Modular GPU Programming with Typed Perspectives

ANONYMOUS AUTHOR(S)

To program the modern GPU, one must balance two frames of mind simultaneously: on the one hand issuing instructions to individual threads to control their behavior, while on the other tracking the convergence of many threads acting in concert to issue *collective operations*—necessary for performance-critical hardware features like Tensor Cores. The tension between these two mindsets makes modular programming error-prone; GPU languages like CUDA offer no support for enforcing invariants about collective behavior, and such invariants are particularly fragile at function and module boundaries. A solution is required that enforces these invariants statically while still granting programmers access to the low-level control structures they need to write performant code. In this work, we provide this solution in the form of Parallax, a new GPU programming language that guarantees correct composition by construction using *typed perspectives*, which materialize the programmer's assumptions about the granularity at which they are controlling the behavior of threads. We describe the design of Parallax, implement a compiler for it, and lay its theoretical foundations in a core calculus called Prism. We evaluate Parallax against a diverse suite of benchmarks and find that it offers programmers the safety guarantees they need to confidently write modular code without sacrificing performance or low-level control.

1 Introduction

CUDA [23] 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 GPU kernel. A CUDA program executes in parallel across this hierarchy, but is written from the *perspective of a single thread*.

While operations are specified per-thread, some operations are only semantically valid when executed *collectively* by a group of threads. The **`__syncthreads()`** primitive, for instance, synchronizes all threads within a block, and will cause a deadlock if executed by only some of the threads within that block. There are many such collective operations, including tensor core instructions [25–27], warp shuffle operations , and other kinds of synchronization primitives . These operations, therefore, 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 groups of threads) hinders compositional reasoning . In CUDA, function calls, like all other statements, are per-thread operations. However, a function may contain code that executes a collective operation, creating a gap between the per-thread syntax for its invocation and the cooperative semantics of its execution.

This gap creates tension between abstraction and correctness, a tension apparent even in widely used CUDA libraries that package common functionality via function interfaces . Consider the following snippet of documentation taken directly from CUB [22], a library of parallel primitives. The documentation describes the **`BlockReduce`** function [7], in which all threads in a block collaboratively apply a reduction operation, such as a maximum or prefi sum, over an array:

> Computes a block-wide reduction for thread$_0$ using the specified binary reduction functor. The first `num_valid` threads each contribute one input element.
>
> - 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.

Through informal documentation, CUB 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. In fact, 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 requirements in CUDA, correct usage of this function depends on the user carefully reading and interpreting the documentation.

In this work, we ask the following question: can we provide low-level access to collective operations while statically guaranteeing that they execute with the necessary configuration of compute resources? By reifying these configurations 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 expose it in a type system that tracks perspectives. This tracking enforces that collective operations are executed with the necessary resources, while still allowing users to access low-level interfaces. 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 is a departure from previous work, which attempts to resolve the mismatch between per-thread syntax and collective execution by restricting access to low-level operations. Triton [37], a tile-based, multi-dimensional array language, limits the user’s perspective to the block level, which prevents programmers from writing the highest-performance kernels. A variety of functional languages, such as Futhark [18] , provide compile-time guarantees through their type systems but are machine-agnostic: they cannot expose low-level control over hardware . Some other efforts, like Descend [20], aim to provide a low-level, safe GPU systems programming language, but lack support for collective operations like tensor cores. As a result, CUDA remains the de-facto standard for writing high-performance kernels on modern GPUs.

We introduce Parallax, a new low-level GPU programming language that guarantees correct usage of hardware resources by construction . Inspired by dependency calculi [1], Parallax statically tracks the configurations of compute and memory resources with type-level perspectives. We also develop Prism, a core calculus underpinning Parallax, provide formal rules for manipulating the perspectives of both code and data, and prove *type-and-perspective safety*. This ensures that Prism is sound, and that code always has the right perspective to execute operations at run time.

A parallel goal, alongside to correctness , is performance. We incorporate modern GPU features such as tensor cores and asynchronous data movement into Parallax, and demonstrate that Parallax can achieve the same performance as hand-written, highly optimized code on an H100 and a 4070 Ti Super. Our contributions are:

(1) Parallax, a low-level GPU programming language with type-level perspectives that track the logical grouping of compute and memory resources (Section 3) ;
(2) Prism, a formal model of Parallax that tracks perspectives in both 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.
(3) An implementation of Parallax (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.

```
 1  int tid = threadIdx.x;
 2  if (tid >= 0 && tid< 31){
 3    float A[4];
 4    float B[2];
 5    float C[4] = { 0 };
 6    # Populate A with unique values
 7    for (int i = 0; i < 4; i++)
 8       A[i] = tid * 4 + 1;
 9    # Populate B with unique values
10    for (int i = 0; i < 2; i++)
11       B[i] = tid * 4 + 1;
12    # Issue a warp-level tensor-core
13    # operation: D = A * B  + C
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. Invoking a warp-level tensor-core instruction in CUDA. We elide some typecasts necessary to invoke the assembly operation.

```
 1  tid : int @ thread[1] = id();
 2  with group(thread[32]):
 3    A : float[4] @ thread[1]
 4    B : float[2] @ thread[1]
 5    C : float[4] @ thread[1]
 6
 7    # Populate A with unique values
 8    for i in range(0, 4, 1):
 9      A[i] = tid * 4 + 1
10      C[i] = 0
11    # Populate B with unique values
12    for i in range(0, 2, 1):
13      B[i] = tid * 4 + 1
14
15    # Issue tensor-core op.
16     intrinsic.mma(
17      A[0], A[1], A[2], A[3],
18      B[0], B[1],
19      C[0], C[1], C[2], C[3],
20      out=[A[0], A[1], A[2],
     ↪  A[3]])
21
22
```

Fig. 2. Invoking a warp-level tensor-core instruction in Parallax.

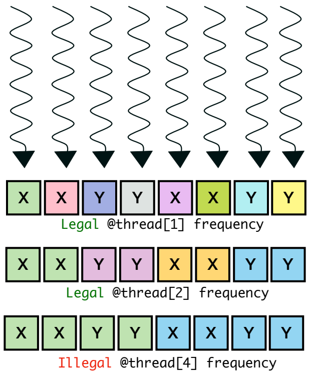
X
X
Y
Y
X
X
Y
Y
Legal @thread[1] frequency
X
X
Y
Y
X
X
Y
Y
Legal @thread[2] frequency
X
X
Y
Y
X
X
Y
Y
Illegal @thread[4] frequency

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 Background & Motivation

Before diving into the design of Parallax, 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 Parallax can prevent users from using these hierachies incorrectly in ways that CUDA would otherwise permit.

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. A *thread*, in turn, is the finest unit of execution, and it is the machine's basic unit of sequential control.

Users define the behavior of their 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 is executing 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 "warp-level" tensor core operation [27], 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.

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 instruction 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. In this case, the starting ID of the group of 32 participating threads must be aligned to a multiple of 32. So, if the condition on line 2 were instead `tid >= 1 && tid < 32`, the instruction 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 a particular point in the program and how those threads are arranged, accounting for alignment. This difficulty is further amplified due to two reasons. 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 the *warp-group*-level tensor core operations, which must be issued collectively by four warps, or the block-level `__syncthreads()` synchronization 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 interprocedurally *across* functions. A callee may assume a certain number of threads or blocks and may structure its computation, including convergence behavior, 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 that function correctly, users must read documentation, or worse, read the callee's implementation to understand its assumptions, breaking modular reasoning.

In Parallax, 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 Parallax. In Figure 2, the call to the `mma` intrinsic is valid only because it is executed within a `group` of 32 threads, as made explicit on line 2. Since Parallax's `group` construct already enforces both the size and alignment of participating threads, the validity of the `mma` operation is guaranteed by construction.

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 typically 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*, which is used for storing the thread's stack. Crucially, registers and local memory are per-thread data, while shared and global memory are accessible by multiple threads.

Thread-Local Data. Because registers and local memory are owned by individual threads, a variable with the same name in a program can have different values on different threads at run time.

```
int x = threadIdx.x;
if (x >= 0 && x< 31){
  float A[4];
  float B[2];
  float C[4] = { 0 };

  // k is A's stride
  A[0] = a_mem[(x/4)*k+(x%4)]);
  A[1] = a_mem[(x/4)*k+(x%4)+4]);
  A[2] = a_mem[(x/4+8)*k+(x%4)]);
  A[3] = a_mem[(x/4+8)*k+(x%4)+4]);

  // n is B's stride
  B[0] = b_mem[(x%4)*n+(x/4)]);
  B[1] = b_mem[(x%4+4)*n+(x/4)]);

  asm("mma.sync...");

  // Write back into c_mem
  // n is C 's stride
  c_mem[(x/4)*n+2*(x%4)] = C[0];
  c_mem[(x/4)*n+2*(x%4)+1] = C[1];
  c_mem[(x/4+8)*n+2*(x%4)] = C[2];

 ↪  c_mem[(x/4+8)*n+2*(x%4)+1]=C[3];}
```

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

In CUDA, threads can use this to induce divergent behavior. We've already seen an example of this: the `tid` variable in Figure 1. Branching on this variable is an instance of data-dependent control flow; in the general case, the possibility of divergence lurks anywhere that execution branches on data .

A key challenge for Parallax is ensuring that all threads organized by other language constructs like `group` remain logically unified, even in the face of data-dependent control flow, which is necessary to compute anything interesting. To do this, Parallax must track the frequency with which values vary in space. The `@` syntax attached to each variable declaration denotes the rate at which a value changes across compute resources on the machine. For example, a `thread[1]` value can vary for every thread, while a `block[1]` 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 the variable's value. Figure 3 illustrates valid and invalid colorings of a variable.

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

Thread-Shared Data. Shared and global memory, on the other hand, is not replicated by every thread; instead, all threads in a block see the same shared memory, while all threads on the grid see the same 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. Parallax, 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 ??. For now, however, we focus on how memory converges and diverges in tandem with the compute hierarchy.

Because threads share their view of global and shared memory, these spaces are subject to convergent and divergent behavior the same way that threads are. For example, when launching a kernel on the GPU, a global memory pointer may initially belong 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 within the larger memory region so each thread can write independently. After these writes, the memory must *reconverge*, ensuring that all threads have completed their updates before the memory can safely be returned to its original logical owner.

Let us reconsider the tensor core example in Figure 1, this time initializing the operands of the tensor core operation from pointers into memory. In Figure 4, `A` and `B` are now populated from data in global memory, `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. When entering this code, each thread in a group of 32 threads sees the same value for `c_mem`. Then, each thread within these 32 locates an offset within `c_mem` that it can write to (lines 21-24). To restore

```
 1 @ccuda("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 : int @ thread[1] =
27     lambda ro, co: (x/4+ro)*n+2*(x%4)+co
28    with parti-
   ↪  tion(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 inferred by name
```

Fig. 5. Invoking a warp-level tensor core instruction in Parallax after loading data from memory.

```
 1  @ccuda("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          # SMEM 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 : int @ thread[1] = id() * 4
33          # To write to C_smem, drop to thread[1] perspective
34          with partition(C_smem,p=thread[1],offset=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 (back edge from for loop)
41          for j in range(0, 4, 1):
42            global_row : int @ thread[1] = warp_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 (back edge 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 (back edge from for loop)
57            with claim(C_smem, p=thread[32]) as Cs_warp:
58                match split(thread):
59                  case 32:
60                    # Call tensor core function inside
61                    # a function interface.
62                    simple_mma(A_smem, B_smem, Cs_warp)
63
64        # Write back final result to C
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. Tensor Float 32 matrix Multiplication in Parallax.

c_mem back to its 32 thread-level ownership, all 32 threads must first synchronize 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 (Section 2.1), CUDA programmers are responsible for tracking the logical owner of a piece of memory as it evolves over the course of a program, and for ensuring that appropriate synchronization occurs whenever that ownership changes.

Parallax, by contrast, makes the evolution of a memory object's logical ownership explicit in the type system and *automatically* inserts the synchronization required to restore memory to its original ownership. In Figure 5, we show how memory is lowered through the compute hierarchy for the same tensor core operation in introduced in Figure 4. The lowering of **c_mem** is required in this program as Parallax only permits writes when data is owned by a single thread. To lower **c_mem**, we use Parallax's **partition** operation on line **31**. The operation takes the name of the memory to partition, **c_mem**, and lowers it to a single thread, and assigns it a new name, **c_thrd**; Parallax will not allow references to the old name **c_mem** within the partition's scope. The **partition** function also takes an indexing function, and each time **c_thrd** is accessed, this indexing function is implicitly applied. Once the **partition**'s scope ends, Parallax will insert a synchronization before the first use of the original name, so that all per-thread writes have completed.

We would like to emphasize that preventing data races is not one of Parallax's goals. In Parallax, data races *can* occur; however, since the data is eventually synchronized before it is reused, the last-writer wins. Out-of-bounds accesses are considered undefined behavior. We believe that prior work, such as Descend [20], lays the foundation for reasoning about data-race free programs, and that a Descend-like memory model can be adapted to Parallax. Parallax is instead focused on the *interaction* between the compute and memory hierarchies, and on reasoning about them simultaneously to ensure that an operation is executed only when sufficient compute resources are available, a guarantee that prior work like Descend cannot provide.

3 The Parallax Language

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

To guide our discussion about Parallax'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 a 16×8 independent tile of the output. To do so, the it first locates the tile index assigned to it (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 on (line **62**), requiring threads in a warp to *converge* at the point. We encapsulate this tensor-core instruction in a function, demonstrating how function composition works in Parallax. The function is the same as Figure 5.

3.1 Levels

Parallax models the machine's compute hierarchy through *levels*. There are three levels in Parallax—**`grid`**, **`block`** and **`thread`**—which are organized as expected: a **`grid`** consists of multiple units of the **`block`** level, which in turn consist of multiple units of the **`thread`** level. Levels have an order defined over them, with **`thread`** < **`block`** < **`grid`**.

There are two key differences between Parallax's levels and those in CUDA. First, Parallax does not model CUDA's three-dimensional grid or block structure. Second, there are two commonly-used "levels", *warp* and *warpgroup*, noticeably absent from our hierarchy. On the hardware, the units of each level are arranged in a single 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 only became a meaningful construct with the release of the Hopper architecture , consists of 128 threads aligned to 128.

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

3.2 Perspectives

Perspectives are the central concept of Parallax. A perspective represents the layer of the hierarchy from which the programmer is defining the machine's behavior. They are what allow Parallax to determine which compute resources the programmer is controlling, whether they are available in the program's context, and, once provided, 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. A perspective is written as **`level`**`[n]` . 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 have a partial order defined on them. We say that $\mathtt{level}_2[\mathtt{n}_2]$ is *broader* than $\mathtt{level}_1[\mathtt{n}_1]$, or that $\mathtt{level}_1[\mathtt{n}_1]$ is *narrower* than $\mathtt{level}_2[\mathtt{n}_2]$, if either:

(1) $\mathtt{level}_1 < \mathtt{level}_2$; or,
(2) $\mathtt{level}_1 = \mathtt{level}_2$ and $\mathtt{n}_2 \,\%\, \mathtt{n}_1 = 0$.

The condition $\mathtt{n}_2 \,\%\, \mathtt{n}_1 = 0$ enforces alignment, and we will see why it is required in Section 3.4.

3.3 Perspectives on Code

Code is associated with a set of perspectives, called a *perspective bound*. Perspective bounds directly correspond to the compute resources a program expects to exercise control over the course of its execution. 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 Parallax 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 bounds.

Each function starts out with a top-level perspective bound defined using the notation **`@requires(grid[`**n_1**`],b`**
In Figure 6, the perspective bound is defined on line 2. Starting from this top-level declaration, programmers can 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. That is, **`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 logically controls the whole grid's behavior. To produce different output tiles, the programmer needs to be able to control the behavior of individual blocks, and they shift their perspective to **`block[1]`** on **todo**. Inside the **`group(block[1])`**'s scope, the behavior of the machine can be defined with respect to a single block, and code is replicated across all blocks in the grid.

As the reader may have already guessed, not all uses of **`group`** are valid. The examples in Figure 7 have no interpretation on the hardware. On line 4, the program tries to broaden its perspective to **`block[1]`** from a **`thread[1]`** perspective, which is clearly illegal. On the other hand, in the second example, the program tries to narrow its perspective to a **`block[5]`** from a **`block[6]`** perspective. While $5 < 6$, Parallax cannot evenly replicate **`block[5]`** perspective inside a **`block[6]`** perspective, and so rejects this program. Recall, from Section 3.2, that the condition on whether a perspective is narrower than another is actually a condition on divisibility, not just size.

```
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`**.

To eliminate such malformed cases, Parallax only allows an invocation of **`group(level[n])`** if the current perspective bounds contains a perspective broader than **`level[n]`**. Once **`group(level[n])`** is invoked, it modifies the current perspective bounds in two key ways. First, it removes all perspective broader than **`level[n]`** from it. Second, it sets the broadest perspective within that **`group`**'s scope to be **`level[n]`**.

Split. Unlike **`group`**, which is used for replication into multiple narrower perspectives (all of equal size), **`split`** is used for sharding the current perspective unequally. For example, Figure 8 shows a **`split`** from **`thread[4]`** perspective to one branch with **`thread[2]`** perspective and two with **`thread[1]`** perspective. 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.

[1]Despite the use of the pattern-matching **`case`** syntax, *all* branches of the **`split`** execute.

Use of **split** is necessary to write warp-specialized 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 63 in Figure 6 shows a **split** that requests the first warp in the block, narrowing from **block[1]** into a single **thread[32]**. Later in Figure 8, this warp is used to execute a tensor-core operation.

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

Fig. 8. Simple example of **split**.

Because **split** corresponds to unordered composition, it must provide each of its branches their requested perspectives simultaneously. Parallax thus checks that the cumulative sum of the perspectives requested by each branch 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 sum to less than or equal to the available units are valid. Figure 10 shows an example of this constraint being violated.

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 exactly **n**.

```
# Example 1
with group(thread[4]):
 match split(thread):
   case 4:
     ...
   # Illegal! 4 + 1 > 4
   case 1:
     ...
```

Fig. 9. Illegal split exceeds the available perspectives.

```
# Example 2
with group(thread[3]):
 match split(thread):
   case 2:
     ...
   case 1:
     ...
   # Illegal! 2 + 1 <= 3,
   # but the second branch
   # is not aligned by 2.
```

Fig. 10. Illegal split violates alignment.

3.4 Perspectives on Data

Section 3.2 described how a programmer can control different layers of the hierarchy by changing the perspective of code through **group** and **split**. To ensure these operations remain meaningful, Parallax must ensure that compute units inside a perspective do not diverge. Parallax must thus reason about how control flow introduced through **for**, **while** and **if** statements affects the control flow of the threads that comprise a perspective. To guarantee data-dependent control flow does not cause divergence, Parallax tracks how data may vary across the threads of a perspective . Parallax distinguishes between two kinds of data: thread-local data (or, register and stack data), and thread-shared data (or, shared memory and global memory).

3.4.1 *Thread-Local Data.*

Thread-local data is data stored in registers and stack-variables that is private to each thread. In the program source in CUDA, a single thread-local variable may contain different values at runtime simultaneously across different threads. Indeed, to compute anything useful, threads must operate on different input values, and so reasoning about this divergent state is essential.

Instead of imposing severe restrictions on data-dependent control flow, Parallax solves this problem by assigning variables perspectives. Intuitively, the perspective of a variable tells Parallax the *frequency* at which its values change in space. For example, a **grid[1]** variable is the same for all threads in a grid, meanwhile a **block[2]** variable may change every two blocks. Crucially, this frequency tells us that a **block[2]** variable is *indistinguishable* by threads within 2 blocks where the starting block ID is aligned to 2. Figure 3 shows different frequencies as different colorings of a variable.

Programmers specify the perspective that each variables live at in its declaration. A variable **v** of type **int** is declared at **thread[1]** perspective using the syntax **v : int @ thread[1]**.[2] Variables are only available for reading and writing from certain perspective. The rule can be summarized

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

as follows: Parallax allows programs to "write down" to narrower perspectives and "read up" from broader ones.

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. Because the variable **`flag`** varies across threads, branching on it causes a subset of threads in the block to reach the **`__syncthreads()`**, causing a deadlock, despite the fact that **`__syncthreads()`** should always be safe in **`block[1]`**.

```
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 perspectives. For example, a **`block[1]`** variable can be used to write into a **`thread[1]`** variable, but values from a **`thread[1]`** variable cannot be written into a **`block[1]`** variable. The latter constraint is demonstrated in Figure 12, where writing from a **`thread[1]`** variable into a **`block[1]`** one again would cause a deadlock.

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 rule "read up" in action on lines **45 and 46**. Meanwhile, "write down" is being used to set up variables on line **18-22**.

```
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.

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 Parallax, there are three mechanisms for obtaining thread-shared data. The first method is to directly allocate data residing in shared memory (Parallax assumes that all global memory allocations have been made before launching the CUDA kernel, as is typically the case with CUDA programs). 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.

Parallax simplifies shared memory allocation by requiring all shared memory allocations have a static size. Since shared memory is only visible to all threads in a block, a shared memory allocation is only valid with if code has a **`block[1]`** perspective. An example of such an allocation is shown on lines **28-30** of Figure 6. During compilation, Parallax will automatically handle the necessary pointer arithmetic to assign each allocation an appropriate offset within the shared memory space. The **`@requires`** annotation specifies the amount of shared memory a function expects to have available, and Parallax 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. Figure 6 declares the amount of shared memory it will require on **2**. We use standard techniques to statically bound memory usage with Parallax's type system.

Now, given an allocation into global or shared memory, Parallax introduces two language constructs **`partition`** or **`claim`** that help programmers narrow the perspective at which a pointer resides.

Partition. The **`partition`** operation plays the same role for memory that **`group`** plays for code. It is used to narrow the perspective of a pointer by creating many copies of it at a narrower perspective, and adding to each a distinct offset from the original pointer.

For example, a use of `partition` could take an array `x : ptr(int) @ thread[4]` and compute offsets `x + i : ptr(int) @ thread[1]`. While the original address was identical across groups of 4 threads, the new offset pointers have diverged to point to different addresses on each thread. These new pointers can then be used within the scope of the `partition`. After the partition ends, the threads participating in the `partition` must synchronize to restore the original pointer `x` to its 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 in an inner scope.

Inside the scope of the partition, original variable `x` cannot be used, and the memory can only be accessed using `y`. Each use of `y` applies the indexing function `f` to compute the true offset of the access. In Figure 6, for example, we partition the `C` buffer to `block[1]` perspective on line **25** first, and then narrow it further to `thread[1]` perspective on line **78** so that we can actually write to it. When the scope of the `partition` statement ends, Parallax automatically inserts the necessary synchronization to restore the memory to its original perspective. Strictly speaking, this synchronization is only required at the next point the memory is to be *re-used*. Parallax has a mechanism for optimizing the placement of these synchronization points, which is describe in Section 5.

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, the resulting partition is free of data races. When the indexing function is not injective, multiple threads may write to the same location, introducing a potential data race. Out-of-bounds accesses are undefined behavior.

As we discussed in Section 2.2, achieving data-race freedom is *not* one of Parallax's goals, and our guarantees continue to hold even when indexing functions are not injective. Because Parallax automatically inserts the necessary synchronization before the original memory name is restored to its higher perspective, Parallax ensures that all writes have completed before accessing the orginal name. The subsequent uses of that pointer will observe the value written by the last writer.

Claim. The `claim` operation lets the programmer narrow a pointer's perspective without adding an offset, by giving it to only one collection of threads with that narrower perspective. For example, line **57** of Figure 6 shows an example of `claim`, where the pointer `C_smem @ block[1]` is *claimed* by a single warp in the block and narrowed to `Cs_warp @ thread[32]` to call a Tensor Core operation.

Concretely, the `claim` construct takes the original pointer variable `x`, the target perspective to narrow to, and a new name `y` to assign to the narrowed memory. As with `partition`, the original pointer `x` is not accessible within the scope of the `claim`. Morevoer, if the scope of the `claim` includes a `splits`, the new pointer `y` is accessible only within *one* of its branches; sibling branches are not permitted to read or write from this memory.

3.5 The `id()` Function

As opposed to exposing users to special hardware `blockIdx.x`, `threadIdx.x` variables directly, Parallax provides the `id()` function instead. The `id` function returns the relative index from a given perspective. The interpretation of the `id` function depends on both the perspective of the variable it is being written to, and the perspective that the `id` function is being invoked in. In our example in Figure 6, we use a call to `id` in `grid[1]` scope at line **18 and 19** to locate the tile that each block is ultimately charge of computing. In Figure 2, however, a call to `id` in `thread[32]` scope at line **7** will not return the global thread ID of the threads launched in the kernel, but rather the *relative* thread ID within the 32 threads, ranging from 0 to 31.

3.6 Collective Operations

With perspectives on code and data in place, we can now turn our attention to Parallax's reasoning about collective operations.

There are two ways to access collective operations in Parallax. The first is to use intrinsics provided by the compiler. These intrinsics have built-in perspective classes attached to them, and Parallax enforces that the opeation is only ever executed if code is being executed from the corret perspective. In Figure 2, the tensor-core call at line 22 is calling into intrinsics provided by our compiler. The second, more flexible approach allows users to extend these operations using the **`unsafe`** feature. Using unsafe code, users can inline a collective operation—often a single assembly instruction—into a Parallax function and ensure that the function's top-level requirements, specified with **`@require`**, faithfully describe the perspectives required to execute that instruction. Then, Parallax will check the call to the function as any other function call, and will check whether it is being called with a compatible perspective. The rules for checking function calls are described in Section 3.8.

3.7 Asynchrony

Asynchronous data-movement works similarly to other data operations.

Users can mark storage as asynchronous with the **`async`** construct. As with **`partition`** and **`claim`**, invoking **`async`** removes the old name and exposes a fresh name that is only accessible within the scope of the **`async`** statement. Inside this scope, Parallax restricts the new name to be used solely with the asynchronous data-movement intrinsics that the language provides. These intrinsics operate on a per-thread level, so Parallax will require users to program from the **`thread[1]`** perspective.

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

Fig. 13. Example of an **`cp.async`** instruction in Parallax.

Parallax models two primary forms of asynchronous data movement: async mempy memory copies and tensor-memory accelerator instructions . We show an example of the former in Figure 13. As with the data operations in ??, Parallax is responsible for inserting the necessary synchronization before the program's first use of the original name, ensuring that the asynchronous transfer has completed.

3.8 Function Composition

To check whether a function call is valid, Parallax needs to ensure that the call site can provide the set of perspectives that a function will attempt to use over its execution, and that the arguments passed to the function also respect the perspectives declared for its input and output types. Both requirements are represented explicitly via the **`@require`** construct.

As mentioned in Section 3.2, each function carries a perspective class. We call this top-level perspective class the function's *perspective signature*. The signature specifies the top-level perspective class the function *assumes* will be provided to it. This signature represents the *minimum amount* of compute and memory resources the function must be called with. In the example above in Figure 6, line 2 defines the perspective signature. Up to this point, we have described programs written *inside* functions under the assumption that the resource signature can be satisfied. In Section 3.2, for example, we used this signature to set up the top-level perspective class for the function.

Compute perspectives. At a function's call site, then, we need to ensure that the function's perspective signature can be satisfied. Snap compares the current compute perspective with the one required by the function, and verifies that only functions whose perspective signatures can be satisfied are called.

Per-argument checks. In Parallax, functions have pass-by-value semantics. For arguments, we distinguish between primitive data types and pointers to memory. For primitive data types, like `int`, `bool`, and stack allocations, users can pass either arguments that live at or above the data perspective specified by the function's signature. For pointers, we allow passing pointers that live at higher data perspective if the pointer is marked as `const`, indicating that it will only be used for reading (line **4 and 5** in Figure 2). Otherwise, we require that pointers match the exact perspective of the function signature.

4 Formalization in Prism

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

In this section, we describe Prism'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 Prism'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 can guarantee that dynamically-realized perspectives match the ones inferred by the type system.

4.1 Prism Type System

The core idea in Prism 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 Parallax.

The typing context also tracks the perspective at which each variable lives; data can only be read or written from 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, which can be 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, and these rules fall into two main categories: rules for managing the code perspective and rules for managing data perspective.

4.1.1 *Managing Perspectives on Code.*

In Parallax, there are two operations which manage code perspectives: **`group`** and **`split`**. In Prism, to better model the details of how perspectives shift throughout programs, we introduce a third construct, **`destruct`**.

Prism's **`group`** statement directly corresponds to Parallax's, and is checked by the T-Group rule. Given some perspective that can be divided into q equally sized narrower πs, the statement **`group`** $q\,s$ will check with perspective $q \cdot \pi$ provided that s itself checks with perspective π. The starting perspective is of the form $q \cdot \pi$, and encodes Parallax's alignment requirement.

The **`split`** statement, meanwhile, is checked by the T-Split rule, and functions like a binary version of the n-ary **`split`** construct in Parallax. 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.

The final rule for managing code perspective is T-Destruct, which shifts perspective down the GPU hierarchy and makes explicit in Prism programs exactly where such shifts occur. The

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

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

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

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

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

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

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

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

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

$\downarrow$ operation on πs "destructs" the perspective into many narrower perspectives at a lower level; $\downarrow(\texttt{Grid},1) = (\texttt{Block},B)$ and $\downarrow(\texttt{Block},1) = (\texttt{Thread},T)$, where B and T are parameters to a particular instantiation of Prism to describe the number of blocks per grid and threads per block.[3] Because $\downarrow$ is only defined on $(\texttt{Grid},1)$ and $(\texttt{Block},1)$, the rule enforces that one can only **`destruct`** their code perspective with exactly one grid or block.

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 the variable to be partitioned, x, is the same as the current perspective on code. After partitioning, a fresh variable y is introduced with a new perspective π/c, which we use as a shorthand to denote a division of a perspective π into c equally-sized narrower perspectives. Within the body of the **`partition`**, we disallow any references to the original variable x and continue checking with the original code perspective π; the **`partition`** has no effect on the current 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, Prism 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

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

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

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

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

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

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

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

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

represents a minor difference from Parallax, which uses additional static analysis to ensure that a claimed variable is only accessed in a single `split` branch.

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.

4.2 Prism Semantics

Having explained the key rules of the type system, we can move on to discuss Prism's operational semantics. To reflect the fact that GPU programs execute in parallel across numerous threads, we model the semantics of Prism in the style of Turon et al. [40], using a two-level small step judgment. We present the key rules of this semantics in Figure 15.

The top level (i.e., device-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* to an updated collection of memories and 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 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.[4]

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,

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

- 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 Prism 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 Prism, but is erased by Parallax during compilation; we will later 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 Prism's semantics can later use this p value to model the way that Parallax automatically adjusts indices into data when partitioning a data perspective. Beyond these key rules for managing perspective on code, we have modeled many of the other features of Parallax, such as asynchronous operations and thread synchronization, in Prism. To handle features like these we equip the operational semantics with additional structure, including sets of semaphores [12] 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 see further details 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 Prism 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 $\eta,\sigma,\Sigma,t,b,p \vdash^{\pi} s \rightsquigarrow^{\star} s' \dashv \eta',\sigma',\Sigma'$ such that $\Gamma' \vdash^{\pi} s'$, 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 Prism, it is not provable with syntactic methods; the proof would require semantic techniques that are notoriously challenging and would be a research contribution [4, 15, 40] in and of itself. We hope to tackle this proof for Prism in future work.

5 Implementation

Parallax is implemented as an embedded language in Python. Once a program type checks, Parallax 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` , NVIDIA's closed-source compiler, to produce an executable. Because Parallax operates at roughly the same abstraction level 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 into shared memory that it can use for allocations.

Inserting Synchronization. Most of Parallax's implementation is straightforward, but inserting synchronization points is more involved. As described in ??, once data has been partitioned, Parallax is responsible for inserting the appropriate synchronization points when the partition's scope ends.

To determine where these points need to be inserted, Parallax constructs a *data-control-flow graph* from the program. Each node corresponds to a partition, and each edge captures program-order precedence: the parent partition must complete before the child begins. The graph can have backedges introduced through loops. In this graph, each partition is categorized as a *read* or a *write* partition. Synchronization points are inserted according to the following scheme:

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

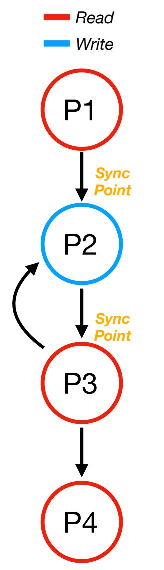

Fig. 16. An example data-control-flow graph lifted by Parallax with its inferred synchronization points.

Figure 16 shows an example graph with the synchronization points derived from these two conditions. The inferred synchronization points in Figure 6 have also been marked on lines **TODO**.

Using this information, Parallax emits *wait* operations before a partition begins (to wait for its parent) and *arrive* operations when a partition ends (to signal completion). CUDA exposes a general *split-barrier* primitive that we use to implement these waits and arrives. For special cases, such as synchronizing an entire block or warp, Parallax 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 load operations to be associated with a split barrier, so Parallax binds each asynchronous transfer to the appropriate wait–arrive pair inferred from the graph. Certain features, such as commit-group–style synchronization , require additional reasoning, and Parallax performs further static analysis to insert the necessary synchronization around these operations.

It is worth noting that naively inserting synchronization immediately after each partition would be correct but prohibitively slow. To avoid this, Parallax applies two optimization passes. A wait-motion pass pushes waits downward toward the first use of the partitioned name, and an arrive-motion pass pulls arrives upward toward its last use, reducing unnecessary stalls while preserving correctness.

6 Evaluation

Having explained the design of Parallax, we now evaluate it in the context of four main questions:

RQ1 Can Parallax express a diversity of composable CUDA programs?
RQ2 Can Parallax express programs that use advanced GPU features?

RQ3 Can Parallax 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. Notably, the H100 introduces a new logical level called the *warpgroup* (collection of 4-aligned warps), and we show that our programming model can accommodate this new level. 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 additionally test programs on a second GPU, the NVIDIA 4070 SuperTi GPU, a consumer-grade chip.

Like mentioned in Section 5, when name typechecks a program, it produces a CUDA file. We compile the CUDA file with `nvcc` version 12.3 [24] with flags `-O3 -use_fast_math`. We initialize all inputs using a random number generator. We report the avergae 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 Section B.

6.1 RQ1: Can Parallax express a diversity of CUDA programs?

We evaluate Parallax's expressivity by writing programs that have fundamentally different patterns of convergence, and writing a library modeled after CUB that contains composable pieces.

6.1.1 Programs with Different Convergence Behavior.

Matrix Multiplication. 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, matrix multiplication allows for a range of increasingly sophisticated implementations that stress different parts of the language, making it ideal for evaluating expressiveness.

To undertake this exploration, we adapt the codebase from [5] to implement matrix multiplication on the `float` datatype, commonly referred to as `sgemm`. Concretely, we compute $C = \alpha * A \times B + \beta$ where A, B and C are matrices with appropriate sizes and α and β are scalars. As this is a `float` benchmark, it *does not need to* use advanced GPU features like asynchrony or tensor cores to achieve speed-of-light performance. We will discuss these advanced features in Section 6.2.

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, 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 remain close to their CUDA counterparts, which adopt a separation between perspectives through disciplined style. Parallax, on the other hand, requires programmers to follow this discipline, and enforces it at compile time. The only notable difference is that some expressions must be hoisted earlier in the program so that their logical introduction corresponds to the required perspective structure. We evaluate the performance of these variants in Section 6.3.

Single-pass Parallel Prefix Scan with Decoupled Look-Back. We also implement scan, a widely used parallel primitive, in Parallax. 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 different 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 [21], an elegant algorithm that does not require multiple passes over the input data. In this algorithm, each block computes a local prefix sum over its assigned region of the array. Once finished, it writes the final element of its region into a global array. Each block then *looks back* to accumulate the contributions of prior blocks, allowing them to independently determine the global prefix without a full sweep over memory. This decoupled look-back mechanism lets blocks progress at different speeds while still producing correct global results.

The algorithm 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 prefix information from earlier blocks becomes available, at which point it accumulates the value and completes its section of the global scan.

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

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

Over the course of our evaluation, we found ourselves developing a small library of functions—–similar in spirit to the CUB library—– that we could call to execute operations. In this section, we would like to qualitatively study how Parallax can help programmers design libraries that they can compose with confidence.

As mentioned in Section 1, CUB occupies a unique design point in the GPU library ecosystem. Unlike many other libraries such as cuBLAS [8], cuDNN [9], and cuSPARSE [10], which provide global interfaces that users can call, 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 single-scan prefix sum in used variants of functions we translated into Parallax already.

Currently, we have a library of functions that perform loads functions across block, warp and thread; store functions across block, warp, thread, and scans across a block and a warp. Let's turn our attention to a particular CUB function—the store function—–and examine how it is equivalently expressed in Parallax, and how Parallax's type system reifies the assumptions implicit in CUB.

In CUB, the store function is implemented as a class, as shown below:

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

To use it, users must first instantiate an object of this class, and then call it with shared memory.

```
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);
```

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. If the starting **`threadIdx.x`** is not 0, the function must compute its relative ID internally.

(2) The “item per thread” design is interesting and serves two purposes. The first is related to performance: if loops have constant bounds, they can be unrolled, enabling downstream optimizations. The second is related to 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 also needs to specify that **`thread_data`** can be data local to each thread.
(4) Finally, the CUB documentation specifies that if shared memory is being overwritten, a **`__syncthreads()`** call must be made to ensure that all reads have completed.

On the other hand, the same function has a completely different interface in Parallax:

```
@ccuda("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]):
```

In Parallax, we are able to encapsulate code effectively, reducing the need to communicate numerous implementation details through documentation:

(1) We do not need to pass in the number of threads at all. Whenever Parallax calls a function, it threads the relative ID through, so each function can be written locally as if it were running alone, rather than having to determine where the thread resides in the global array.
(2) We do not need to make `item_per_thread` a template argument for *correctness*. Its frequency is set at the function signature, so Parallax will never allow a function to be called with a value living at a narrower perspective than its signature requires.
(3) In our interface, **`thread_data`** is explicitly set to a thread-local value. Since it is not marked as **`const`**, Parallax 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 **`src`** is going to be used for writing.

Moreover, our CUB functions is built as a composition of 2 other functions:

```
@ccuda("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]):

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

If we had mistakenly attempted to call **`warp_load`** from inside **`thread_load`**, Parallax would return an error, not silently compute fail to meet the contract set out up the function.

6.2 RQ2: Can Parallax express programs that use advanced GPU features?

To test whether Parallax can express programs that use advanced features of modern GPUs, we write a matrix multiplication for the **`bf16`** datatype for the H100.

This benchmark is an acid test of our language, as the H100 **`bf16`** matrix multiplication pushes several language features to the extreme. To write a matrix multiplication that can hit peak throughput on an H100, we need to write a warp-specialized kernel that uses the Tensor Memory Accelerator (TMA) , an asynchronous hardware copy engine that can move tiles of data at a time, and uses the warp-group–level tensor core instructions, or **`wgmma`**, specifically introduced for the Hopper architecture. A high-performance kernel for this matrix-multiplication overlaps computation with data movement by pipelining loads.

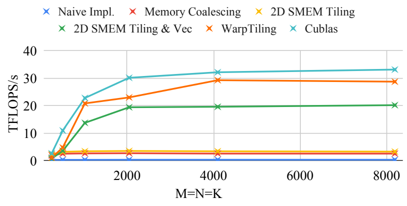

Fig. 17. Performance of **`sgemm`** on sqaure matrices as matrix dimension $M = N = K$ increases.

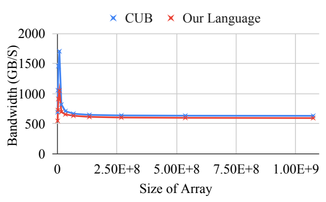

Fig. 18. Performance for prefix sum as input array size increases.

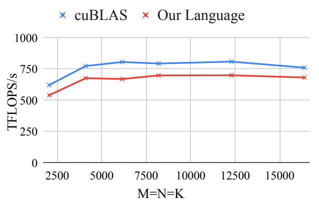

Fig. 19. Performance of **`hgemm`** on sqaure matrices as matrix dimension $M = N = K$ increases.

The implementation in Parallax looks different from normal CUDA code, particularly in how pipelining is expressed. Since Parallax uses *names* and subsequent **`partition`** or **`claim`** constructs to determine the synchronization each region requires, when pipelining, we cannot dynamically change the pipeline slot simply by maintaining an index variable that wraps around based on the pipeline length. Instead, each pipeline slot must be given a separate name so that Parallax can track them independently and actually overlap compute with data-movement. This leads to pipeline slots that must be individually named and forces the load logic to be effectively "unrolled," since we can no longer iterate over a pipeline-slot index. In turn, this forces that all pipelines in Parallax be statically sized. In practice. these pipelines are statically sized in CUDA anyway because they occupy shared memory, which is a small, finite resource that must be explicitly managed.

Notably, for this benchmark, in addition to **`wgmma`** and TMA, the program needs to dynamically reallocate registers between producer and consumer warpgroups, an instrutcion only avaialble for the Hopper. This redistribution is a warproup-level collective operation, and Parallax can check it like any other collective operation. 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 Parallax must eventually insert the appropriate synchronization for these transfers.

6.3 RQ3: Can Parallax 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, expressed operations that rely on multiple points of convergence, and used advanced GPU features. We now discuss the performance of these programs.

The performance of the matrix multiplication benchmarks is shown in , where we demonstrate that Parallax is competitive with cuBLAS [8].

For the prefix scan, we compare our performance to , the library introduced earlier in Section 1, and show that we can reach approximately % of CUB's peak bandwidth.

Finally, we evaluate our H100 implementation and show that it is competitive with cuBLAS on square sizes, falling between %. We emphasize that this is an exacting benchmark, and achieving this level of performance requires writing large kernels with precise control over low-level features.

7 Related Work

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

Imperative Languages for GPUs. CUDA [23], ROCm [2], and OpenCL [36] are imperative, C-style programming languages that expose low-level access to GPU hardware. These languages do not model perspectives for code or data, statically or otherwise; instead, programmers must orchestrate computation on the machine by describing how individual threads execute code. The subtle and often silent failures permitted by this model are the motivation for Parallax.

Descend [20], a new, Rust-inspired 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, while ours is ensuring that collective operations execute in valid contexts. Moreover, 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 would require major changes to the language. 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 [19], Accelerate [6] and Vertigo [14] 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. Parallax, however, makes an intentional decision to expose 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 [38], Pallas [35], Tilus [13], Helion [28]—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 vantage point in the GPU compute hierarchy: a block (Triton, Tilus, Helion), or a warpgroup (Pallas). Gluon [39] and is a new low-level, tile-based language. However, 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 [41], Halide [29], Fireiron [16] and RISE/ELEVATE [17] expose a mapping specification or scheduling language that lets users modify an existing reference program through external commands. Because they operate by transforming a fixed source-of-truth rather than generating the program directly, they expose a fundamentally different programming model than Parallax.

Libraries Built on CUDA. Many GPU libraries, such as CUTLASS [34], CUB [22], and ThunderKittens [30], 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 with a compiler. 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, leaving room for fragile errors.

Theoretical Foundations. The design of Prism 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 [11]. In Prism, 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 Prism's type system.

8 Conclusion, Limitations, and Future Work

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

Currently, Parallax does not natively support explicit references to pointers; programmers must rely on built-in constructs to interact with memory. Prior work such as Descend [20], which builds on Rust's ownership model [33], provides a promising direction for extending Parallax with more flexible pointer semantics.

We also aim to improve the ergonomics of programming with Parallax, particularly by enhancing the pipelining experience. As discussed in Section 6.2, Parallax currently requires pipeline slots to be explicitly name; we can improve this by adding language support for generating pipeline-style code.

Parallax 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 the model presented here to hierarchical compute environments, including distributed systems.

On the theoretical side, we plan to explore a terminating fragment of Prism 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. [40] and Spies et al. [31, 32].

We believe that Parallax 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] 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

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

[6] 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

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

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

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

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

[11] 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

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

[13] 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

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

[15] 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

[16] 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

[17] 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

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

[19] 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

[20] 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

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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

[30] 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

[31] 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

[32] 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

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

[34] 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

[35] 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.

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

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

[38] 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

[39] 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.

[40] 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

[41] 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

A Complete Prism Type System, Semantics, and Syntactic Soundness Proofs

A.1 Basic Definitions

$$\begin{aligned}
\text{Hierarchy Levels } h &::= \texttt{Grid} \,|\, \texttt{Block} \,|\, \texttt{Thread} \\
\text{Memory Kinds } l &::= \texttt{Local} \,|\, \texttt{Shared} \,|\, \texttt{Global} \\
\text{Perspectives } \pi &: h \times \mathbb{N} \\
\text{Base Types } b &::= \texttt{bool} \,|\, \texttt{int} \,|\, \texttt{float} \\
\text{Types } \tau &::= b \,|\, b[\,]^l \,|\, \textbf{Fun}(\Gamma,\pi,m) \,|\, \texttt{async}\ \tau \\
\text{Contexts } \Gamma &::= \cdot \,|\, \Gamma, x :^{\pi} \tau \\
\text{Shared Memory Remaining } m &: \mathbb{N}
\end{aligned}$$

Perspectives π are part of an algebra parameterized over some constant values T (the number of threads per block) and B (the number of blocks per grid). With these values, we have two isomorphisms:

$$(\texttt{Block},1) \cong (\texttt{Thread},T)$$

and

$$(\texttt{Grid},1) \cong (\texttt{Block},B)$$

The **`group`** and **`split`** operations of Parallax allows us to move along this isomorphism, from left to right. For clarity, in Prism we split these operations into three: a **`split`** operation that can split perspectives into multiple narrower ones, and a **`destruct`** operation that directly moves us along the isomorphism, and a **`group`** that divides our current perspective into equal sized parts.

Perspectives π are also lexicographically ordered in the obvious way. hs are ordered such that $\texttt{Thread} \leq \texttt{Block} \leq \texttt{Grid}$, and $(h_1,n_1) \leq (h_2,n_2)$ iff $n_1 | n_2$ and $h_1 \leq h_2$.

We define scalar multiplication $i \times h$ of natural numbers with hs: $i \times (h,n) = (h,in)$.

We also define division of perspectives and hierarchy levels of type $\pi \times \pi \to \mathbb{N}$. $\texttt{Grid}/\texttt{Block} = B$ and $\texttt{Block}/\texttt{Thread} = T$. We lift this to perspectives like so: $(h_1,n_1)/(h_2,n_2) = ((h_1/h_2) \cdot n_1)/n_2$.

Lastly we define a partial $\downarrow$ operator on hs such that $\downarrow \texttt{Block} = \texttt{Thread}$ and $\downarrow \texttt{Grid} = \texttt{Block}$. Note that $\downarrow \texttt{Thread}$ is undefined. This operator lifts to πs whose second component is 1 and encodes the leftward component of the isomorphism above: $\downarrow (\texttt{Block},1) = (\texttt{Thread},T)$ and $\downarrow (\texttt{Grid},1) = (\texttt{Block},B)$.

Note also that the presentation of these rules in the main body of the paper elide the m portion, which tracks the maximum amount of memory a given computation is allowed to use. In the full system presented here, both the typing rules and the operational semantics carry an additional piece of information tracking allocated memory.

A.2 Complete Prism Typing Rules

A.2.1 Expressions.

$$\frac{x :^{\pi} \tau \in \Gamma}{\Gamma \vdash^{\pi} x : \tau}\ \textbf{T-Var} \qquad \frac{}{\Gamma \vdash^{\pi} n : \texttt{int}}\ \textbf{T-Int} \qquad \frac{}{\Gamma \vdash^{\pi} f : \texttt{float}}\ \textbf{T-Float}$$

$$\frac{}{\Gamma \vdash^{\pi} b : \texttt{bool}}\ \textbf{T-Bool} \qquad \frac{\pi < (\texttt{Grid},1)}{\Gamma \vdash^{\pi} \texttt{partition\_id} : \texttt{int}}\ \textbf{T-Partition-Id}$$

$$\frac{\Gamma \vdash^{\pi'} e_1 : \tau[\,]^l \quad \Gamma \vdash^{\pi} e_2 : \texttt{int} \quad l = \texttt{Global or } l = \texttt{Local} \quad \pi \leq \pi'}{\Gamma \vdash^{\pi} e_1[e_2] : \tau}\ \textbf{T-Arr-Access}$$

$$\frac{\Gamma\vdash^{\pi'} e_1:\tau[]^{\texttt{Shared}} \quad \Gamma\vdash^{\pi} e_2:\texttt{int} \quad \pi\le(\texttt{Block},1) \quad \pi\le\pi'}{\Gamma\vdash^{\pi} e_1[e_2]:\tau}\ \textbf{T-Arr-Access-Shared}$$

$$\frac{\Gamma\vdash^{\pi} e_1:\texttt{int} \quad \Gamma\vdash^{\pi} e_2:\texttt{int}}{\Gamma\vdash^{\pi} e_1\ \texttt{bop}\ e_2:\texttt{int}}\ \textbf{T-Bop} \qquad \frac{\Gamma\vdash^{\pi} e_1:\texttt{int} \quad \Gamma\vdash^{\pi} e_2:\texttt{int}}{\Gamma\vdash^{\pi} e_1\ \texttt{cmp}\ e_2:\texttt{bool}}\ \textbf{T-Cmp}$$

A.2.2 Statements.

$$\frac{f:^{\pi}\mathbf{Fun}(x_i:\tau_i,\pi,m')\in\Gamma \quad \Gamma\vdash^{\pi} e_i:\tau_i \quad m'\le m}{\Gamma\vdash^{\pi}_{m} f(e_1,\ldots,e_n)}\ \textbf{T-Function-Call}$$

$$\frac{\Gamma\vdash^{(h,n_1)}_{m} s_1 \quad \Gamma\vdash^{(h,n_2)}_{m} s_2 \quad n_1,n_2\ \textbf{align to}\ n}{\Gamma\vdash^{(h,n)}_{m} \texttt{split}(n_1,n_2)\{s_1\}\{s_2\}}\ \textbf{T-Split} \qquad \frac{\Gamma\vdash^{\downarrow\pi}_{m} s}{\Gamma\vdash^{\pi}_{m} \texttt{destruct in}\ s}\ \textbf{T-Destruct}$$

$$\frac{\Gamma\vdash^{(h,n)}_{m} s}{\Gamma\vdash^{(h,q\cdot n)}_{m} \texttt{group}\ q\ s}\ \textbf{T-Group} \qquad \frac{x:^{\pi'}\tau\in\Gamma \quad \Gamma\vdash^{\pi} e:\tau \quad \pi'\le\pi}{\Gamma\vdash^{\pi}_{m} x=e}\ \textbf{T-Assn}$$

$$\frac{}{\Gamma\vdash^{\pi}_{m}\texttt{init}_{\psi}}\ \textbf{T-Sync-Init} \qquad \frac{}{\Gamma\vdash^{\pi}_{m}\texttt{dec}_{\psi}}\ \textbf{T-Sync-Dec}$$

$$\frac{}{\Gamma\vdash^{\pi}_{m}\texttt{wait}_{\psi}}\ \textbf{T-Sync-Wait} \qquad \frac{}{\Gamma\vdash^{\pi}_{m}\texttt{skip}}\ \textbf{T-Skip} \qquad \frac{n\le m}{\Gamma\vdash^{\pi}_{m}\texttt{free}\ n}\ \textbf{T-Free}$$

$$\frac{\Gamma\vdash^{\pi} e:\tau \quad \Gamma,x:^{\pi'}\tau\vdash^{\pi}_{m} s \quad \pi'\le\pi \quad \tau\ \text{not an array type}}{\Gamma\vdash^{\pi}_{m} x:\tau\ \texttt{@}\ \pi'=e\ \texttt{in}\ s}\ \textbf{T-Decl}$$

$$\frac{\Gamma\vdash^{\pi'} e_1:\tau[]^{l} \quad \Gamma\vdash^{\pi} e_2:\texttt{int} \quad \Gamma\vdash^{\pi} e_3:\tau \quad l=\texttt{Global or}\ l=\texttt{Local} \quad \pi'\le\pi}{\Gamma\vdash^{\pi}_{m} e_1[e_2]=e_3}\ \textbf{T-Arr-Assn}$$

$$\frac{\Gamma\vdash^{\pi'} e_1:\tau[]^{\texttt{Shared}} \quad \Gamma\vdash^{\pi} e_2:\texttt{int} \quad \Gamma\vdash^{\pi} e_3:\tau \quad \pi\le(\texttt{Block},1) \quad \pi'\le\pi}{\Gamma\vdash^{\pi}_{m} e_1[e_2]=e_3}\ \textbf{T-Arr-Assn-Shared}$$

$$\frac{\Gamma\vdash^{\pi} e:\texttt{bool} \quad \Gamma\vdash^{\pi}_{m_1} s_1 \quad \Gamma\vdash^{\pi}_{m_2} s_2}{\Gamma\vdash^{\pi}_{\mathbf{max}(m_1,m_2)} \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}\ \textbf{T-If}$$

$$\frac{\Gamma\vdash^{\pi} e:\texttt{bool} \quad \Gamma\vdash^{\pi}_{m} s}{\Gamma\vdash^{\pi}_{m}\texttt{while}\ e\ \texttt{do}\ s}\ \textbf{T-While} \qquad \frac{\Gamma\vdash^{\pi}_{m_1} s_1 \quad \Gamma\vdash^{\pi}_{m_2} s_2}{\Gamma\vdash^{\pi}_{\mathbf{max}(m_1,m_2)} s_1;s_2}\ \textbf{T-Seq}$$

$$\frac{\Gamma,x:^{\pi}\tau[]^{l}\vdash^{\pi}_{m} s \quad l=\texttt{Global or}\ l=\texttt{Local}}{\Gamma\vdash^{\pi}_{m+n\cdot\texttt{size}(\tau)} x:=\texttt{alloc}\ l\ \tau\ n\ \texttt{in}\ s}\ \textbf{T-Alloc}$$

$$\frac{\Gamma,x:^{(\texttt{Block},1)}\tau[]^{\texttt{Shared}}\vdash^{\pi}_{m} s}{\Gamma\vdash^{(\texttt{Block},1)}_{m+n\cdot\texttt{size}(\tau)} x:=\texttt{alloc Shared}\ \tau\ n\ \texttt{in}\ s}\ \textbf{T-Alloc-Shared}$$

$$\frac{\Gamma,y:^{(h,n/c)}\tau[]^{l}\vdash^{(h,n)}_{m} s \quad c|n \quad l\neq\texttt{Local}}{\Gamma,x:^{(h,n)}\tau[]^{l}\vdash^{(h,n)}_{m}\texttt{partition}_{\psi}\ x\ \texttt{into}\ y\ \texttt{by}\ c\ \texttt{in}\ s}\ \textbf{T-Partition}$$

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

$$\frac{\Gamma, y :^{\downarrow\pi} \tau[]^{l} \vdash^{\pi}_{m} s \quad l \neq \texttt{Local}}{\Gamma, x :^{\pi} \tau[]^{l} \vdash^{\pi}_{m} \texttt{lower}_{\psi}\ x\ \texttt{into}\ y\ \texttt{in}\ s}\ \textbf{T-Lower}$$

$$\frac{\Gamma, y :^{(\texttt{Thread},1)} \texttt{async}\ \ \tau[]^{l} \vdash^{(\texttt{Thread},1)}_{m} s}{\Gamma, x :^{(\texttt{Thread},1)} \tau[]^{l} \vdash^{(\texttt{Thread},1)}_{m} \texttt{async\_partition}_{\phi}\ x\ \texttt{into}\ y\ \texttt{in}\ s}\ \textbf{T-Async-Partition}$$

$$\frac{}{\Gamma, x :^{(\texttt{Thread},1)} \texttt{async}\ \ \tau[]^{l}, y :^{(\texttt{Thread},1)} \tau[]^{l'} \vdash^{(\texttt{Thread},1)}_{m} \texttt{async\_mempcy}(x,y)}\ \textbf{T-Async-Memcpy}$$

$$\frac{}{\Gamma, x :^{\pi} \tau[]^{l}, y :^{\pi} \tau[]^{l'} \vdash^{\pi}_{m} \texttt{memcpy}(x,y)}\ \textbf{T-Memcpy}$$

A.3 Complete Prism Semantics

A.3.1 Definitions.

$$\begin{aligned}
\text{Global Memory } \Sigma &::= \cdot \mid \Sigma, n \mapsto^{\pi} v \\
\text{Shared Memory } \sigma &::= \cdot \mid \sigma, n \mapsto^{\pi} v \\
\text{Local Memory } \eta &::= \cdot \mid \eta, n \mapsto^{\pi} v
\end{aligned}$$

$$\begin{aligned}
\text{Block Memory Map } S &::= \forall n \in B, n \mapsto \sigma \\
\text{Thread Memory Map } L &::= \forall n \in T, n \mapsto \eta
\end{aligned}$$

$$\begin{aligned}
\text{Synchronization Map } \Psi &: \psi \to \mathbb{N} \to \mathbb{N} \\
\text{Deferred Computations Map } \Phi &: \phi \to \{s\}
\end{aligned}$$

In real GPUs, thread IDs are only unique within their block. However, in this calculus for simplicity we assume thread IDs are global. One can convert back and forth between this abstracted notion of a thread ID and a block-unique ID via addition modulo T. That is, $t_{\text{real}} = t_{\text{simplified}} \textbf{ mod } T$ and $t_{\text{simplified}} = t_{\text{real}} + b \cdot T$.

By convention the names for local and shared and global memory do not conflict, as on the GPU they will be separate pointer spaces. Additionally, we freely interchange between using names for variables and integer locations.

In the main body of the paper, for simplicity we elide the synchronization map and the deferred computation map from the operational semantics, as our theorems do not make any guarantees about non-interference. However, as they are part of the full semantics, we include them here for completeness. By convention the synchronization and deferred computation maps are a total functions, initialized to map to $\lambda\_.0$ for ψs and $\lambda\_.\{\}$ for ϕs not explicitly initialize.

The shape of the judgment for a single thread is $\eta,\sigma,\Sigma,t,b,p,\Psi \vdash^{\pi}_{m} s \rightsquigarrow s' \dashv_{m'} \eta',\sigma',\Sigma',\Psi'$. The t here is the thread ID, the b is the block ID, and the p is the perspective ID. The last of these three is modified and managed by the rules for **`split`**, **`group`** and **`destruct`**, and tracks the relative position of the thread within a group. This semantics is in a small step style.

The shape of the judgment for expressions is $\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} e \Downarrow v$. The two πs represent the ambient compute context (i.e., the context in which resources are being read), while π', represents the target compute context (i.e, the compute context of the variable into which the result of the expression is going to be written. This is relevant for computing the value of **`partition_id`**, which divides the two contexts. As a shorthand, we can divide a perspective by a scalar value like so: $(h,n)/c = (h,n)/(h,c)$.

The overall evaluation of a program is expressed as

$$L,S,\Sigma,P,\Psi,\Phi \rightsquigarrow L',S',\Sigma',P',\Psi',\Phi'.$$

In this judgment P serves as a *thread pool*, mapping pairs of thread and block IDs (which don't change) to statements and memory (which can be updated by stepping). One can think of P as tracking which program is running on each thread. This steps according to the following rule:

$$\frac{L(t),S(b),\Sigma,t,b,0,\Psi,\Phi \vdash_m^{(\texttt{Grid},1)} s \rightsquigarrow s' \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi' \quad P(t,b)=(s,m)}{L,S,\Sigma,P,\Psi,\Phi \rightsquigarrow L[t\mapsto\eta'],S[b\mapsto\sigma'],\Sigma',P[(t,b)\mapsto(s',m')],\Psi',\Phi'}\ \textbf{S-Program}$$

For simplicity of notation, we define an **update** operation that searches the three environments for the one that contains the variable being used (by convention, there is no conflict between the environments, as in reality they exist in three separate address spaces). We also define a similar **get** operation that retrieves a variable from memory, and a **rename** operation that remaps a variable with the same value but under a different name.

$$\begin{aligned}
\textbf{update}(\eta,\sigma,\Sigma,x,v) &= (\eta[x\mapsto^{\pi} v],\sigma,\Sigma)\ \textit{when}\ x\in^{\pi}\eta\\
\textbf{update}(\eta,\sigma,\Sigma,x,v) &= (\eta,\sigma[x\mapsto^{\pi} v],\Sigma)\ \textit{when}\ x\in^{\pi}\sigma\\
\textbf{update}(\eta,\sigma,\Sigma,x,v) &= (\eta,\sigma,\Sigma[x\mapsto^{\pi} v])\ \textit{when}\ x\in^{\pi}\Sigma
\end{aligned}$$

$$\begin{aligned}
\textbf{get}(\eta,\sigma,\Sigma,x) &= \eta(x)\ \textit{when}\ x\in^{\pi}\eta\\
\textbf{get}(\eta,\sigma,\Sigma,x) &= \sigma(x)\ \textit{when}\ x\in^{\pi}\sigma\\
\textbf{get}(\eta,\sigma,\Sigma,x) &= \Sigma(x)\ \textit{when}\ x\in^{\pi}\Sigma
\end{aligned}$$

$$\begin{aligned}
\textbf{rename}(\eta,\sigma,\Sigma,x,y,\pi') &= (\eta[y\mapsto^{\pi'}\eta(x)],\sigma,\Sigma)\ \textit{when}\ x\in^{\pi}\eta\\
\textbf{rename}(\eta,\sigma,\Sigma,x,y,\pi') &= (\eta,\sigma[y\mapsto^{\pi'}\sigma(x)],\Sigma)\ \textit{when}\ x\in^{\pi}\sigma\\
\textbf{rename}(\eta,\sigma,\Sigma,x,y,\pi') &= (\eta,\sigma,\Sigma[y\mapsto^{\pi'}\Sigma(x)])\ \textit{when}\ x\in^{\pi}\Sigma
\end{aligned}$$

A.3.2 *Perspective Management Rules.*

$$\frac{p<n_1 \quad n_1,n_2\ \textbf{align to}\ n \quad \eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_m^{(h,n_1)} s_1 \rightsquigarrow s_1' \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_m^{(h,n)} \texttt{split}(n_1,n_2)\{s_1\}\{s_2\} \rightsquigarrow \texttt{split}(n_1,n_2)\{s_1'\}\{s_2\} \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'}\ \textbf{S-Split-Left}$$

$$\frac{p<n_1 \quad n_1,n_2\ \textbf{align to}\ n}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_m^{(h,n)} \texttt{split}(n_1,n_2)\{\texttt{skip}\}\{s_2\} \rightsquigarrow \texttt{skip} \dashv_m \eta,\sigma,\Sigma,\Psi,\Phi}\ \textbf{S-Split-Left-Done}$$

$$\frac{p\ge n_1 \quad p<n_1+n_2 \quad n_1,n_2\ \textbf{align to}\ n \quad \eta,\sigma,\Sigma,t,b,p-n_1,\Psi,\Phi \vdash_m^{(h,n_2)} s_2 \rightsquigarrow s_2' \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_m^{(h,n)} \texttt{split}(n_1,n_2)\{s_1\}\{s_2\} \rightsquigarrow \texttt{split}(n_1,n_2)\{s_1\}\{s_2'\} \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'}\ \textbf{S-Split-Right}$$

$$\frac{p\ge n_1 \quad p<n_1+n_2 \quad n_1,n_2\ \textbf{align to}\ n}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_m^{(h,n)} \texttt{split}(n_1,n_2)\{s_1\}\{\texttt{skip}\} \rightsquigarrow \texttt{skip} \dashv_m \eta,\sigma,\Sigma,\Psi,\Phi}\ \textbf{S-Split-Right-Done}$$

$$\frac{p\ge n_1+n_2 \quad n_1,n_2\ \textbf{align to}\ n}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_m^{(h,n)} \texttt{split}(n_1,n_2)\{s_1\}\{s_2\} \rightsquigarrow \texttt{skip} \dashv_m \eta,\sigma,\Sigma,\Psi,\Phi}\ \textbf{S-Split-None}$$

$$\frac{\eta,\sigma,\Sigma,t,b,t \ \mathbf{mod}\ T,\Psi,\Phi \vdash_{m}^{(\texttt{Thread},T)} s \rightsquigarrow s' \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'}{\eta,\sigma,\Sigma,t,b,0,\Psi,\Phi \vdash_{m}^{(\texttt{Block},1)} \texttt{destruct in } s \rightsquigarrow \texttt{destruct in } s' \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'} \ \textbf{S-Destruct-Block}$$

$$\frac{\eta,\sigma,\Sigma,t,b,b \ \mathbf{mod}\ B,\Psi,\Phi \vdash_{m}^{(\texttt{Block},B)} s \rightsquigarrow s' \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'}{\eta,\sigma,\Sigma,t,b,0,\Psi,\Phi \vdash_{m}^{(\texttt{Grid},1)} \texttt{destruct in } s \rightsquigarrow \texttt{destruct in } s' \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'} \ \textbf{S-Destruct-Grid}$$

$$\frac{}{\eta,\sigma,\Sigma,t,b,0,\Psi,\Phi \vdash_{m}^{\pi} \texttt{destruct in skip} \rightsquigarrow \texttt{skip} \dashv_{m} \eta,\sigma,\Sigma,\Psi,\Phi} \ \textbf{S-Destruct-Done}$$

$$\frac{\eta,\sigma,\Sigma,t,b,p \ \mathbf{mod}\ n,\Psi,\Phi \vdash_{m}^{(h,n)} s \rightsquigarrow s' \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_{m}^{(h,q\cdot n)} \texttt{group } q \ s \rightsquigarrow \texttt{group } q \ s'; \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'} \ \textbf{S-Group}$$

$$\frac{}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_{m}^{\pi} \texttt{group } q \ \texttt{skip} \rightsquigarrow \texttt{skip}; \dashv_{m} \eta,\sigma,\Sigma,\Psi,\Phi} \ \textbf{S-Group-Done}$$

A.3.3 *Thread Synchronization.*

We define a **size** operation on perspectives to compute the size of a perspective (the number of individual compute resources sharing it). The operation is defined thusly:

$$\begin{aligned} \mathbf{size}(\texttt{Thread},n) &= n \\ \mathbf{size}(\texttt{Block},n) &= n\cdot T \\ \mathbf{size}(\texttt{Grid},n) &= n\cdot B\cdot T \end{aligned}$$

$$\frac{\Psi(\psi)(p)=0}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_{m}^{\pi} \texttt{wait}_{\psi} \rightsquigarrow \texttt{skip} \dashv_{m} \eta,\sigma,\Sigma,\Psi,\Phi} \ \textbf{S-Sync-Wait-Done}$$

$$\frac{\Psi(\psi)(p)\neq 0}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_{m}^{\pi} \texttt{wait}_{\psi} \rightsquigarrow \texttt{wait}_{\psi} \dashv_{m} \eta,\sigma,\Sigma,\Psi,\Phi} \ \textbf{S-Sync-Wait-Spin}$$

$$\frac{\Psi'=\Psi(\psi)[p\mapsto \Psi(\psi)(p)-1]}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_{m}^{\pi} \texttt{dec}_{\psi} \rightsquigarrow \texttt{skip} \dashv_{m} \eta,\sigma,\Sigma,\Psi',\Phi} \ \textbf{S-Sync-Dec}$$

$$\frac{\Psi(\psi)(p)=0 \quad \Psi'=\Psi(\psi)[p\mapsto \mathbf{size}(\pi)]}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_{m}^{\pi} \texttt{init}_{\psi} \rightsquigarrow \texttt{skip} \dashv_{m} \eta,\sigma,\Sigma,\Psi',\Phi} \ \textbf{S-Sync-Init-Zero}$$

$$\frac{\Psi(\psi)(p)\neq 0}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_{m}^{\pi} \texttt{init}_{\psi} \rightsquigarrow \texttt{skip} \dashv_{m} \eta,\sigma,\Sigma,\Psi,\Phi} \ \textbf{S-Sync-Init-Nonzero}$$

A.3.4 *Asynchrony.*

$$\frac{\mathbf{rename}(\eta,\sigma,\Sigma,x,y,(\texttt{Thread},1)),t,b,p,\Psi,\Phi \vdash_{m}^{(\texttt{Thread},1)} s \rightsquigarrow s' \dashv_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'}{\begin{array}{c}\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_{m'}^{(\texttt{Thread},1)} \texttt{async\_partition}_{\phi} \ x \ \texttt{into} \ y \ \texttt{in} \ s \rightsquigarrow \\ \texttt{async\_partition}_{\phi} \ x \ \texttt{into} \ y \ \texttt{in} \ s' \dashv_{m} \eta',\sigma',\Sigma',\Psi',\Phi'\end{array}} \ \textbf{S-Async-Partition-Congr}$$

$$\frac{\Phi=\Phi'[\phi\mapsto \Phi'(\phi)\cup\{s\}]}{\begin{array}{c}\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash_{m}^{(\texttt{Thread},1)} \texttt{async\_partition}_{\phi} \ x \ \texttt{into} \ y \ \texttt{in skip} \rightsquigarrow \\ (\texttt{async\_partition}_{\phi} \ x \ \texttt{into} \ y \ \texttt{in} \ s) \dashv_{m} \eta,\sigma,\Sigma,\Psi,\Phi'\end{array}} \ \textbf{S-Async-Partition-Unwind}$$

$$\frac{\Phi(\phi)=\emptyset}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{(\texttt{Thread},1)}\texttt{async\_partition}_\phi\ x\ \texttt{into}\ y\ \texttt{in}\ \texttt{skip}\rightsquigarrow \texttt{skip}\dashv_m \eta,\sigma,\Sigma,\Psi,\Phi}\ \textbf{S-Async-Partition-Done}$$

$$\frac{}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{(\texttt{Thread},1)}\texttt{async\_mempcy}(x,y)\rightsquigarrow \texttt{skip}\dashv_m \eta,\sigma,\Sigma,\Psi,\Phi[\phi\mapsto\Phi(\phi)\cup\{\texttt{memcpy}(x,y)\}]}\ \textbf{S-Async-Memcpy}$$

$$\frac{(\eta',\sigma',\Sigma')=\textbf{update}(\eta,\sigma,\Sigma,x,\textbf{get}(\eta,\sigma,\Sigma,y))}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi}\texttt{memcpy}(x,y)\rightsquigarrow\texttt{skip}\dashv_m\eta',\sigma',\Sigma',\Psi,\Phi}\ \textbf{S-Memcpy}$$

A.3.5 *Variables and Memory.*

$$\frac{\eta,\sigma,\Sigma\vdash_{\pi'}^{\pi} e\Downarrow v\qquad \pi'\le\pi}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi} x:\tau\ \texttt{@}\ \pi':=e\ \texttt{in}\ s\rightsquigarrow s\dashv_m \eta[x\mapsto^{\pi'} v],\sigma,\Sigma,\Psi,\Phi}\ \textbf{S-Decl}$$

$$\frac{}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi}\texttt{free}\ \ n\dashv_{m-n}\eta,\sigma,\Sigma,\Psi,\Phi}\ \textbf{S-Free}$$

$$\frac{}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi} x:=\texttt{alloc Local}\ \tau\ n\ \texttt{in}\ s\rightsquigarrow s;\ \texttt{free}\ \ (n\cdot\text{size}(\tau))\dashv_{m+n\cdot\text{size}(\tau)}\eta[x\mapsto^{\pi}\langle x,n\rangle],\sigma,\Sigma,\Psi,\Phi}\ \textbf{S-Alloc-Local}$$

$$\frac{\pi=(\texttt{Block},1)}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi} x:=\texttt{alloc Shared}\ \tau\ n\ \texttt{in}\ s\rightsquigarrow s;\ \texttt{free}\ \ (n\cdot\text{size}(\tau))\dashv_{m+n\cdot\text{size}(\tau)}\eta,\sigma[x\mapsto^{\pi}\langle x,n\rangle],\Sigma,\Psi,\Phi}\ \textbf{S-Alloc-Shared}$$

$$\frac{}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi} x:=\texttt{alloc Global}\ \tau\ n\ \texttt{in}\ s\rightsquigarrow s;\ \texttt{free}\ \ (n\cdot\text{size}(\tau))\dashv_{m+n\cdot\text{size}(\tau)}\eta,\sigma,\Sigma[x\mapsto^{\pi}\langle x,n\rangle],\Psi,\Phi}\ \textbf{S-Alloc-Global}$$

$$\frac{x\in^{\pi'}\eta,\sigma,\Sigma\qquad \eta,\sigma,\Sigma\vdash_{\pi'}^{\pi} e\Downarrow v\qquad (\eta',\sigma',\Sigma')=\textbf{update}(\eta,\sigma,\Sigma,x,v)\qquad \pi'\le\pi}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi,\Phi\vdash_m^{\pi} x=e\rightsquigarrow\texttt{skip}\dashv_m\eta',\sigma',\Sigma',\Psi,\Phi}\ \textbf{S-Assn}$$

$$\frac{\begin{array}{cc}\eta,\sigma,\Sigma,t,b,p\vdash_{\pi}^{\pi} e_1\Downarrow\langle l,n\rangle & i<n\quad \pi'\le\pi\\ \eta,\sigma,\Sigma,t,b,p\vdash_{\pi}^{\pi} e_2\Downarrow i & x\in^{\pi'}\eta,\sigma,\Sigma\\ \eta,\sigma,\Sigma\vdash_{\pi'}^{\pi} e_3\Downarrow v & (\eta',\sigma',\Sigma')=\textbf{update}(\eta,\sigma,\Sigma,l+i,v)\end{array}}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi} e_1[e_2]=e_3\rightsquigarrow\texttt{skip}\dashv_m\eta',\sigma',\Sigma',\Psi,\Phi}\ \textbf{S-Arr-Assn}$$

$$\frac{s'=s[(y+c\cdot p)/y]\qquad (\eta',\sigma',\Sigma')=\textbf{rename}(\eta,\sigma,\Sigma,x,y,\pi/c)}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi} partition\psi xycs\rightsquigarrow\texttt{init}_\psi;s';\texttt{dec}_\psi;\texttt{wait}_\psi\dashv_m\eta',\sigma',\Sigma',\Psi,\Phi}\ \textbf{S-Partition}$$

$$\frac{(\eta',\sigma',\Sigma')=\textbf{rename}(\eta,\sigma,\Sigma,x,y,(h,n_1))}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{(h,n_1+n_2)}\texttt{claim}_\psi\ x\ \texttt{into}\ y\ \texttt{at}\ n_1\ \texttt{in}\ s\rightsquigarrow \texttt{init}_\psi;\texttt{split}(n_1,n_2)\{s'\}\{\texttt{skip}\};\texttt{dec}_\psi;\texttt{wait}_\psi\dashv_m\eta',\sigma',\Sigma',\Psi,\Phi}\ \textbf{S-Claim}$$

$$\frac{(\eta',\sigma',\Sigma')=\textbf{rename}(\eta,\sigma,\Sigma,x,y,\downarrow\pi)}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi}\texttt{lower}_\psi\ x\ \texttt{into}\ y\ \texttt{in}\ s\rightsquigarrow\texttt{init}_\psi;s;\texttt{dec}_\psi;\texttt{wait}_\psi\dashv_m\eta',\sigma',\Sigma',\Psi,\Phi}\ \textbf{S-Lower}$$

A.3.6 Control Flow.

$$\frac{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi} e \Downarrow \texttt{true}}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m} \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \rightsquigarrow s_1 \dashv_m \eta,\sigma,\Sigma,\Psi,\Phi} \textbf{ S-If-True}$$

$$\frac{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi} e \Downarrow \texttt{false}}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m} \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \rightsquigarrow s_2 \dashv_m \eta,\sigma,\Sigma,\Psi,\Phi} \textbf{ S-If-False}$$

$$\frac{}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m} \texttt{while } e \texttt{ do } s \rightsquigarrow \texttt{if } e \texttt{ then } (s;\ \texttt{while } e \texttt{ do } s) \texttt{ else skip} \dashv_m \eta,\sigma,\Sigma,\Psi,\Phi} \textbf{ S-While}$$

$$\frac{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m} s_1 \rightsquigarrow s_1' \dashv^{\pi'}_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m} s_1;s_2 \rightsquigarrow s_1';s_2 \dashv^{\pi'}_{m'} \eta',\sigma',\Sigma',\Psi',\Phi'} \textbf{ S-Seq-First}$$

$$\frac{}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m} \texttt{skip};s_2 \rightsquigarrow s_2 \dashv_m \eta,\sigma,\Sigma,\Psi,\Phi} \textbf{ S-Seq-Done}$$

$$\frac{\Sigma(f) = \{[x_1:\tau_1,\ldots,x_n:\tau_n],s\} \quad \sigma,\Sigma \vdash^{\pi}_{\pi} e_i \Downarrow v_i \quad m' \le m}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m} f(e_1,\ldots,e_n) \rightsquigarrow s \dashv_m \eta[x_i \mapsto v_i],\sigma,\Sigma,\Psi,\Phi} \textbf{ S-Function-Call}$$

A.3.7 Expressions.

$$\frac{\pi < (\texttt{Grid},1) \quad \pi' \le \pi}{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} \texttt{partition\_id} \Downarrow \pi/\pi' - 1} \textbf{ E-Partition-Id}$$

$$\frac{}{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} x \Downarrow \textbf{get}(\eta,\sigma,\Sigma,x)} \textbf{ E-Var}$$

$$\frac{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} e_1 \Downarrow \langle l,n\rangle \quad \eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} e_2 \Downarrow i \quad i < n \quad \pi' \le \pi}{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} e_1[e_2] \Downarrow \textbf{get}(\eta,\sigma,\Sigma,l+i)} \textbf{ E-Arr-Access}$$

$$\frac{}{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} \vdash n \Downarrow n} \textbf{ E-Int} \qquad \frac{}{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} \vdash b \Downarrow b} \textbf{ E-Bool}$$

$$\frac{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} \vdash e_1 \Downarrow v_1 \quad \eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} e_2 \Downarrow v_2 \quad v = v_1 \texttt{ bop } v_2}{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} \vdash e_1 \texttt{ bop } e_2 \Downarrow v} \textbf{ E-Bop}$$

$$\frac{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} \vdash e_1 \Downarrow v_1 \quad \eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} e_2 \Downarrow v_2 \quad v = v_1 \texttt{ cmp } v_2}{\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} \vdash e_1 \texttt{ cmp } e_2 \Downarrow v} \textbf{ E-Cmp}$$

A.4 Theorems and Proofs

Note that in this section we assume no out of bounds array accesses. In general Parallax (and by extension Prism) makes no guarantees about array out of bounds.

A.4.1 More definitions. As a premise to our type safety theorems, we need to assume we have a well-typed environment, written $\Gamma \vdash \eta,\sigma,\Sigma$. We define what this means inductively

$$\frac{}{\eta,\sigma,\Sigma \vdash n : \texttt{int}} \textbf{ V-Int} \qquad \frac{}{\eta,\sigma,\Sigma \vdash b : \texttt{bool}} \textbf{ V-Bool} \qquad \frac{}{\eta,\sigma,\Sigma \vdash f : \texttt{float}} \textbf{ V-Float}$$

$$\frac{\forall i < n, \eta,\sigma,\Sigma \vdash \textbf{get}(\eta,\sigma,\Sigma,x+i) : \tau}{\eta,\sigma,\Sigma \vdash \langle x,n\rangle : \tau[]^{l}} \textbf{ V-Array} \qquad \frac{\Gamma, x_i :^{\pi} \tau_i \vdash^{\pi}_{m} s \quad \Gamma \vdash \cdot,\cdot,\textbf{fns}\ \Sigma}{\eta,\sigma,\Sigma \vdash \{x_i : \tau_i, s\} : \textbf{Fun}(x_i : \tau_i, \pi, m)} \textbf{ V-Function}$$

$$\frac{}{\cdot \vdash \eta,\sigma,\Sigma} \textbf{ G-Empty} \qquad \frac{\eta(x) =^{\pi} v \quad \eta,\sigma,\Sigma \vdash v : \texttt{int}}{\Gamma, x :^{\pi} \texttt{int} \vdash \eta,\sigma,\Sigma} \textbf{ G-Int}$$

$$\frac{\eta(x) =^{\pi} v \quad \eta,\sigma,\Sigma \vdash v : \texttt{bool}}{\Gamma, x :^{\pi} \texttt{bool} \vdash \eta,\sigma,\Sigma}\ \textbf{G-Bool} \qquad \frac{\eta(x) =^{\pi} v \quad \eta,\sigma,\Sigma \vdash v : \texttt{float}}{\Gamma, x :^{\pi} \texttt{float} \vdash \eta,\sigma,\Sigma}\ \textbf{G-Float}$$

$$\frac{\eta(x) =^{\pi} v \quad \eta,\sigma,\Sigma \vdash v : \tau[]}{\Gamma, x :^{\pi} \tau[]^{\texttt{Local}} \vdash \eta,\sigma,\Sigma}\ \textbf{G-Local} \qquad \frac{\sigma(x) =^{\pi} v \quad \eta,\sigma,\Sigma \vdash v : \tau[]}{\Gamma, x :^{\pi} \tau[]^{\texttt{Shared}} \vdash \eta,\sigma,\Sigma}\ \textbf{G-Shared}$$

$$\frac{\Sigma(x) =^{\pi} v \quad \eta,\sigma,\Sigma \vdash v : \tau[]}{\Gamma, x :^{\pi} \tau[]^{\texttt{Global}} \vdash \eta,\sigma,\Sigma}\ \textbf{G-Global} \qquad \frac{\Sigma(x) =^{\pi} v \quad \eta,\sigma,\Sigma \vdash v : \textbf{Fun}(\Gamma',\pi,m)}{\Gamma, x :^{\pi} \textbf{Fun}(\Gamma',\pi,m) \vdash \eta,\sigma,\Sigma}\ \textbf{G-Function}$$

We can prove a couple simple lemmas about well-typed environments under operations like **rename**, **update**, and **get**.

Lemma A.1. *(Well-typed **get**) If $\Gamma \vdash \eta,\sigma,\Sigma$ and $x :^{\pi} \tau \in \Gamma$ then $\eta,\sigma,\Sigma \vdash \textbf{get}(\eta,\sigma,\Sigma,x) : \tau$.*

Lemma A.2. *(Well-typed **rename**) If $\Gamma, x :^{\pi} \tau \vdash \eta,\sigma,\Sigma$ then $\Gamma, x :^{\pi} \tau, y :^{\pi'} \tau \vdash \textbf{rename}(\eta,\sigma,\Sigma,x,y,\pi')$.*

Lemma A.3. *(Well-typed **update**) If $\Gamma \vdash \eta,\sigma,\Sigma$ and $\eta,\sigma,\Sigma \vdash v : \tau$ then $\Gamma, x :^{\pi} \tau \vdash \textbf{update}(\eta,\sigma,\Sigma,x,v)$.*

We also define a well-formedness precondition on p with respect to π:

$$(h,n) \vdash p ::= p < n$$

We also define well-formedness for the async stack:

$$\Gamma \vdash \Phi ::= \forall \phi, s \in \Phi(\phi), \Gamma \vdash_{m}^{(\texttt{Thread},1)} s$$

A.5 Proofs

Lemma A.4. *(Expression Safety) If $\Gamma \vdash^{\pi} e : \tau$ and $\Gamma \vdash \eta,\sigma,\Sigma$ and $\pi \vdash p$ and $\pi' \leq \pi$, then there is some v such that $\eta,\sigma,\Sigma \vdash^{\pi}_{\pi'} e \Downarrow v$ and $v : \tau$.*

Proof. This proof proceeds by induction on the typing relation for expressions. Despite the fact that this property implies termination, we do not need a logical relation to prove it because the expression language is very simple.

The **T-Int**, **T-Float**, and **T-Bool** cases are trivial, using the rules **E-Int**, **E-Bool** and **E-Float** to compute values. In the case for **T-Partition-Id**, the π premises of the typing rules and our assumption that $\pi' \leq \pi$ match the premises of the evaluation rule, so this rule is simple as well.

The cases for **T-Bop** and **T-Cmp** follow directly from the inductive hypotheses, assuming a valid and correctly implemented set of binary operators and comparators.

The only interesting cases are **T-Arr-Access** and **T-Arr-Access-Shared**.

In both cases our inductive hypotheses and inversion give us that $\pi' \leq \pi$, and e_1 evaluates to a $\langle x,n \rangle$, and that all the values between x and $x+n$ in the appropriate environment are typed at τ. We also know that e_2 evaluates to an integer i. We assume that all array accesses are in bounds, so $i < l$, which is sufficient to use the **E-Arr-Access** rule to complete this case, and the proof. □

Lemma A.5. *(Expression Determinism) If $\eta,\sigma,\Sigma,t,b,p \vdash^{\pi}_{\pi'} e \Downarrow v_1$ and $\eta,\sigma,\Sigma,t,b,p \vdash^{\pi}_{\pi'} e \Downarrow v_2$ then $v_1 = v_2$.*

Proof. Straightforward by induction on the semantic derivation. □

Lemma A.6. *(Expression Well-Typedness) If $\Gamma \vdash^{\pi}_{\pi'} e : \tau$ and $\Gamma \vdash \eta,\sigma,\Sigma$ and $\pi \vdash p$, and $\eta,\sigma,\Sigma,t,b,p \vdash^{\pi}_{\pi'} e \Downarrow v$, then $v : \tau$.*

Proof. By our expression safety lemma our well-typed expression must evaluate to a well-typed value v'. By our determinism lemma v' must be the same as v, so v is well-typed. □

Lemma A.7. *(Statement Progress) If $\Gamma \vdash \eta,\sigma,\Sigma$ and $\Gamma \vdash^{\pi}_{m} s$ and $\pi \vdash p$, then either s is* **`skip`** *or there is some s' such that $\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m'} s \rightsquigarrow s' \dashv_{m''} \eta',\sigma',\Sigma,\Psi',\Phi'$*

Proof. This proceeds by induction on the typing derivation.

Perspective Management Rules.

- Case **T-Split**:
  In this case we have by our assumption that $\pi \vdash p$ that $p < n$. We also have that n_1,n_2 **align to** n, so $n_1 + n_2 \leq n$. There are three cases to consider, then: when $p < n_1$, when $p \geq n_1$ and $p < n_1 + n_2$, and when $p \geq n_1 + n_2$.
  In the first case, we have by our inductive hypothesis that s is either `skip` or that it can step in an (h,n_1) context. In the former case we can use the **S-Split-Left-Done** rule and in the latter we can use the **S-Split-Left rule**.
  The second case is almost symmetric. The only additional work we have to do is to argue that $(h,n_2) \vdash p - n_1$, or equivalently that $p - n_1 < n_2$. This, however, is immediate from our assumption that $p < n_1 + n_2$.
  In the last case, we just use the **S-Split-None** rule to step to **`skip`**.
- Case **T-Destruct**
  By our inductive hypothesis, we know that s can step at $\downarrow\pi$ if $\downarrow\pi \vdash p$. The $\downarrow$ operation is only defined at (**`Block`**,1) or (**`Grid`**,1), so we only need to consider the cases where π is one of those.
  In the former case p becomes t **mod** T while $\downarrow\pi$ is (**`Thread`**,T). t **mod** $T < T$ for any t so this satisfies the requirement that $\pi \vdash p$, which lets us use our inductive hypothesis: s is either **`skip`** or can step. If it can step, we can use this to satisfy the premise of **S-Destruct-Block** to step in this case. If it is **`skip`**, then we use the rule **S-Destruct-Done** to step instead.
  The latter case is the same, except using the fact that b **mod** $B < B$ and the **S-Destruct-Grid** rule.
- Case **T-Group**
  In this case we have by assumption that $(h,q \cdot n) \vdash p$, i.e., that $p < q \cdot n$.
  In this case we have our IH that if $(h,n) \vdash p'$ for some p', then s is either **skip** or steps with (h,n) perspective with p' as our perspective ID.
  We choose p' to be p **mod** n. This is always $< n$, so $(h,n) \vdash p'$. This lets us use our IH to get that s is either **`skip`** (in which case we can use the **S-Group-Done** rule to step) or itself steps, which lets us use the **S-Group** rule to step.

Thread Synchronization Rules.

- Case **T-Sync-Wait**
  $\Psi(\psi)(p)$ is either zero or it is not. In the former case we use the **S-Sync-Wait-Done** rule and in the latter we use **S-Sync-Wait-Spin**.
- Case **T-Sync-Dec**
  We use the **S-Sync-Dec** rule to step.
- Case **T-Sync-Init**
  We use the **S-Sync-Init-Zero** or **S-Sync-Init-Nonzero** rules depending on whether $\Psi(\psi)(p)$ is zero or not.

Asynchrony Rules.

- Case **T-Async-Partition**
  In this case, we have via our IH that if $\Gamma, y :^{(\texttt{Thread},1)}$ **`async`** $\tau[]^{l} \vdash \eta',\sigma',\Sigma'$, then we can either step s with (**`Thread`**,1) perspective under environments η', σ', and Σ', or s is **`skip`**.

We have by assumption that $\Gamma, x :^{(\texttt{Thread},1)} \texttt{async}\ \tau[]^{l} \vdash \eta, \sigma, \Sigma$. By our environment renaming lemma, this gives us what we need to use our IH with $(\eta', \sigma', \Sigma')$ as $\textbf{rename}(\eta, \sigma, \Sigma, x, y, (\texttt{Thread}, 1))$. Thus s either steps or is `skip`. In the former case we can step with **S-Async-Partition-Congr**, and in the latter we can use either **S-Async-Partition-Unwind** or **S-Async-Partition-Done** depending on whether $\Phi(\phi)$ is empty or not.

- Case **T-Async-Memcpy**
  Immediate via use of the **S-Async-Memcpy** rule.
- Case **T-Memcpy**
  Immediate via use of the **S-Memcpy** rule.

Memory Rules.

- Case **T-Decl**
  Via our lemma about expression type safety and our hypothesis that e is well-typed, we obtain the premises necessary to use the **S-Decl** rule to step.
- Case **T-Arr-Assn**
  Each of e_1, e_2 and e_3 must evaluate to a well-typed value by the expression type safety lemma. In particular, both e_1 evaluates to some $\langle l, n \rangle$ and e_2 evaluates to some i. We assume all array accesses are in bounds, so this is sufficient to use the **S-Arr-Assn** rule to step.
- Case **T-Arr-Assn-Shared**
  Same as previous case.
- Case **T-Free**
  Trivial via the **S-Free** rule.
- Case **T-Partition**
  Trivial via the **S-Partition** rule.
- Case **T-Claim**
  Trivial via the **S-Claim** rule.
- Case **T-Lower**
  Trivial via the **S-Lower** rule.
- Case **T-Alloc**
  We assume that l is not `Shared`, so we can use the **S-Alloc-Local** or **S-Alloc-Global** rule, depending on whether l is `Local` or `Global`.
- Case **T-Alloc-Shared**
  Trivial via the **S-Alloc-Shared** rule.

Control Rules.

- Case **T-Skip**
  Trivial
- Case **T-While**
  Trivial, all while loops step via the **S-While** rule
- Case **T-If**
  By our proof of expression type safety, the expression e steps to either the boolean value true or false. We can thus use either the **S-If-True** or **S-If-False** rules to step.
- Case **T-Seq**
  In this case we know by our IH that s_1 is either `skip` or can step. In the former case we use the `S-Seq-Done` rule and in the latter we use the `S-Seq-First` rule.
- Case **T-Function-Call**
  In this case we know by our expression safety lemma that each of the arguments will evaluate to a well-typed value. We also have by assumption that f has a function type, which by inversion

on the **V-Function** rule tells us that it is a closure type. Additionally our assumption that $\Gamma \vdash \eta,\sigma,\Sigma$ tell us that Σ contains f at the same type that Γ does. These premises are sufficient to use the **S-Function-Call** rule.

□

Lemma A.8. *(Statement Preservation) If $\Gamma \vdash \eta,\sigma,\Sigma$ and $\Gamma \vdash^{\pi}_{m} s$ and $\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m'} s \rightsquigarrow s' \dashv_{m''} \eta',\sigma',\Sigma,\Psi',\Phi'$ and $\pi \vdash p$ and $m \geq m'$ and $\Gamma \vdash \Phi$, then there is some Γ' such that $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash^{\pi}_{m} s'$ and $\Gamma' \vdash \eta',\sigma',\Sigma'$ and $m \geq m''$ and $\Gamma' \vdash \Phi'$.*

Proof. We proceed by induction on the derivation of $\Gamma \vdash^{\pi}_{m} s$.

Perspective Management Rules.

- Case **T-Split**
  In this case we have by assumption that $(h,n) \vdash p$, $\Gamma \vdash \eta,\sigma,\Sigma$, and $\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m'}$ `split`$(n_1,n_2)\{s_1\}\{s_2\} \rightsquigarrow s' \dashv_{m''} \eta',\sigma',\Sigma,\Psi',\Phi'$. Our inductive hypotheses give us that if for any p, if $(h,n_1) \vdash p$ and s_1 steps with perspective ID p, or if $(h,n_2) \vdash p$ and s_2 steps with perspective ID p then their results are well typed.
  By inversion on our semantic derivation, we are in one of 5 cases.
  In the **S-Split-Left** case $p < n_1$ and s_1 steps to s'_1. This is sufficient to tell us that s'_1 is well-typed and the output environments of that relation η',σ', and Σ' are all well-typed by $\Gamma' \supseteq \Gamma$, and that the memory is properly bounded by the typing rules.
  We can thus use the **T-Split-Left** rule to conclude that the result of this case is well-typed.
  The **S-Split-Left-Done** rule is trivial via the **T-Skip** rule.
  The **Right** cases are symmetric, with the observation that when $p \geq n_1$ and $p < n_1+n_2$ then $p-n_1 < n_2$.
  The last **S-Split-None** rule is trivial via the **T-Skip** rule.
- Case **T-Destruct**
  In this case we have by assumption that $\downarrow\pi$ is defined, so π is either (`Block`,1) or (`Grid`,1). We also assume that $\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{\pi}_{m'}$ `destruct in` $s \rightsquigarrow s' \dashv_{m''} \eta',\sigma',\Sigma,\Psi',\Phi'$. We also have by our inductive hypothesis that for any p such that $\downarrow\pi \vdash p$ and s'' such that s steps to s'' at p, then that step preserved well-typedness.
  By inversion on the step relation, we are in one of three cases.
  If the rule used **S-Destruct-Block**, then we know that s steps to s'' and p is t **mod** T. (`Thread`,T) $\vdash t$ **mod** T for any t, so we can use our inductive hypothesis to conclude that the s'' stepped to by s is well-typed, as are its environments and memory usage. The **T-Destruct** rule then gives us our desired goal.
  The **S-Destruct-Grid** case proceeds similarly, while the **S-Destruct-Done** case is trivial.
- Case **T-Group**
  In this case we have by assumption that
  $\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi \vdash^{(h,q\cdot n)}_{m'}$ `group` $q\ s \rightsquigarrow s' \dashv_{m''} \eta',\sigma',\Sigma,\Psi',\Phi'$. We have by our inductive hypothesis that s steps to s'' at some perspective ID p' and $(h,n) \vdash p'$, then s'' is well typed, as are the other outputs of that step.
  By inversion on the step relation, we are in one of two cases. The **S-Group-Done** case is trivial, so we shall focus on the **S-Group** case. In this case we have that s steps to s'' at perspective ID p **mod** n. It is always the case that $(h,n) \vdash p$ **mod** n for any p, so we can use our inductive hypothesis to conclude that s'' is well-typed, as are its output environments and memory usage. From there, it is a simple application of the **T-Group** rule to conclude that `group` $q\ s'$ is well-typed, and to finish the case.

Thread Synchronization Rules. These rules are all trivial: with one exception all thread synchronization primitives step to skip without changing environment or memory, and are thus obviously well-typed. **S-Sync-Wait-Spin** does not produce skip, but it steps to the same statement as we already assumed typechecks in the premise of the lemma, so is straightforward nonetheless.

If we wanted to say something about deadlock freedom we'd have more work here, but we aren't doing that, so these rules are easy.

Asynchrony Rules.

- Case **T-Async-Partition** In this case we have that s is well-typed in a context where x has been renamed into y, with $(\texttt{Thread},1)$ perspective. We also have that $\Gamma,x:^{(\texttt{Thread},1)}\tau[]^l \vdash \eta,\sigma,\Sigma$ and $\Gamma,x:^{(\texttt{Thread},1)}\tau[]^l\Phi$.
  By inversion, we are in one of three cases.
  In the **S-Async-Partition-Done** case, we are done.
  In the **S-Async-Partition-Congr case**, our inductive hypothesis gives us that there is some Γ' such that $\Gamma,y:^{(\texttt{Thread},1)}\texttt{async}\ \ \tau[]^l \subseteq \Gamma'$ and $\Gamma' \vdash \Phi'$ and $\Gamma' \vdash \textbf{rename}(\eta,\sigma,\Sigma,x,y,(\texttt{Thread},1))$ via our well-typed renaming lemma. This lets us use the **T-Async-Partition** rule to check this case, with a choice of Γ' as $\Gamma',x:^{(\texttt{Thread},1)}\tau[]^l$.
  In the **S-Async-Partition-Unwind case**, our assumption that Φ is well-typed tells us that $\Gamma,y:^{(\texttt{Thread},1)}\vdash^{\{}\ (\texttt{Thread},1)_m s$. Thus, we can use the **T-Async-Partition** rule to type this case.
- Case **T-Async-Memcpy**
  In this case the statement and environment typing are trivial, we need only to show that the async stack remains well typed.
  In this case we have that x and y have the same type at $(\texttt{Thread},1)$. This is sufficient for us to check $x=y$ at $(\texttt{Thread},1)$, meaning that adding that instruction to the stack maintains its well-typedness.
- Case **T-Memcpy**
  Immediate via use of the **S-Memcpy** rule. We just need to show that the environment remains well typed, which we know via our lemmas about **update** and **get**.

Memory Rules.

- Case **T-Decl**
  By inversion, we are using **S-Decl** rule for evaluation. Our well-typed expression lemma gives us that v is well-typed, so it follows from our assumptions and our lemmas about extending environments that the extended η and s are well-typed by $\Gamma,x:^{\pi}\tau$.
- Case **T-Free**
  Trivial.
- Case **T-Alloc**
  By inversion we are either in the **S-Alloc-Local** or **S-Alloc-Global** rules. In either case, we assume that $\Gamma,x:^{\pi}\tau[]^l$ checks s, meaning we can use our extended environment lemmas and the **T-Seq** and **T-Free** rules to check these cases.
- Case **T-Alloc-Shared**
  Same as previous case.
- Case **T-Partition**
  In this case we have by assumption that $\Gamma,y:^{(h,n/c)}\tau[]^l s$ and l is not local and c divides n. By inversion on our step relation we must be in the **S-Partition** case, so we have $\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash^{\pi}_{m}$ $\mathit{partition}\psi xycs \rightsquigarrow \texttt{init}_{\psi};s';\texttt{dec}_{\psi};\texttt{wait}_{\psi} \dashv_m \eta',\sigma',\Sigma',\Psi,\Phi$ where $s' = s[(y+c\cdot p)/y]$ and $(\eta',\sigma',\Sigma') = \textbf{rename}(\eta,\sigma,\Sigma,x,y,\pi/c)$. Via our well-typed renaming lemma contexts we know

that we can check the renamed environments in the extended environment $\Gamma, y :^{\pi/c} \tau[]^{l}, x :^{\pi} \tau[]^{l}$, and this context is also sufficient to check s' (via a substitution-preserves-typing lemma that is obvious). Using the **T-Skip** rule this is exactly what we need to show to complete this case, as the thread sync primitives check trivially via their typing rules.

- Case **T-Claim**
  Essentially the same as **T-Partition**.
- Case **T-Lower**
  Essentially the same as **T-Partition**.

Control Rules.

- Case **T-If**
  We have by assumption that `if` e `then` s_1 `else` s_2 steps to some s', and by inversion we know that either e evaluates to `true` and s' is s_1, or e evaluates to `false` and s' is s_2.
  In either case, our inductive hypotheses is sufficient to tell us that these are well-typed. In particular, in both cases our IHs tell us that the amount of memory used by stepping each branch of the `if` is less than the amount of memory computed by the type system for each branch. Because the whole `if` expression checks using the greater of the memory usage of m_1 or m_2 (i.e., the memory usage on each branch), the resulting usage for the whole conditional is also bounded by the type system.
- Case **T-Skip**
  Trivial: `skip` does not step
- Case **T-Seq**
  In this case we have that s_1 and s_2 are both well-typed (with m_1 memory and m_2 memory respectively), and $m = \mathbf{max}(m_1, m_2)$. We also have by inversion that s_1 either steps to `skip` or s_1', and our inductive hypothesis tells us that s_1' is well-typed.
  In the former case we can use the **S-Seq-Done** rule to trivially finish the case. In the latter, our IH allows us to finish the case, since m_1 is always $\leq \mathbf{max}(m_1, m_2)$
- Case **T-While**
  By inversion, we have that

$$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{\pi} \texttt{while } e \texttt{ do } s$$
$$\rightsquigarrow \texttt{if } e \texttt{ then } (s; \texttt{ while } e \texttt{ do } s) \texttt{ else skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi.$$

  We also have by assumption that e and s are well-typed. With this information, through use of the **T-If**, **T-Seq**, **T-While**, and **T-Skip** rules, we can conclude that the result of this rule is also well-typed.
- Case **T-Function-Call**
  By inversion we have that

$$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{\pi} f(e_1, \ldots, e_n)$$
$$\rightsquigarrow \texttt{call } s \texttt{ with } (x_i :^{\pi_i} \tau_i \mapsto v_i)@\{\eta, \sigma, \Sigma, m'\} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi,$$

  and also that $\Sigma(f) = \{[x_1 : \tau_1, \ldots, x_n : \tau_n], s\}$, and $\eta, \sigma, \Sigma, t, b, p \vdash^{\pi} e_i \Downarrow v_i$, and $m' \leq m$.
  We also have from the premises of our case that $f :^{\pi} \mathbf{Fun}(x_1 :^{\pi_1} \tau_1, \ldots, x_n :^{\pi_n} \tau_n, \pi, m') \in \Gamma$, and $\Gamma \vdash^{\pi} e_i : \tau_i$, and $m' \leq m$.
  Our lemma for expression well-typedness tells us that that each v_i is a well-typed value, and our assumption that $\Gamma \vdash \eta, \sigma, \Sigma$ tells us that $\Sigma(f)$ is a well-typed function and thus that $\mathbf{fns}\ \Gamma, x_i :^{\pi_i} \tau_i \vdash_{m'}^{\pi} s$. We can take the union of this with Γ to produce $\Gamma, x_i :^{\pi_i} \tau_i \vdash_{m'}^{\pi} s$, which clearly checks s and the output environments.

□

```
@ccuda("global")
@requires(grid[1], block[1], thread[1], smem=49000)
def my_sgemm_kernel_1( M : int @ grid[1],
                       N : int @ grid[1],
                       K : int @ grid[1],
                       alpha : float @ grid[1],
                       A: ptr(const(float)) @ grid[1],
                       B: ptr(const(float)) @ grid[1],
                       beta : float @ grid[1],
                       C: ptr(float) @ grid[1],
                       block_size : int @ grid[1]):
    with group(grid[1]):
        bid : int @ block[1] = id()
        with partition(C, dimension=block[1], offset=0) as C_blk:
            with group(block[1]):
                tid : int @ thread[1] = bid * block_size + id()
                with partition(C_blk, dimension=thread[1], offset=0) as C_thrd:
                    x : const(int) @ thread[1] = tid / N
                    y : const(int) @ thread[1] = tid % N
                    with group(thread[1]):
                        tmp : float @ thread[1] = 0
                        for i in range(0, K, 1):
                            tmp += A[x * K + i] * B[i * N + y]
                        C_thrd[x * N + y] = alpha * tmp + beta * C[x * N + y]
    return
```

```
@ccuda("global")
@requires(grid[1], block[1], thread[1], smem=49000)
def my_sgemm_kernel_2( M : int @ grid[1],
                        N : int @ grid[1],
                        K : int @ grid[1],
                        alpha : float @ grid[1],
                        A: ptr(const(float)) @ grid[1],
                        B: ptr(const(float)) @ grid[1],
                        beta : float @ grid[1],
                        C: ptr(float) @ grid[1],
                        tile_size : int @ grid[1]):
    with group(grid[1]):
        global_tid: int @ thread[1] = id()
        total_elements : const(int) @ grid[1] = M * N
        num_tiles_per_row :const(int) @ grid[1] = (N + tile_size - 1) / tile_size
        tile_id : const(int) @ thread[1] = global_tid / (tile_size * tile_size)
        tile_row : const(int) @ thread[1] = tile_id / num_tiles_per_row
        tile_col : const(int) @ thread[1] = tile_id % num_tiles_per_row
        local_id : const(int) @ thread[1] = global_tid % (tile_size * tile_size)
        local_row : const(int) @ thread[1] = local_id / tile_size
        local_col : const(int) @ thread[1] = local_id % tile_size
        cRow : const(int) @ thread[1] = tile_row * tile_size + local_row
        cCol : const(int) @ thread[1] = tile_col * tile_size + local_col
        with partition(C, dimension=thread[1], offset=cRow * N + cCol) as C_thrd:
            with group(thread[1]):
                if global_tid < total_elements:
                    if cRow < M and cCol < N:
                        tmp : float @ thread[1] = 0
                        for i in range(0, K, 1):
                            tmp += A[cRow * K + i] * B[i * N + cCol]
                        C_thrd[0] = alpha * tmp + beta * C_thrd[0]


    return
```

B Programs from Evaluation

We reproduce programs that we wrote for Section 6 in this section.

```
@ccuda("global")
@attr("__launch_bounds__((BM * BN) / (TM * TN), 1)")
@template([("BM", c_int),
          ("BN", c_int),
          ("BK", c_int),
          ("TM", c_int),
          ("TN", c_int)])
@requires(grid[1], block[1], thread[1], smem=49000)
def my_sgemm_kernel_3( M : int @ grid[1],
                      N : int @ grid[1],
                      K : int @ grid[1],
                      alpha : float @ grid[1],
                      A: ptr(const(float)) @ grid[1],
                      B: ptr(const(float)) @ grid[1],
                      beta : float @ grid[1],
                      C: ptr(float) @ grid[1],
                      num_blocks_N : int @ grid[1]):
    with group(grid[1]):
        cRow : const(int) @ block[1] = id() / num_blocks_N
        cCol : const(int) @ block[1] = id() % num_blocks_N
        totalResultsBlocktile: const(uint) @ grid[1] = BM * BN
        numThreadsBlocktile: const(uint) @ grid[1] = totalResultsBlocktile / (TM * TN)
        with partition(C, dimension=block[1], offset = cRow * BM * N + cCol * BN) as C_blk:
            with partition(A, dimension=block[1], offset = cRow * BM * K) as A_blk:
                with partition(B, dimension=block[1], offset = cCol * BN) as B_blk:
                    with group(block[1]):
                        As : shared(float[128 * 8]) @ block[1]
                        Bs : shared(float[128 * 8]) @ block[1]
                        threadRow : const(int) @ thread[1] = id() / (BN / TN)
                        threadCol : const(int) @ thread[1] = id() % (BN / TN)

                        innerRowA : const(int) @ thread[1] = id() / BK
                        innerColA : const(int) @ thread[1] = id() % BK
                        strideA : const(int) @ thread[1] = numThreadsBlocktile / BK

                        innerRowB : const(int) @ thread[1] = id() / BN
                        innerColB : const(int) @ thread[1] = id() % BN
                        strideB : const(int) @ thread[1] = numThreadsBlocktile / BN

                        threadResults : float[64] @ thread[1] = { 0 }
                        regM : float[8] @ thread[1] = { 0 } # Need to go add the {}
                        regN : float[8] @ thread[1] = { 0 }

                        i : int @ thread[1] = 0

                        for bkIdx in range(0, K, BK):

                            with partition(As, dimension=thread[1], offset = 0) as As_thrd:
                                with partition(Bs, dimension=thread[1], offset = 0) as Bs_thrd:
                                    with group(thread[1]):
                                        for loadOffset in range(0, BM, strideA):
                                            As_thrd[(innerRowA + loadOffset) * BK + innerColA] = \
                                                A_blk[ i * BK + (innerRowA + loadOffset) * K + innerColA]

                                        for loadOffset in range(0, BK, strideB):
                                            Bs_thrd[(innerRowB + loadOffset) * BN + innerColB] = \
                                                B_blk[(i * BK * N) + (innerRowB + loadOffset) * N + innerColB]

                                        i += 1

                                        for dotIdx in range(0, BK, 1):
                                            for i in range(0, TN, 1):
                                                regM[i] = As[(threadRow * TM + i) * BK + dotIdx]
                                            for i in range(0, TN, 1):
                                                regN[i] = Bs[dotIdx * BN + threadCol * TN + i]
                                            for resIdxM in range(0, TM, 1):
                                                for resIdxN in range(0, TN, 1):
                                                        threadResults[resIdxM * TN + resIdxN] += regM[resIdxM] * regN[resIdxN]

                        for resIdxM in range(0, TM, 1):
                            for resIdxN in range(0, TN, 1):
                                with partition(C_blk, dimension=thread[1], offset = (threadRow * TM + resIdxM) * N + threadCol * TN +
                                ↪ resIdxN) as C_thrd:
                                    with group(thread[1]):
                                        C_thrd[0] = alpha * threadResults[resIdxM * TN + resIdxN] + beta * C_blk[(threadRow * TM +
                                        ↪ resIdxM) * N + threadCol * TN + resIdxN]

    return
```

```

```

```
@ccuda("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]):
    with group(block[1]):
        tid : int @ thread[32]  = id()
        with partition(input, dimension=thread[32], offset = tid * items_per_thread) as input_thrd:
            with group(thread[32]):
                warp_load(input_thrd, output, items_per_thread)

    return
```

```
@ccuda("device")
@requires(thread[32])
def warp_load(input : ptr(const(int)) @ thread[32],
               output : ptr(int) @ thread[1],
               items_per_thread : int @ thread[32]):
    with group(thread[32]):
        tid : int @ thread[1]  = id()
        with partition(input, dimension=thread[1], offset = tid * items_per_thread) as input_thrd:
            with group(thread[1]):
                thread_load(input_thrd, output, items_per_thread)

    return
```

```
@ccuda("device")
@requires(thread[1])
def thread_load(input : ptr(const(int)) @ thread[1],
               output : ptr(int) @ thread[1],
               items_per_thread : int @ thread[1]):
    with group(thread[1]):
        for i in range(0, items_per_thread, 1):
            output[i] = input[i]
    return
```

```
@ccuda("device")
@requires(block[1], thread[32])
def block_store(input : ptr(const(int)) @ thread[1],
               output : ptr(int) @ block[1],
               items_per_thread : int @ block[1]):
    with group(block[1]):
        tid : int @ thread[32]  = id()
        with partition(output, dimension=thread[32], offset = tid * items_per_thread) as output_thrd:
            with group(thread[32]):
                warp_store(input, output_thrd, items_per_thread)
    return
```

```
@ccuda("device")
@requires(thread[32])
def warp_store(input : ptr(const(int)) @ thread[1],
               output : ptr(int) @ thread[32],
               items_per_thread : int @ thread[32]):
    with group(thread[32]):
        tid : int @ thread[1]  = id()
        with partition(output, dimension=thread[1], offset = tid * items_per_thread) as output_thrd:
            with group(thread[1]):
                thread_store(input, output_thrd, items_per_thread)
    return
```

```
@ccuda("device")
@requires(thread[1])
def thread_store(input : ptr(const(int)) @ thread[1],
            output : ptr(int) @ thread[1],
            items_per_thread : int @ thread[1]):
    with group(thread[1]):
        for i in range(0, items_per_thread, 1):
            output[i] = input[i]
    return
```

```
@ccuda("global")
@attr(" __launch_bounds__(128*3) ")
@requires(grid[1], block[1], thread[1], warp[12], smem=227000)
def cutlass_my_h100_match_no_tail(
         M: int @ grid[1],
         N: int @ grid[1],
         K: int @ grid[1],
         C: ptr(bf16) @ grid[1],
         tensorMapA: const(CUtensorMap) @ grid[1],
         tensorMapB: const(CUtensorMap) @ grid[1]):

   with group(grid[1]):
      BM : constexpr(int) @ grid[1] = 128
      BN : constexpr(int) @ grid[1] = 256
      BK : constexpr(int) @ grid[1] = 64
      NUM_THREADS : constexpr(int) @ grid[1] = 128*3
      QSIZE : constexpr(int) @ grid[1] = 4
      NUM_SM : constexpr(int) @ grid[1] = 128

      WGMMA_M : constexpr(int) @ grid[1] = 64
      WGMMA_K : constexpr(int) @ grid[1] = 16
      WGMMA_N : constexpr(int) @ grid[1] = BN
      num_consumers : constexpr(int) @ grid[1] = (NUM_THREADS / 128) - 1
      B_WG_M : constexpr(int) @ grid[1] = BM / num_consumers

      TM : constexpr(int) @ grid[1] = 16
      TN : constexpr(int) @ grid[1] = 8
      schedule_block : int @ block[1] = id()

      with partition(C, offset=0, dimension=block[1]) as block_C:
         with group(block[1]):

            # Allocate SA
            #  sA[QSIZE][BK*BM]
            sA_slot0 : shared(bf16[64* 128], align=128) @ block[1]
            sA_slot1 : shared(bf16[64* 128]) @ block[1]
            sA_slot2 : shared(bf16[64* 128]) @ block[1]
            sA_slot3 : shared(bf16[64* 128]) @ block[1]

            # Allocate SB
            #  sB[QSIZE][BK*BN]
            sB_slot0 : shared(bf16[64* 256], align=128) @ block[1]
            sB_slot1 : shared(bf16[64* 256]) @ block[1]
            sB_slot2 : shared(bf16[64* 256]) @ block[1]
            sB_slot3 : shared(bf16[64* 256]) @ block[1]


            num_blocks_k : const(int) @ block[1] = K / BK
            wg_idx : int @ warp[4] = id()
            blk_thrd_id : int @ thread[1] = id()


            tid : int @ thread[1] = id() % 128
            is_producer : bool @ warp[4] = wg_idx == 0

            num_block_m : int @ block[1] = 0
            num_block_n : int @ block[1] = 0

            with group(warp[4]):
               if (is_producer):
                  pass
               else:
                  wg_idx = wg_idx - 1

            schedule_it : int @ block[1] = 0
            total_blocks_m : int @ block[1] = (((M + BM) - 1) / BM)
            total_blocks_n : int @ block[1] = (((N + BN) - 1) / BN)
            unsafe("assert(CEIL_DIV(M, BM)%TM == 0 && total_blocks_n%TN == 0);")

            while(true):
               num : int @ block[1] = ((schedule_it * NUM_SM) + schedule_block)
               if (num >= (total_blocks_m * total_blocks_n)):
                  break
               cur_tile : int @ block[1] = (num / (TM * TN))
               cur_tile_pos : int @ block[1] = (num % (TM * TN))
               num_block_m = (TM * (cur_tile / (total_blocks_n / TN)))
               num_block_n = (TN * (cur_tile % (total_blocks_n / TN)))
               num_block_m += (cur_tile_pos / TN)
               num_block_n += (cur_tile_pos % TN)
               schedule_it += 1
               d : c_float[1][16][8] @ thread[1]

               with group(thread[1]):
                  d_ptr : ptr(float) @ thread[1]
                  # TODO: ANNOYONG TO NOT HAVE TO SET EACH INDIVIDUAL TO 0.
                  unsafe("d_ptr = (float *)d;")
                  memset_wrapper(d_ptr, 0, 1 * 16 * 8 * 4)

               # Now remove these registers...
               with group(warp[4]):
                  if (is_producer):
                     num_regs : constexpr(int) @ warp[4] =  24 if (num_consumers <= 2) else 32
                     warpgroup_reg_dealloc(template=[num_regs])
```