles of data at a time—and the warpgroup–level Tensor Core instructions, or **wgmma** [9]—new to the Hopper architecture. A high-performance kernel for this matrix-multiplication overlaps computation with data movement by pipelining loads.

The implementation in Prism looks different from CUDA code, particularly in how pipelining is expressed. Since Prism uses named variables introduced by **partition**s or **claim**s to determine the synchronization each region requires, when pipelining, we cannot dynamically change the pipeline slot simply by maintaining an index that wraps around based on the pipeline's length. Instead, each pipeline slots must be given separate names so that Prism can track them independently and overlap compute with data-movement. This leads to pipeline slots that must be individually named and forces the load logic to be effectively "unrolled". This, in turn, forces all pipelines in Prism to be statically sized. In practice, these pipelines are statically sized anyway to ensure they fit in shared memory.

Notably, for this benchmark, in addition to **wgmma** and TMA, the program needs to dynamically reallocate registers between producer and consumer warpgroups, an instruction only available on Hopper. This redistribution is a warpgroup-level collective operation, and Prism can check it like any other. Moreover, getting **wgmma** to work did not require introducing a new perspective into the language; **thread[128]** was sufficient. We did, however, need to add a dedicated TMA-style asynchronous data-movement construct, since Prism must eventually insert the appropriate synchronization for these transfers.

6.3 RQ3: Can Prism Match the Performance of Existing, Speed-of-Light CUDA Code?

In Section 6.1 and Section 6.2, we examined programs that expressed the same computation in multiple ways, relied on multiple points of convergence, and used advanced GPU features. We now discuss their performance.

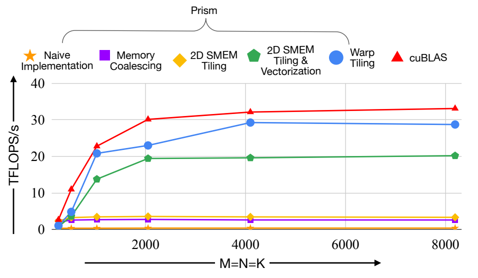

Fig. 21. Performance of **sgemm** on square matrices as matrix dimension $M = N = K$ increases.

The performance of the matrix multiplication benchmarks is shown in Figure 21, where we demonstrate that Prism is competitive with cuBLAS [12].

For the prefix scan, we compare our performance to CUB in Figure 22, which shows that Prism is within 7% of CUB's achieved bandwidth for arrays that do not fit in the L2 cache.

Finally, we evaluate our H100 implementation and show (in Figure 23) that it is competitive with cuBLAS on square sizes, coming within 15%. This performance difference arises due to one of **nvcc**'s optimization passes failing to trigger: the dynamic register allocation instruction is merely a hint—not a directive—to **nvcc**, and its optimization pass is sensitive to the way that pipeline structure is expressed. We emphasize that this is an exacting benchmark, and coming close to cuBLAS's performance demonstrates Prism's ability to control low-level features.

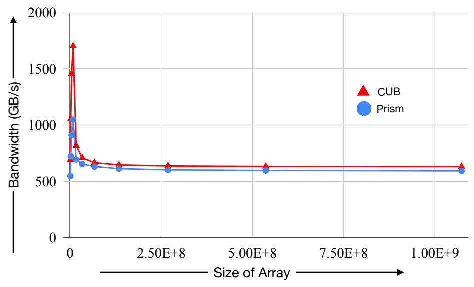

Fig. 22. Performance for prefix sum as input array size increases.

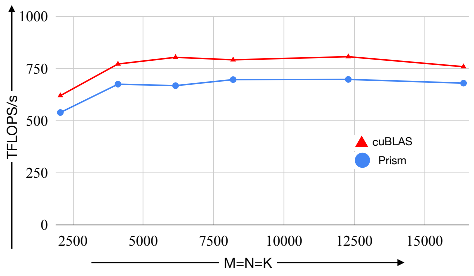

Fig. 23. Performance of **hgemm** on square matrices as matrix dimension $M = N = K$ increases.

7 Related Work

Prism builds on a rich tradition of systems languages for GPU programming and theoretical foundations of parallel programming. Prism differs from these systems in a crucial way: it treats collective operations, and therefore composability, as first-class concerns.

Imperative Languages for GPUs. CUDA [39], ROCm [2], and OpenCL [53] are imperative programming languages that expose low-level access to GPU hardware. These languages offer no support for managing collective operations, statically or otherwise; instead, users must orchestrate computation on the machine by describing how individual threads execute code. The subtle failures permitted by this model are the motivation for Prism.

Descend [35], a new Rust-inspired [50] language, uses a type system to track aspects of the compute and memory hierarchy and is thus the closest to our work. However, Descend's main concern is preventing data races, as opposed to ensuring that collective operations execute in valid contexts. As a result, Descend lacks support for many collective operations like Tensor Cores; in fact, because Descend allows threads to read their own IDs and thus induce data-dependent control flow, adding such support is not possible with its current design. It is also worth noting that while Descend does formalize a type system, it does not attempt to prove any properties about it.

Functional Languages for GPUs. Futhark [33], Accelerate [7] and Vertigo [27] are functional array languages with compilers targeting CUDA. These languages expose a high level interface, abstracting away details of the hardware entirely in exchange for stronger safety guarantees. Prism, however, exposes low-level details of the hardware and thus does not trade performance for safety.

Tile-Based Kernel DSLs. More recently, tile-based GPU languages—Triton [55], Pallas [52], Tilus [26], and Helion [44]—offer a middle ground between high-level abstraction and low-level control. However, these languages sidestep the question of composability entirely because they restrict programmers to a single layer of the hierarchy: a block (Triton, Tilus, Helion), or a warpgroup

(Pallas). Gluon [56], meanwhile, is a new low-level, tile-based language, but it does not check and cannot enforce that collective operations are executed at the correct layer of the hierarchy.

Task-Based and Scheduling Languages. Languages like Cypress [58], Halide [45], Fireiron [29] and RISE/ELEVATE [30] expose a scheduling language that lets users modify an existing reference program through external commands. Because these languages operate by transforming a fixed source of truth, they expose a fundamentally different programming model than Prism.

Libraries Built on CUDA. Many GPU libraries, such as CUTLASS [51], CUB [38], and ThunderKittens [47], expose device functions that operate at various levels of the compute hierarchy. These functions are typically organized via C++ namespaces to reflect their intended usage (e.g., warp-level, block-level). However, this organization is purely conventional: it encodes hierarchy through naming rather than enforcing it statically. As a result, correct use requires careful discipline from both the library implementer (to uphold naming and usage invariants) and the user (to correctly interpret them). Any mismatch or subtle misunderstanding between the intended and actual use of these functions goes unchecked by the compiler.

Theoretical Foundations. The design of Bundl is heavily inspired by existing work on dependency tracking [1]. Dependency tracking calculi allow type systems to track how data and code depend on each other, and have commonly been used to implement secure information flow analyses [24]. In Bundl, data that lives at a narrow perspective is unable to flow into data living at a broader perspective, and we use dependency tracking to capture this restriction in Bundl's type system.

Prior theoretical work has tackled reasoning about thread divergence. In particular, Muller and Hoffmann [37] build a sophisticated quantitative program logic for this use case. Singhania [46], meanwhile, uses static analysis techniques to predict thread divergence to unlock optimizations.

8 Conclusion, Limitations, and Future Work

We have presented Prism, a new, low-level GPU language that statically guarantees safe usage of compute resources by construction, without sacrificing low-level control. Prism introduces a new mental model for writing GPU code, which we are excited to make more expressive and ergonomic.

Particularly, we plan to improve the experience of writing pipelines in Prism. As discussed in Section 6.2, Prism currently requires pipeline slots to be explicitly named; we can make this more ergonomic this by adding language support for generating pipeline-style code.

Prism can be extended to more architectures; in particular, it can accommodate newer GPUs—such as Blackwell—that support coarser-grained Tensor Core operations. More broadly, we hope to generalize Prism's model to other hierarchical compute environments, including distributed systems.

We believe that Prism is capable of guaranteeing data-race freedom, but additional work is required to support this both formally and in practice. In particular, we think that if Prism restricted users to `partition`s with injective indexing functions, data-race freedom would follow naturally. We are also interested in exploring how the design principles of Descend [35], which build on Rust's ownership model [50], could be applied to Prism.

On the theoretical side, we plan to explore a terminating fragment of Bundl and prove the liveness property discussed in Section 4.3: that all threads sharing perspective π eventually reach the parts of a program viewed at that π. This amounts to showing that threads sharing the same perspective execute the same code and observe the same data, which we hope to prove using logical relations, following Turon et al. [57] and Spies et al. [48, 49].

We believe that Prism is a promising low-level substrate that enables confident composition and can serve as a foundation for building higher-level libraries and abstractions.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (San Antonio Texas USA, 1999-01). ACM, 147–160. doi:10.1145/292540.292555

[2] Advanced Micro Devices, Inc. 2024. AMD ROCm™ Software. https://www.amd.com/en/products/software/rocm.html. Accessed: 2025-11-10.

[3] Danel Ahman and Matija Pretnar. 2021. Asynchronous effects. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 1–28. doi:10.1145/3434305

[4] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: leveraging warp specialization for high performance on GPUs. *SIGPLAN Not.* 49, 8 (Feb. 2014), 119–130. doi:10.1145/2692916.2555258

[5] Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. 2012. A Concurrent Logical Relation. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL (2012)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 107–121. doi:10.4230/LIPIcs.CSL.2012.107

[6] Simon Boehm. 2022. SGEMM_CUDA: Fast CUDA matrix multiplication from scratch. https://github.com/siboehm/SGEMM_CUDA. GitHub repository. Accessed: 2025-11-10.

[7] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (Austin, Texas, USA) *(DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/1926354.1926358

[8] NVIDIA Corporation. 2022. NVIDIA H100 GPU Whitepaper. (2022). https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c Accessed: 2025-11-12.

[9] NVIDIA Corporation. 2025. Asynchronous Warpgroup Level Matrix Multiply-Accumulate Instructions. https://docs.nvidia.com/cuda/parallel-thread-execution/#asynchronous-warpgroup-level-matrix-instructions Accessed: 2025-11-13.

[10] NVIDIA Corporation. 2025. Coalesced Access to Global Memory. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#coalesced-access-to-global-memory Accessed: 2025-11-13.

[11] NVIDIA Corporation. 2025. cub::BlockReduce. https://nvidia.github.io/cccl/cub/api/classcub_1_1BlockReduce.html Accessed: 2025-11-09.

[12] NVIDIA Corporation. 2025. cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html Accessed: 2025-11-09.

[13] NVIDIA Corporation. 2025. cuBLASDx. https://docs.nvidia.com/cuda/cublasdx/#nvidia-cublasdx Accessed: 2025-11-11.

[14] NVIDIA Corporation. 2025. cuDNN. https://docs.nvidia.com/deeplearning/cudnn/latest/ Accessed: 2025-11-09.

[15] NVIDIA Corporation. 2025. cuSPARSE. https://docs.nvidia.com/cuda/cusparse/index.html Accessed: 2025-11-09.

[16] NVIDIA Corporation. 2025. Data Movement and Conversion Instructions: Bulk Copy. https://docs.nvidia.com/cuda/parallel-thread-execution/#data-movement-and-conversion-instructions-bulk-copy Accessed: 2025-11-12.

[17] NVIDIA Corporation. 2025. Data Movement and Conversion Instructions: cp.async.bulk.commit_group. https://docs.nvidia.com/cuda/parallel-thread-execution/#data-movement-and-conversion-instructions-cp-async-bulk-commit-group Accessed: 2025-11-13.

[18] NVIDIA Corporation. 2025. Data Movement and Conversion Instructions: cp.async.commit_group. https://docs.nvidia.com/cuda/parallel-thread-execution/#data-movement-and-conversion-instructions-cp-async-commit-group Accessed: 2025-11-13.

[19] NVIDIA Corporation. 2025. Data Movement and Conversion Instructions: Non-bulk Copy. https://docs.nvidia.com/cuda/parallel-thread-execution/#data-movement-and-conversion-instructions-non-bulk-copy Accessed: 2025-11-12.

[20] NVIDIA Corporation. 2025. Parallel Synchronization and Communication Instructions: mbarrier. https://docs.nvidia.com/cuda/parallel-thread-execution/#parallel-synchronization-and-communication-instructions-mbarrier Accessed: 2025-11-13.

[21] NVIDIA Corporation. 2025. Synchronization Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#synchronization-functions Accessed: 2025-11-11.

[22] NVIDIA Corporation. 2025. Warp Shuffle Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-shuffle-functions Accessed: 2025-11-10.

[23] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge,

Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. DeepSeek-V3 Technical Report. doi:10.48550/arXiv.2412.19437 arXiv:2412.19437.

[24] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (July 1977), 504–513. doi:10.1145/359636.359712

[25] Edsger Wybe Dijkstra. 1963. Over de sequentialiteit van procesbeschrijvingen. (1963). https://training-ir7.tdl.org/handle/123456789/594

[26] Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Yu Hao, Yida Wang, and Gennady Pekhimenko. 2025. Tilus: A Tile-Level GPGPU Programming Language for Low-Precision Computation. arXiv:2504.12984 [cs.LG] https://arxiv.org/abs/2504.12984

[27] C. Elliott. 2005. Vertigo — GPU Compiler & Embedded Language for Graphics Processors. http://conal.net/Vertigo/. Accessed: 2025-11-10.

[28] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2016. Proving Liveness of Parameterized Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, New York NY USA, 185–196. doi:10.1145/2933575.2935310

[29] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 71–82. doi:10.1145/3410463.3414632

[30] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP, Article 92 (Aug. 2020), 29 pages. doi:10.1145/3408974

[31] Horace He. 2024. Strangely, Matrix Multiplications on GPUs Run Faster When Given "Predictable" Data! https://www.thonking.ai/p/strangely-matrix-multiplications Accessed: 2025-11-13.

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

[33] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. *SIGPLAN Not.* 52, 6 (June 2017), 556–571. doi:10.1145/3140587.3062354

[34] Jan Hoffmann. 2011. *Types with potential: polynomial resource bounds via automatic amortized analysis.* Ph. D. Dissertation. lmu.

[35] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. 2024. Descend: A Safe GPU Systems Programming Language. 8 (2024), 841–864. Issue PLDI. doi:10.1145/3656411

[36] Duane Merrill and Michael Garland. 2016. *Single-pass Parallel Prefix Scan with Decoupled Look-back.* Technical Report NVR-2016-002. NVIDIA Corporation. Accessed: 2025-11-10.

[37] Stefan K. Muller and Jan Hoffmann. 2021. Modeling and analyzing evaluation cost of CUDA kernels. 5 (2021), 1–31. Issue POPL. doi:10.1145/3434306

[38] NVIDIA Corporation. 2025. *CUB: CUDA Unbound.* https://docs.nvidia.com/cuda/cub/index.html CUDA Toolkit Documentation.

[39] NVIDIA Corporation. 2025. *CUDA C++ Programming Guide.* NVIDIA. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ Accessed: 2025-11-06.

[40] NVIDIA Corporation. 2025. CUDA Toolkit 12.3 Downloads (Archive). https://developer.nvidia.com/cuda-12-3-0-download-archive. Accessed: 2025-11-11.

[41] NVIDIA Corporation. 2025. Parallel Thread Execution (PTX) ISA — Asynchronous Warpgroup Level Matrix Multiply-Accumulate Instructions. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html?highlight=tcgen05%2520cp#asynchronous-warpgroup-level-matrix-instructions. Accessed: 2025-11-10.

[42] NVIDIA Corporation. 2025. Parallel Thread Execution (PTX) ISA — TensorCore 5th Generation Instructions (tcgen05). https://docs.nvidia.com/cuda/parallel-thread-execution/index.html?highlight=tcgen05%2520cp#tensorcore-5th-generation-instructions. Accessed: 2025-11-10.

[43] NVIDIA Corporation. 2025. Parallel Thread Execution (PTX) ISA — Warp Level Matrix Instructions. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html?highlight=tcgen05%2520cp#warp-level-matrix-instructions. Accessed: 2025-11-10.

[44] PyTorch Team at Meta. 2025. Helion: A High-Level DSL for Performant and Portable ML Kernels. https://pytorch.org/blog/helion/. Accessed: 2025-11-10.

[45] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. doi:10.1145/2499370.2462176

[46] Nimit Singhania. 2018. *Static Analysis for GPU Program Performance*. PhD dissertation. University of Pennsylvania, Philadelphia, PA, USA. Supervisors: Rajeev Alur and Joseph Devietti.

[47] Benjamin Frederick Spector, Simran Arora, Aaryan Singhal, Arjun Parthasarathy, Daniel Y Fu, and Christopher Re. 2025. ThunderKittens: Simple, Fast, and $\textit{Adorable}$ Kernels. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=0fJfVOSUra

[48] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 80–95. doi:10.1145/3453483.3454031

[49] Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021. Transfinite step-indexing for termination. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 1–29. doi:10.1145/3434294

[50] Rust Team. 2025. Rust Programming Language. https://rust-lang.org/

[51] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Aditya Atluri, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. *CUTLASS*. https://github.com/NVIDIA/cutlass

[52] The JAX Authors. 2024. Pallas:Mosaic GPU — A JAX Kernel Language for GPUs. https://docs.jax.dev/en/latest/pallas/gpu/index.html. Accessed: 2025-11-10.

[53] The Khronos Group Inc. 2025. OpenCL™ — The Open Standard for Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opencl/. Accessed: 2025-11-10.

[54] Philippe Tillet. 2025. Introducing Triton: Open-source GPU programming for neural networks. https://openai.com/index/triton/

[55] 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. doi:10.1145/3315508.3329973

[56] triton-lang. 2025. "01-intro.py" – Introduction to Gluon tutorial in Triton. https://github.com/triton-lang/triton/blob/main/python/tutorials/gluon/01-intro.py. Accessed: 2025-11-10.

[57] Aaron J. Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Rome Italy, 2013-01-23). ACM, 343–356. doi:10.1145/2429069.2429111

[58] 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. doi:10.1145/3729262