Modular GPU Programming with Typed Privileges

ANONYMOUS AUTHOR(S)

Abstract

1 Introduction

CUDA [10] is a low-level, imperative programming language for NVIDIA GPUs. These GPUs are organized into a hierarchy of compute resources: threads, blocks, and a grid. *Threads* are the basic units 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. CUDA programs are written from the *perspective of a single thread*, and replicated in parallel across the entire grid.

Although programs are written per-thread, some operations supported by the hardware are only semantically valid when executed *collectively* by a group of threads. For example, the `__syncthreads()` primitive, which synchronizes all threads within a block, causes the machine to deadlock if executed by a subset of threads within that block. Similarly, tensor core instructions—hardware-accelerated matrix-multiply operations—are only well-defined when invoked by a 32-thread group (known as a *warp*) whose starting index is 32-aligned. In essence, these operations require the programmer to carefully marshal compute resources; coordinating which threads execute which blocks of code. As a result, collective operations *break the illusion* of CUDA programs executing as independent threads.

The conceptual clash between the per-thread programming paradigm and these collective operations hinders compositional reasoning. In CUDA, function calls — like all other statements — are independently executed on all threads. However, functions may contain collective operations that are only valid when executed together by a group of threads. This creates a semantic gap: the call site appears to be a per-thread operation, but the function body may require collective execution by a specific configuration of the device's compute resources, such as an entire block or warp.

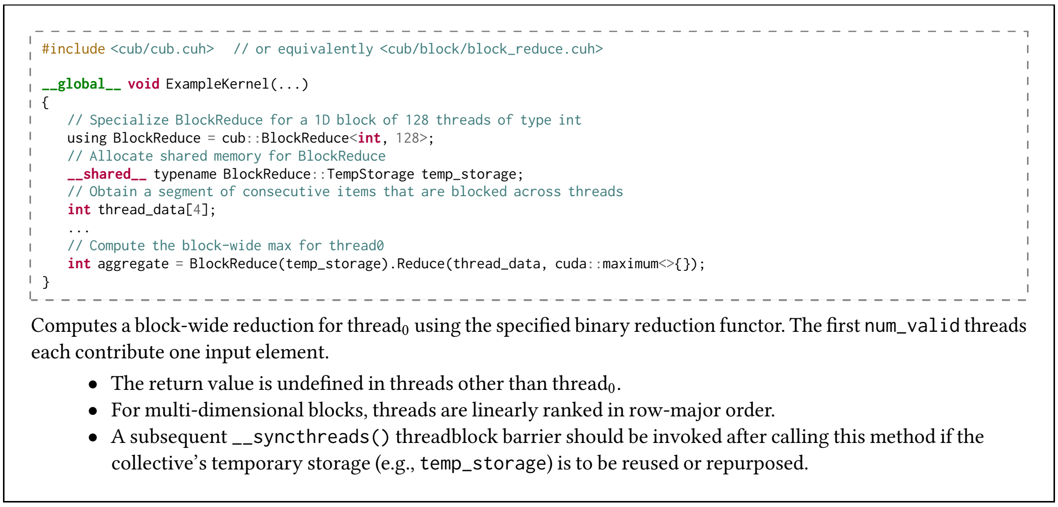
This gap between the syntax and semantics of a CUDA program additionally creates tension between abstraction and correctness. This tension is apparent even in widely used CUDA libraries that package common functionality via function interfaces . Consider the snippet of documentation presented in Figure 1 taken directly from CUB [9], a library from NVIDIA that provides wrappers around parallel primitives like scan, sort, and histogram, among others. The documentation describes a function called `BlockReduce` , in which all threads in a block collaboratively apply a reduction operation, such as a maximum or prefix sum, over an array.

The CUB documentation attempts to specify several preconditions that must be satisfied by callers of `BlockReduce`. First, the function is only well-defined when invoked by an entire block's worth of threads. Second, the function accesses shared memory, a scratchpad space visible to all threads in a block. Users wishing to reuse this storage must first ensure all the block's threads have finished using it by inserting a `__syncthreads()` barrier before the reuse. Whether this synchronization is required cannot be determined from the function alone; it depends on how the storage is used afterward.

In effect, CUB attempts to informally retrofit CUDA with information about the compute and memory requirements of collective operations.

By prefixing functions with identifiers such as `Block`, CUB effectively creates "namespaces" for different kinds of these operations. Even then, this organization is purely conventional: it encodes hierarchy through naming and cannot be enforced by the compiler. Ultimately, correctness depends not only on the user carefully reading and interpreting the documentation, but also on the library implementer upholding the invariants of each namespace.

Previous attempts to resolve the mismatch between the syntax and semantics of CUDA programs have generally done so by raising the level of abstraction. Triton [14], a tile-based, multi-dimensional



```
#include <cub/cub.cuh>   // or equivalently <cub/block/block_reduce.cuh>

__global__ void ExampleKernel(...)
{
    // Specialize BlockReduce for a 1D block of 128 threads of type int
    using BlockReduce = cub::BlockReduce<int, 128>;
    // Allocate shared memory for BlockReduce
    __shared__ typename BlockReduce::TempStorage temp_storage;
    // Obtain a segment of consecutive items that are blocked across threads
    int thread_data[4];
    ...
    // Compute the block-wide max for thread0
    int aggregate = BlockReduce(temp_storage).Reduce(thread_data, cuda::maximum<>{});
}
```

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$_0$.
- For multi-dimensional blocks, threads are linearly ranked in row-major order.
- 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.

Fig. 1. CUB documentation for the `BlockReduce` function

array language, circumvents the challenges of compositional reasoning by restricting users to a single level of the hierarchy, namely the block level. While this makes for a simplified programming model, the lack of fine-grained control prevents GPU programmers from writing the highest-performance kernels.

Meanwhile, Pallas is a recent entrant similar to Triton that restricts users to programming at the level of "warpgroups" (four warps).

A variety of functional languages, such as Futhark [7] , provide compile-time guarantees through their type systems but do not expose low-level control over the hardware. Some other efforts, like Descend [8], aim to provide a low-level, safe GPU systems programming language, but lack support for collective operations entirely. As a result, CUDA remains the de-facto standard for writing high-performance kernels on modern GPUs .

Rather than sacrificing low-level control for safety, we aim to make fine-grained GPU programming both safe and compositional by design. Our core insight is that GPU programmers must carefully map programs to available compute resources, and unlike CUDA, which obscures this mapping, we can expose it explicitly in syntax. By making this mapping explicit, we can provide low-level access to collective operations while ensuring compile-time safety: the compiler can check that requirements about what hardware resources are required to execute code are satisfied by the context in which that code is executed.

We introduce Snap, a new low-level GPU programming language that guarantees correct usage of hardware resources by construction. Inspired by coeffect systems [11] and dependency calculi [1], Snap statically tracks the configurations of compute and memory resources at the type level.

We reify the notion of compute resources via *privileges*, so named because they dictate which operations are permitted in which contexts. Snap uses these privileges to verify that every collective operation is executed with the necessary resources. To enable modular reasoning, function interfaces include *privilege signatures* that declare the privilege each function requires for correct execution.

To validate the design of Snap, we build Bundl, a core calculus model of Snap that tracks privilege in both its type system and operational semantics. We provide formal rules for manipulating the privilege of both code and data, and prove *type-and-privilege safety*, ensuring that Bundl is sound, and that all threads are statically privileged to run the operations they attempt at run time.

A parallel goal, in addition to correctness, is performance. To that end, we incorporate modern GPU features such as tensor cores and asynchronous data movement into Snap, and demonstrate that Snap can achieve the same performance as hand-written, highly optimized code on an H100 and a 4070 Ti Super.

Concretely, our contributions are:

(1) We provide background on GPU compute architecture, CUDA's per-thread programming model, and the convergence requirements of collective operations. We motivate the need for compile-time checking of resource usage by example (Section 2).
(2) We present Snap, a low-level GPU programming language with type-level privileges that track compute and memory resources (Section 3).
(3) We formalize the core concepts of Snap in Bundl, a calculus that tracks privileges in both its type system and operational semantics. We prove type-and-privilege safety, ensuring that threads only execute operations for which they are statically privileged (Section 4).
(4) We compile Snap programs into CUDA (Section 5), implement various GPU kernels in Snap, and demonstrate that they match the performance of hand-optimized CUDA implementations (Section 6).

Section 7 addresses the limitations of our approach, Section 8 discusses related work, and Section 9 concludes and outlines future work.

2 Background & Motivation

Before diving into the design of Snap, we begin with an overview of the GPU's compute and memory hierarchies and outline the challenges posed by reasoning about them collectively.

2.1 Compute Hierarchy

In CUDA, programmers launch computations that run on thousands of threads. On the hardware, work is scheduled at the granularity of blocks on a *streaming multiprocessor*, or an *SM*. Within an SM, execution follows the the Single Instruction, Multiple Threads (SIMT) model. Instructions are issued to groups of 32 threads, or a *warp*, at a time, and each thread in the warp executes the instruction in lockstep.

Users define the behavior of different levels of the compute hierarchy by defining the behavior of a single thread . CUDA programs can read 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. Combining these two IDs, threads can locate themselves in their share of the full computation.

A natural and tempting way to interpret these built-in identifiers is to think of them as indices of implicit "parallel `for`-loops" 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, a warp-level tensor-core operation, shown in Figure 2, is only *meaningful* when invoked by a collection of 32

```
int tid = threadIdx.x;
if (tid >= 0 && tid< 31){
  float A[4];
  float B[2];
  float C[4] = { 0 };
  # Populate A with unique values
  for (int i = 0; i < 4; i++)
     A[i] = tid * 4 + 1;
  # Populate B with unique values
  for (int i = 0; i < 2; i++)
     B[i] = tid * 4 + 1;
  # Issue a warp-level tensor-core
  # operation. D = A * B  + C
  asm("mma.sync.aligned.m16n8k8..."
    "{%0, %1, %2, %3},    " /*D*/
    "{%4, %5, %6, %7},    " /*A*/
    "{%8, %9},          " /*B*/
    "{%10, %11, %12, %13};" /*C*/
    : "=r"(C[0]), "=r"(C[1]), ...
    : "r"(A[0]), "r"(A[1]), ...
    "r"(B[0]), "r"(B[1]),
    "r"(C[0]), "r"(C[1]), ...);}
```

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

threads—a “warp”—acting together. If the operation was invoked by fewer than 32 threads, the instruction would be invalid . 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. For example, if the condition on line 2 of Figure 2 were instead `tid > 1 && tid < 33`, the instruction would still be invalid, even though 32 threads would execute it. Once the instruction has been correctly invoked, the tensor-core executes once collectively for all threads, and its result is distributed across the registers of the participating threads.

Collective operations make reasoning about CUDA programs challenging because they force programmers to track the convergence behavior of threads. Specifically, programmers must reason 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 it changes over the course of a program’s execution. Second, threads may be participating in *different kinds 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 `__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 *across* functions . A callee may assume a certain number of threads and have internally structured its computation based on that assumption, including reasoning about convergence behavior. However, that assumption is not visible in the callee’s function interface, which only has information about 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.

```
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      # Populate A with unique values
7      for i in range(0, 4, 1):
8          A[i] = tid * 4 + 1
9          C[i] = 0
10     # Populate B with unique values
11     for i in range(0, 2, 1):
12         B[i] = tid * 4 + 1
13     # Issue a warp-level tensor-core
14     intrinsic.mma(
15       A[0], A[1], A[2], A[3],
16       B[0], B[1],
17       C[0], C[1], C[2], C[3],
18       out=[A[0], A[1], A[2], A[3]]
19     )
```

Fig. 3. Invoking a warp-level tensor-core instruction in Snap.

In this work, we make the programmer’s implicit assumptions about a program’s convergence behavior explicit in the program’s source. Consider the example in Figure 3 which shows an equivalent rewrite of the CUDA program in Figure 2 using Snap. In Figure 3, the call to the `mma` intrinsic is legal only because it is executed within a `group` of 32 threads, as made explicit on line 2. Since Snap’s `group` construct already enforces both the size and alignment of participating threads, the legality of the `mma` operation is guaranteed by construction.

Now, if we had attempted to introduce a condition based on `tid` *within* the `group`’s scope—for example, `if (tid < 16)` — which would cause the number of active threads in the group to change during its execution , Snap would reject the program at compile time . This is because Snap explicitly tracks the frequency at which values diverge, using the `@` syntax, and code executing at a lower divergence frequency than a given piece of data does not have the *privilege* to read or branch on that data. We explain the syntax and rules of legal Snap programs in greater detail in Section 3.

2.2 Memory Hierarchy

GPUs also have a *memory hierarchy*, which, like the compute hierarchy, is organized into three levels. 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 data that will be repeatedly used. Finally, each thread maintains its private state in *registers* and *local memory*, which is used for storing the thread's stack.

This memory hierarchy has its own notion of convergence and divergence, again mirroring that of compute. For example, when launching a kernel on the GPU, a 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 *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 in the hierarchy.

CUDA does not explicitly model these memory spaces, nor does it track whether allocations in a given memory space, such as shared memory, exceed device limits. Snap, on the other hand, reifies these spaces in its type system by explicitly decorating each variable with its associated memory space and restricting shared memory to static allocations, throwing an error if an allocation exceeds device limits. We will discuss these aspects in detail in Section 3.5. For now, however, we focus on how memory diverges and reconverges in tandem with the compute hierarchy.

```
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.

Let us reconsider the tensor-core example in Figure 2, this time initializing the operands of the tensor-core operation from pointers into memory. In Figure 4, the registers A, B, and C are populated by first loading from a_mem and b_mem, which point into global memory. After the tensor-core operation completes, the result is written back to c_mem.

In this program, c_mem is accessed from two different levels of the compute heirarchy.

At the start, c_mem is accessible collectively by the group of 32 threads, which determines the base offset into the global array with which the group interacts . This is the ownership assumed when entering this code . Then, c_mem is further partitioned across individual threads so that each thread can write back the portion of the result it computed . To restore c_mem back to its 32 thread-level ownership, all 32 threads must first synchronize to ensure all per-thread writes have completed. In Section 1 , we saw a similar requirement being documented in the CUB library.

Similarly to the compute hierarchy (Section 2.1), CUDA programmers are responsible for tracking the logical ownership 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.

Snap, by contrast, makes the evolution of a memory object's logical ownership explicit in the type system and *automatically* inserts the synchronization required

```
@ccuda("device")
@requires(thread[32])
def simple_mma(
    a: ptr(const(float)) @ thread[32],
    b: ptr(const(float)) @ thread[32],
    c: ptr(float) @ thread[32]):
  x : int @ thread[1] = id()
  with group(thread[32]):
   A : float[4] @ thread[1]
   B : float[2] @ thread[1]
   C : float[4] @ thread[1]

   # Reads do not need to be lowered.
   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]);

   B[0] = b_mem[(x%4)*n+(x/4)]);
   B[1] = b_mem[(x%4+4)*n+(x/4)]);
   # Skipping initialize C to 0...
   intrinsic.mma(
     A[0], A[1], A[2], A[3],
     B[0], B[1],
     C[0], C[1], C[2], C[3],
     out=[C[0], C[1], C[2], C[3]]
     )
   # Must be at thread[1] to write.
   idx : int @ thread[1] =
       lambda ro, co: (x/4+ro)*n+2*(x%4)+co
   with partition(c_mem, p=thread[32],
  ↪    f=idx) as c_thrd:
     c_thrd[0, 0] = C[0]
     c_thrd[0, 1] = C[1]
     c_thrd[8, 0] = C[2]
     c_thrd[8, 1)] = C[3]
   # <-- Sync point inferred by name
```

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

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 Snap only permits writes when data is owned by a single thread. To lower `c_mem`, we use Snap's `partition` operation on line 31. This takes the name of the memory to partition, `c_mem`, and lowers it to a compute hierarchy below the original one, a single thread, and assigns it a new name, `c_thrd`; Snap will not allow references to the old name `c_mem` within the partition's associated 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, Snap will insert a synchronization before the first use of the original name.

Finally, let's examine the function interface on line 3 . The function marks `c_mem` as a non-const pointer, signaling to Snap that it may be used for writing. In the calling context of this function, we also synchronize `c_mem`'s logical owner, if one exists, before restoring `c_mem`'s parent in its scope.

We would like to emphasize that preventing data races is not an one of Snap's goals. In Snap, 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 [8], lays the foundation for reasoning about data-race free programs, and that a Descend-like memory model can be adapted to Snap. Snap 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 Descend cannot provide.

3 The Snap Language

Snap is an imperative, low-level programming language designed at a level of abstraction comparable to CUDA. Snap's surface syntax explicitly represents how compute (grid, block, threads) and data (global, shared and local memory) resources are used and distributed over the course of its execution. Using this explicit representation, Snap can enforce safety properties that CUDA cannot. Namely, Snap rejects programs that attempt to execute operations with insufficient compute and memory resources.

In this section, we introduce Snap's core constructs through a running example, a tensor-core matrix multiplication, shown in Figure 6. This is a standard computation a GPU programmer might write while being sufficiently complex to reveal the interesting challenges in GPU programming.

The program in Figure 6 computes a matrix multiplication between two float arrays, `A` and `B` to produce an output matrix `C`. In this kernel, each block computes an independent tile of the output `C`. To perform the computation, each block first locates its assigned tile index, then loads the corresponding rows and columns from `A` and `B` (line). Since matrix multiplication offers ample data reuse, to amortize the cost of loading data, we stage `A` and `B` in shared memory (lines). At this point, each thread within the block *diverges* and loads different elements of `A` and `B`. Finally, the program invokes an Ampere warp-level tensor-core instruction to compute the output, requiring threads in a warp to *converge* (line). We encapsulate this tensor-core instruction in a function, demonstrating how function composition works in Snap. This function's implementation is identical to the one shown in Figure 5.

3.1 Levels

Snap models the machine's compute hierarchy through *levels*. There are three levels in Snap—`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 a strict

```
@ccuda("global")
# Top-level privilege class and required shared memory usage.
@requires(grid[1], block[1], thread[32], smem=1280)
def mmaTF32NaiveKernel(A: ptr(const(float)) @ grid[1],
            B: ptr(const(float)) @ grid[1],
            C: ptr(float) @ grid[1],
            M : int @ grid[1],
            N : int @ grid[1],
            K : int @ grid[1]):
  # Replicate this code across 1 grid.
  with group(grid[1]):
    # @ grid[1] inferred from compute privelege
    MMA_N  : constexpr(int) = 8
    ...
    num_blocks_n : const(int) = (N + MMA_N - 1) / MMA_N
    # id() function returns the block id, inferred from
    # @ block[1].
    block_row : const(int) @ block[1] = id() / num_blocks_n
    block_col : const(int) @ block[1] = id() % num_blocks_n

    warp_row: const(int) @ block[1] = block_row * MMA_M
    warp_col : const(int) @ block[1] = block_col * MMA_N
    # Give each block an offset into C
    offset = lambda x: warp_row * N + warp_col + x
    with partition(C, p=block[1], f=offset) as C_blk:
      with group(block[1]):
        # SMEM declarations only allowed with block[1] privilege.
        A_smem : shared(float[16 * 8]) @ block[1]
        B_smem : shared(float[8 * 8])  @ block[1]
        C_smem : shared(float[16 * 8]) @ block[1]
        # Now, id() returns the thread id
        idx : int @ thread[1] = id() * 4
        # To write to C_smem, must drop to thread[1] privilege
        with partition(C_smem, p=thread[1], offset=idx) as C_thrd:
          for i in range(0, 4, 1):
            with group(thread[1]):
              C_thrd[i] = 0

        for i in range(0, K_tiles, 1):
          a_idx : int @ thread[1] = id() * 4
          # <--- Name will insert a sync point
          # because this partition is a parent partition
          # of itself (back edge from for loop)
          for j in range(0, 4, 1):
            global_row : int @ thread[1] = warp_row + row
            global_col: int @ thread[1] = i * MMA_K + col
            with partition(A_smem, p=thread[1], f=..) as As_thrd:
              with group(thread[1]):
                As_thrd[0] = A[global_row * K + global_col]

          b_idx : int @ thread[1] = id() * 2
          # <--- Name will insert a sync point
          # (back edge from for loop)
          for j in range(0, 2, 1):
            # Similar to write into C_smem ...
            with partition(B_smem, p=thread[1], f=...) as Bs_thrd:
              with group(thread[1]):
                Bs_thrd[0] = B[global_row_b * N + global_col_b]

            # <--- Name will insert a sync point
            # (back edge from for loop)
            # Give each warp an offset into C_smem
            with claim(C_smem, p=thread[32]) as Cs_warp:
              match split(thread):
                case 32:
                  # Can pass A_smem and B_smem
                  # as arguments to simple_mma for
                  # those two are declared const.
                  # "read up" is okay.
                  simple_mma(A_smem, B_smem, Cs_warp)

        # Write back final result to C
        for j in range(0, 4, 1):
          flat_idx_c : int @ thread[1] = id() * 4 + j
          row_c : int @ thread[1] = flat_idx_c / MMA_K
          col_c : int @ thread[1] = flat_idx_c % MMA_K
          offset = lambda x: row_c * N + col_c + x
          with partition(C_blk, p=thread[1], f=offset) as C_thrd:
            with group(thread[1]):
              C_thrd[0]  = C_smem[row_c * MMA_N + col_c]
  return
```

Fig. 6. Tensor Float 32 matrix Multiplication in Snap.

order defined over them. In particular, thread < block < grid.

There are two key differences between Snap's levels and those in CUDA. First, unlike CUDA, Snap does not model a a three-dimensional grid or block structure, say through different levels for grid_x, grid_y, or grid_z. Second, there are two "levels", *warp* and *warp group*, noticeably absent from our hierarchy.

These differences are a deliberate design choice. On the hardware, the units of each level are arranged in a single linear order, and the three-dimensional structure is simply *interpretations* of this linear ordering, not distinct hardware resources. Similarly, warp and warp group 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 warp group, which only became a meaningful construct with the release of the Hopper architecture , consists of 128 threads aligned to 128.

Rather than embedding these interpretations into the language as special cases by introducing new levels, Snap provides constructs that let users group units of a given level . With these constructs, users can express multi-dimensional structures and define custom groupings with specific quantities and alignments.

The primary mechanism for organizing units at each level is a *privilege*.

3.2 Privileges & Privilege Classes

Privileges are the central concept of in Snap, allowing it to determine which compute resources are being requested by the programmer, whether they are available in the program's context, and, once provided, if those resources are sufficient for a given operation.

A *privilege* is a level—grid, block, or thread—parametrized by a static constant n, specifying the number of units at that level. A privilege is written as level[n]. For example, thread[2] denotes a privilege of two threads, block[4] denotes a scope of four blocks, and so on.

Privileges also carry *alignment information*: a privilege parameterized by n aligned to n. In this way, a warp is is simply a desugaring of thread[32], and a warp group is a desugaring of thread[128].

Finally, privileges have a *partial order* defined on them. We say that $\mathtt{level}_2[\mathtt{n}_2]$ is a *higher privilege* than $\mathtt{level}_1[\mathtt{n}_1]$ iff either $\mathtt{level}_1 < \mathtt{level}_2$, or, if $\mathtt{level}_1 = \mathtt{level}_2$, then $\mathtt{n}_2 \,\%\, \mathtt{n}_1 = 0$. The condition $\mathtt{n}_2 \,\%\, \mathtt{n}_1 = 0$ may appear somewhat mysterious at first, but we will see why it is required in Section 3.2.2.

In Snap, *both code and data* are bound to privileges . Code is bound to a set of privileges, called a *privilege class* meanwhile data is bound to a single privilege.

3.2.1 Privileges for Code, or Privilege Classes.

Privilege classes capture what compute resources a program has access to. Each function starts out with a top-level privilege class that defines the total available resources that it *assumes* are available to it. As a shorthand, when we refer to a code's privilege, we mean the highest privilege available to it in its class. In Figure 6, the privilege class is declared on line using the notation @requires(grid=..,block=..) Starting from this top-level declaration, programmers shape the current privilege class in the function's body through two constructs, group and split.

Group. The group construct bundles privileges available in the current privilege class into "groups" of a particular granularity. Intuitively, group narrows the current privilege class and replicates it across collections of a finer granularity. For example, Figure 6, line shows a grouping of a single block, while line in Figure 5 shows a grouping of 32 warps, that will ultimately invoke a tensor-core operation.

```
# Example 1
with group(thread[2]):
    # Illegal because block > thread
    with group(block[1]):
        pass
# Example 2
with group(block[1]):
    # Illegal because 1 % 2 != 0
    with group([block[2]]):
        pass
```

Fig. 7. Illegal uses of group.

An invocation of `group(level[n])` is allowed if and only if the privilege class contains privilege higher than `level[n]`. For example, `group(warp[32])` on in Figure 5 is allowed because the privilege class of the function declares that the top level on line starts out with at least 32 threads.

Once `group(level[n])` is invoked, it modifies the current privilege class in two key ways. First, it removes all parents of `level[n]` from it. Second, the number of available units for `level` is set to exactly `n`. Combined, these rules eliminate malformed programs that would result in unsatisfiable requirements. We shown some examples of illegal `group` statements in Figure 7.

Split. Unlike `group` that is used for replication over smaller, equal sized resource sets, `split` is used for sharding current available units into unequal sized resource sets. For example, line in Figure 6 shows a `split` of a `thread[32]` privilege that is then used to execute a warp-level tensor-core operation.

The `split` operation reflects unordered composition that is not mere replication, and all branches of the `split` execute in parallel. Once `split level` is invoked, the privilege classes of its branches diverge. A split construct will eventually compile to an if-statement during code generation.

```
# Example 1
with group(thread[32]):
 match split(thread):
   case 32:
     simple_mma(..)
   # Illegal! 32 + 1 > 33
   case 1:
       pass
```

Fig. 8. Illegal split that exceeds available privileges.

Therefore, to satisfy a `split` operation, Snap checks that the *cumulative sum* of the privileges requested by each branch of the `split` can be satisfied. For example, if we were to change the program in Figure 6 slightly (on line), to the one shown in Figure 8, Snap would throw a compile-time error. Finally, because each privilege assumes alignment, every branch of the split must also be aligned. In other words, not all `split`s whose sizes sum to less than or equal to the available units are necessarily valid. Figure 9 shows an example of this constraint. This alignment constraint is also why we compare privileges by checking whether one quantity is a multiple of the other.

```
# Example 2
with group(thread[33]):
 match split(thread):
   case 1:
     pass
   # Illegal! 32 + 1 <= 33,
   # but the second branch
   # is not aligned by 32.
   case 32:
     simple_mma(..)
```

Fig. 9. Illegal split that violates alignment.

Inside each branch that requests `n` units, all parent levels of `level[n]` are removed from its privilege class, and the number of available units for `level` is set to exactly `n`, and, within that scope, all resource requirements are checked against that privilege class.

3.2.2 *Privileges for Data.*

Like code, data also has a privilege attached to it that dictates the set of privileges that must be held for reading and writing to it. Ultimately, it is this interaction between privilege classes for code and privileges for data that let us make meaningful statements about the set of privileges available at runtime. The precise details of this interaction are formalized in Section 4.

In this section, we focus tracking thread-local stack allocations and local variables. We defer discussion of pointers to memory to Section 3.5, where we will show how shared views of memory are lowered and restored through the compute hierarchy.

For thread-local data, its privilege is specified in its declaration using the `@ level[n]` syntax. For example, an integer variable v is declared with `thread[1]` privilege with the syntax `v : int @ thread[1]`. If not explicitly annotated, Snap infers a variable's privilege to be the privilege of the code where it was declared. For example, `MMA_N`, `num_blocks_n` are all inferred to be at `grid[1]` scope on lines in Figure 6.

The rules for reading and writing to data are asymmetric: higher-privilege data can flow into lower-privilege data, but not the other way around. In particular, a value coming from a higher-privilege variable can be written into lower-privilege data, but a value coming from a lower-privilege variable cannot be written into higher-privilege data. In short, the rule is: "read up, write down".

If a value is read without being written into a variable, for example, in an `if` statement, the above rule is enforced using the privilege the code that is executing the read. For example, branching on a `thread[1]` variable within `block[1]` privilege code is rejected by our compiler, as it constitutes a "read down".

```
with group(block[1]):
  id : int @ thread[1] = id()
  # Illegal read of ID from
  # block[1] scope.
  if (id)
    __syncthreads();
```

Fig. 10. Illegal read of `thread[1]` variable.

In Figure 6, we can see rule "read up" in action on lines . Meanwhile, "write down" is being used to set up variables on line .

Intuitively, `@ level[n]` tells Snap the *frequency* at which a value changes in space . For example, a `grid[1]` value is the same for all threads in a grid, whereas a `block[2]` value changes every two blocks. The rules described above enforce that the frequency declared for each variable is respected throughout the program. For instance, rule (1) allows variables with higher replication frequency to read from lower-frequency variables without violating their declared frequency. Conversely, higher-frequency variables cannot be used to write into lower-frequency variables, since that would violate the variable's declared frequency.

By tracking frequency in this way, Snap can ensure that threads that with a given privilege can only read variables that change at the same frequency or higher. This guarantees that all threads with that privilege follow the same control flow, and that our accounting of available privilege class remains consistent with the program's runtime behavior.

Until now, we have discussed up rules for reading and writing to variables that change at different frequencies, but we have not yet set up a mechanism for actually actually setting up variables that diverge and change at different frequencies. The main tool for doing this is the `id()` function.

3.3 The `id()` Function

As opposed to exposing users to special hardware `blockIdx.x`, `threadIdx.x` variables directly, Snap provides the `id()` function instead. The `id` function returns the relative index, or a "label", for a given compute privilege. The interpretation of the `id` function depends on both the privilege of the variable it is being written to, and the privilege class 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 to locate the tile that each block is ultimately charge of computing. In Figure 3, however, a call to `id` in `thread[32]` scope at line 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.

3.4 Collective Operations

With privilege classes for code and data in place, we can now turn our attention to Snap's reasoning about collective operations. By explicitly tracking privilege classes for code, Snap statically guarantees that collective operations are semantically valid by ensuring that they are invoked with sufficient privileges.

There are two ways to access collective operations in Snap. The first is to use intrinsics provided by the compiler. These intrinsics have built-in privilege classes attached to them, which the compiler can directly enforce. In Figure 3, the tensor-core call at line is made using one of the intrinsics provided by our compiler. The second, more flexible approach allows users to extend these operations using the `unsafe` construct, explained in Section 3.8. Using unsafe code, users can inline a collective operation—often a single assembly instruction—into a Snap function and ensure that the function's top-level requirements, specified with `@require`, faithfully describe the privileges required to execute that instruction. Then, Snap will check the call to the function as any other function call, enforcing the correct privilege. The rules for checking function calls are described in Section 3.7.

A quirk of some collective operations is that, although they need to be invoked with higher compute privileges, they often produce results with lower data privileges. For example, the tensor-core operation in Figure 3 is invoked at `thread[32]` granularity but generates output at the `thread[1]` level. That is, even though the call to the tensor call is a convergent operation requiring the `thread[32]` compute privilege, it produces divergent output with `thread[1]` data privilege. To be able to handle these operations be must be able to allow `thread[1]` state to coexist with the `thread[32]` privilege.

One potential solution is to artificially coarsen the granularity of the operation by having the intrinsic internally scatter the `thread[1]` output into a `thread[32]` variable. However, this approach is often unacceptable for low-level CUDA performance turning. Often, performance engineers need to manually schedule and interleave memory and arithmetic instructions to hide latency, with loads or stores occurring well in advance of their dependent operations and interleaved with other instructions. If we coarsen these operations, we will disallow these optimizations entirely.

In Snap, however, this behavior does not pose a problem. Because of the "write down" rule, registers can be set up well in advance of the operation (on line in Figure 3, for example) without violating correctness.

3.5 Memory

In Snap, there are three mechanisms for obtaining a pointer type. The first method is to directly allocate data. The second method is to obtain offsets into existinh data by using the `partition` construct and the `claim` construct. The `partition` and `claim` constructs mirror the `group` and `split` constructs, and are used to lower the data privilege associated with memory.

3.5.1 Allocations in Different Memory Spaces.

In Section 2.2, we saw that the GPU has distinct memory spaces, like global, local, and shared memory. Local allocations reside in either the stack or registers, which Snap can distinguish upfront. Shared memory pointers are explicitly annotated with the `shared` keyword, while global pointers are unannotated.

The primary memory resource Snap needs to manage are shared memory allocations. Local memory allocations are just stack allocations, and don't need special treatment. We also assume that sufficient global memory has been allocated before the kernel launch and don't provide a mechanism for runtime allocations—this is typically the case in real-world programs.

In CUDA, shared memory is a limited resource, typically a few hundred kilobytes per block. Programmers manually manage this memory by bumping pointers to allocate offsets for different data. Each allocation's pointer must be carefully tracked throughout the program, and exceeding the available space will lead to a runtime error.

Snap simplifies shared memory management 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 `block[1]` privilege. An example of such an allocation is shown on lines of Figure 6. During compilation, Snap 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 Snap 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 . We use standard techniques to conservatively bound memory usage in Snap's type system.

3.5.2 Privileges for Memory.

In Section 3.2.2, we focused on tracking for primitive data types and stack allocations, where each thread maintained its own local copy of the data. In contrast, pointers into memory, such as global or shared memory, do not represent distinct copies, but instead refer to the same underlying storage that may be accessed simultaneously by multiple threads. To use them, we must first *lower* their privileges, allowing individual threads to write different values to different locations, and then *restore* them back to their original privilege before execution continues.

As with variables in Section 3.2.2, lower-privilege data may read from higher-privilege data without restrictions. For writes, however, Snap enforces that all pointers into memory be written with `thread[1]` privilege; they therefore must be lowered before they can be safely updated.

Partition. The `partition` construct plays the same role for memory that `group` plays for compute. It is used to *lower* a memory object from a higher privilege to a lower one by assigning each lower privilege a distinct offset into the original pointer. It takes the original name of a memory object, `base_name`, an indexing function `f`, and a target privilege `level[n]` to which the memory is being lowered, and produces a new name, `new_name`, for the partitioned memory. In Figure 6, for example, we partition the `C` buffer to `block[1]` privilege on line first, and then lower it further to `thread[1]` privilege on line so that we can actually write to it.

Once a memory object has been partitioned, the original name referring to it goes out of scope, and the memory can only be accessed through the new name introduced by `partition`. Each use of this new name applies the indexing function `f` to compute the true offset of the access. Within the scope of the `partition` statement, the original name cannot be repartitioned. Because `split` denotes unordered composition, Snap prohibits sibling `split` branches from partitioning the same memory object. When the scope of the `partition` statement ends, Snap automatically inserts the necessary synchronization to restore the memory to its original privilege. The mechanism for inserting and optimizing these synchronization points is described in Section 3.9.

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 lower-privilege privilege receives a distinct offset into the underlying array, and the resulting partition is free of data races. When the indexing function is not injective, multiple privileges 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 Snap's goals, and our guarantees continue to hold even when indexing functions are not injective. Because Snap automatically inserts the necessary synchronization before the original memory name is restored to its higher privilege, Snap 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.

We believe that extending Snap to reason about data races and out-of-bounds accesses is a promising direction for future work.

Claim. The `claim` construct is the analogue of `split` for memory . Using `claim`, the entire memory region can be moved to a lower privilege.

Unlike `partition`, `claim` does not take an indexing function. The `claim` construct takes the original memory's name, the target privilege to lower to, and a new name to assign to the lowered memory. The `new_name` is accessible only within a single, explicit `split` branch; sibling branches are not permitted to read or write from this memory. In Figure 6, we use the `claim` operation on line to lower the memory to `thread[32]` privilege.

As with `partition`, a `split` construct introduces its own scope, and the original name is inaccessible within this scope.

3.6 Asynchrony

3.7 Function Composition

To check whether a function is valid, Snap needs to ensure that the call site can provide the set of privileges that a function will attempt to request over its execution, and that the arguments passed to the function also respect the privileges declared for its input and output types. Both requirements are represented explicitly in the function signature in the `@require` construct that we seen in Section 3.2 and Section 3.5.

Like mentioned in Section 3.2, each function carries a privilege class. We call this top-level privilege class the function's *privilege signature*. The signature specifies the top-level privilege 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 defines the privelege signature of a function. Up to this point, we have described programs written *inside* the function 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 privilege class for the function.

Compute privileges. At a function's call site, then, we need to ensure that the function's privilege signature can be satisfied. Using the same mechanism as Section 3.2 and Section 3.5, Snap can track the current compute privilege of the program, as well as how much shared memory is available for use. Through this, Snap can verify that only functions whose privilege signature can be satisfied are called.

Per-argument checks. In Snap, 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 the exact data privilege, or at the higher data privilege. For pointers, we allow passing in pointers that live at higher data privilege if the pointer is marked as `const`, indicating that it will only be used for reading (line in Figure 3). Otherwise, we require that pointers match the exact scope of the function signature.

3.8 Unsafe

3.9 Compiler Internal: Synchronization

4 Formalism

Having introduced the full Snap language, we now describe Bundl, a core calculus that formalizes its most fundamental aspects. Bundl statically tracks compute and data privileges, and we would like to use it to prove that well-typed Snap programs successfully evaluate and will never get stuck trying to execute operations for which they do not have the required compute resources. To that end, in this section we describe the Bundl's type system and operational semantics—in particular how it manages compute and data privileges—and build up to a formal proof of type-and-privilege safety.

4.1 Bundl Type System

Bundl's core idea is type-level tracking of the program's current compute privilege. To achieve this, we borrow techniques from the literature on *coeffects* [11] and *dependency tracking* [1]. In particular, the current compute privilege 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 compute privilege of e and s.

The πs themselves are pairs of levels h of the GPU hierarchy at which each privilege lives (either `Thread`, `Block` or `Grid`), and numbers tracking how many copies of that level each privilege possesses. So, for example example, a (`Thread`,4) in Bundl would directly correspond to a `thread[4]` in Snap.

Meanwhile, each variable in the typing context has a data privilege attached to it. Only when these two privileges match can data be read or written from a variable. This requirement is made manifest

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

$$\frac{x :^{\pi} \tau \in \Gamma}{\Gamma \vdash^{\pi} x : \tau}\ \textsc{T-Var}$$

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

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

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

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

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

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

Fig. 11. Core typing rules of Bundl. The typing rules presented here are a simplified selection of the full rules, which can be found in Appendix A.2.

in the T-Var rule, which can be found in Figure 11, along with the other key typing rules of Bundl. The premise of this rule also reveals that the context Γ maps variables to both types τ and also the privileges π with which they may be read and written.

The rules we discuss in detail in the section fall into two main categories: rules for managing the current compute privilege and rules for managing data privileges.

4.1.1 *Compute Privilege Management.*

In Snap, there are two operations which manage compute privileges: codegroup and `split`. In Bundl, to better tease apart the subtleties of privilege attenuation, we introduce a third construct, `destruct`.

Bundl's `group` statement directly corresponds to Snap's, and is checked by the T-Group rule. Given some privilege that can be divided into q equally sized smaller privileges π, the statement `group` $q\,s$ will check with privilege $q \cdot \pi$ provided that s itself checks with privilege π. Because the starting privilege is of the form $q \cdot \pi$, we do not need to check for alignment.

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

The final rule for managing compute privileges is T-Destruct, which handles downward movement from one level of the GPU hierarchy to another and makes explicit in Bundl programs exactly where such movement occurs. The $\downarrow$ operation on πs "destructs" the privilege into a many smaller privileges 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 Bundl to describe the number of blocks per grid and threads per block. Because Bundl is abstracted over these B and T values, instead of tracking classes of privileges the way that Snap does, it only tracks the top-level privilege described in Section 3.2.

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

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

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

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

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

Fig. 12. Core semantic rules of Bundl. As with the typing rules, we present only a simplified selection of the full rules, which can be found in Appendix A.3.

Because $\downarrow$ is only defined on two inputs, the rule enforces that one can only `destruct` their compute privilege when in possession of exactly one grid or block.

4.1.2 Data Privilege Management. The data privilege management rules mirror those handling compute privilege, with one operation for data corresponding to each operation for code.

The `partition` operation corresponds to grouping a compute privilege. The typing rule for this operation, T-Partition, requires that the partitioned variable x is in the context with the same data privilege as the current compute privilege. However, it then renames the variable into a fresh variable y with a new privilege π/c, a division of all the computational resources in π among c equally-sized smaller privileges. Checking of s then proceeds with the original privilege π: partitioning data with a new data privilege does not change the current compute privilege. Note, however, that the original name of the variable is not visible in s; only the new name may be accessed from within the partition.

In the T-Claim rule, on the other hand, we do change our compute privilege and our data privilege at the same time. Unlike `partition`, which divides up a piece of data equally among all the privileges that divide its current one, the `claim` operation gives the entire variable one smaller privilege, and then checking of s proceeds with that attenuated privilege. This represents a minor difference from Snap: whereas Snap can use additional static analysis to ensure that the claimed variables are not accessed multiple times across sibling `split`s, Bundl instead links the data privilege of the variable to the compute privilege of the code claiming it, ensuring that they remain linked.

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 child privileges in the same manner as T-Partition.

4.2 Bundl Semantics

To reflect the fact that GPU programs execute in parallel across numerous threads, we model the semantics of Bundl in the style of Turon et al. [15], using a two-level small step judgment. We present the key rules of this semantics in Figure 12.

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.

The thread-level judgment has the shape $\eta,\sigma,\Sigma,t,b,p \vdash^{\pi} s \rightsquigarrow s' \dashv \eta',\sigma',\Sigma'$. The ηs in this judgment represent the thread's local memory, the σs represent the shared memory, the Σs represent global memory, and the s's represent the statement being executed by the thread. The primed versions of each of these are the output of the step judgment. 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 verifies privileges*. The same way a program can get stuck if a value does not have the right type, the semantics of Bundl also get stuck if code attempts to access data or invoke commands with the wrong privilege. This is an artifact of Bundl and is not present in Snap itself, but will allow us to prove later in Section 4.3 that well-privileged programs will always execute with the same privilege that the type system checked them against.

The other variables in the judgment track the positions of threads within their scopes. The t is the thread's id, the b is the id of the block in which the thread lives, and the p is the relative position of the thread in the compute privilege with which it is executing. For example, threads with a (`Thread`,4) privilege each have a unique p between 0 and 3, while threads with a (`Block`,16) privilege will have the same p within in each block, between 0 and 15 depending on the block.

The semantic rules for privilege management involve manipulating this p value to track which threads take which code paths when privileges are split or grouped. Notice that in the S-Program rule, the thread stepping judgment always begins with privilege (`Grid`,1): all the privilege management rules are congruences, handling further evaluation with lower privileges as determined by the particular syntactic construct used.

The first set of these privilege management rules handle the `split` construct. They enforce the same alignment requirements as the T-Split rule. The rules choose which path to take based on the value of p: if p is less than the size of the left-hand privilege n_1, execution of s_1 will proceed with that lower privilege via the S-Split-Left rule, while if p is between n_1 and n_2 execution of s_2 will proceed via the S-Split-Right. In this latter case, we execute s_2 with a new p value that subtracts off the value of n_1. As an example, if we encounter a `split`(2,2){s_1}{s_2} with a (`Thread`,4) privilege, the first two threads will go "left" and execute s_1 with a (`Thread`,2) privilege, while the third and fourth threads will go "right" and execute s_2 with a (`Thread`,2) privilege.

The S-Group rule is much simpler, as every thread with a compute privilege that encounters this operation will execute the sub-statement s. What changes across threads is how their p value will be modified by the `group`ing operation. In each case, the p of the thread is reduced modulo n, where n is the size of the privilege with which s is executed. This effectively recolors all the threads privilege the new, smaller, privilege with which they will see s.

The S-Destruct-Block rule for `destruct` behaves similarly to S-Group. However, as p is an index into the current privilege, it is always 0 when a privilege has size 1. The `destruct` statement can only be executed with a size 1 privilege, and so p is not particularly useful here. To compute the new p for the execution of s with a privilege of size T, we reduce the thread id modulo T and continue executing s with that as the new value of p. A similar rule exists for destructing a `Grid` privilege into a size B `Block` privilege.

These rules take great care to ensure that p always describes the relative position of a compute resource within its privilege; the payoff is that Bundl's semantics can later use this p value to model the way that Snap automatically adjusts indices into data when partitioning a data privilege.

Beyond these key rules for compute privilege management, we have modeled many of the other features of Snap, such as asynchronous operations and thread synchronization, in Bundl. To handle features like these we equip the operational semantics with additional structure, including sets of semaphores [5] for thread syncs and a stack of effect handlers for handling deferred asynchronous computations in the style of Ahman and Pretnar [2]. We have elided these details here for simplicity, but interested readers can see further details in Appendix A.3.

4.3 Bundl Theorems

Theorem 4.1. *(Type-and-Privilege 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 $\eta,\sigma,\Sigma,t,b,p \vdash^{\pi} \rightsquigarrow s' \dashv \eta',\sigma',\Sigma'$ where η', σ', and Σ' are well-typed by an extension of Γ'.

Proof. Via the usual progress and preservation lemmas. The full proof of these lemmas can be found in Appendix A.4. □

In addition to the usual guarantees about type safety, this theorem also guarantees that programs will never get stuck trying to execute operations for which they do not have the required compute resources. By instrumenting Bundl's operational semantics with privileges, we allow our safety theorem to guarantee that the dynamically-realized privilege matches the one inferred by the type system. It is, however, worth noting that this soundness theorem is a safety theorem, not a liveness theorem: it guarantees that if all threads that can reach a point in the program with statically-determined privilege π do reach that point, then those threads will be logically grouped according to that same privilege π. It does not, however, guarantee that all threads that can reach the point will eventually do so. Indeed, in the presence of nontermination, liveness does not hold: a subset of the threads could `split` off and loop forever. While we believe the liveness version of this theorem holds for a terminating fragment of Bundl, it is beyond the scope of this paper—such proofs are notoriously challenging and are often research contributions [3, 6, 15] in and of themselves. We hope to tackle this proof for Bundl in future work.

5 Compilation

If a program typechecks, Snap lowers the program into a CUDA file, which does not contain any run-time checks. The CUDA file is then compiled by `nvcc`, NVIDIA's closed-source compiler, to produce a machine-executable. As Snap is an imperative language that provides the same level of abstraction as CUDA, there is a one-to-one mapping between most language constructs and their CUDA counterparts.

Snap is implemented as a compiler tool on top of the CLANG/LLVM compiler stack. Textually, Snap programs are written as the examples described in the paper, but the constructs themselves lower to traditional C++ statements that our frontend tool can analyze and generate code for. This is accomplished by making liberal use of attributes, macros, and other sugaring features of C++.

- Make correspondence to CUDA clear
- Would be awesome to have a super clear figure about how the compilation step actually works (show the one-to-one mapping, though it is slightly more complicated)
- Make optimizations (lazy sync, commit group etc) as a separate optimization section.
- highlight how new function interfaces work out: we add extra arguments

6 Evaluation

We evaluate Snap in the context of the four main questions:

RQ1 Can Snap express a diversity of CUDA programs?
RQ2 Can Snap express programs that use advanced GPU features?
RQ3 Can Snap achieve performance competitive with existing, speed-of-light CUDA code?
RQ4 Can Snap catch bugs that CUDA would otherwise allow?

Although the following question is not central to our paper's main claims—since library design ultimately reflects choices made by programmers, and our language allows abstraction design much like any other language—we were, nevertheless, curious to explore it based on empirical and anecdotal observation:

RQ5 Can Snap help build compositional libraries that users can use with confidence?

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 on this hardware are already highly optimized 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.

Machine Specification

The NVIDIA H100 SXM5
The NVIDIA 4070 SuperTi GPU

Experimental Setup

When Snap produces a CUDA program, we compile the program with `nvcc` version with flags .
To measure performance, we run

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

Single-pass Parallel Prefix Scan with Decoupled Look-back here!

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

To test whether Snap can express programs that use advanced features of modern GPUs, we write a matrix multiplication for the `bf16` datatype for the **H100**.
The H100 `bf16` matrix multiplication pushes several language features to the extreme.

6.3 RQ3: Can Snap achieve performance competitive with existing, speed-of-light CUDA code?

6.4 RQ4: Can Snap catch bugs that CUDA would otherwise allow?

To

6.5 Can Snap help build compositional libraries that users can use with confidence?

We use Snap to build an equivalent of NVIDIA's CUB library []. CUB

CUB occupies a unique design point in the GPU library ecosystem. Unlike other many libraries like cuBLAS, cuDNN, cuSparse which provide global interfaces that users can call and configure, CUB provides a device-side library, organized into different tiers.

CUB also serves as an interesting comparison due it's focus on *collective reduction operations* that call into hardware primitives (like warpshuffles).

7 Limitations

Make sure intro highlights all of this/at least signals to them.

- Can make sub-optimal decision regarding commit group
- Grid sync: we don't handle right now:
- Static pipeline only
- Not about CPU & GPU: only about GPU

8 Related Work

GPU Programming Languages.

Theoretic Foundations. The design of Bundl is heavily inspired by existing work on coeffect systems [11, 12] and dependency tracking [1]. Coeffects allow type systems to talk about how programs depend on their environments, and have achieved widespread use in Rust [13] in the form of linear types. In Snap and Bundl, privileges act like coeffects by describing what compute resources are necessary for programs to execute. Dependency tracking calculi, meanwhile, allow type systems to track how data and code depend on each other, and have commonly been used to implement secure information flow analyses [4]. In Bundl, low privilege data is unable flow into high privilege contexts, and we use dependency tracking to capture this restriction in Bundl's type system.

9 Conclusion

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] 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

[3] 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

[4] 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

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

[6] 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

[7] 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

[8] 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

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

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

[11] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2013. Coeffects: Unified Static Analysis of Context-Dependence. In *Automata, Languages, and Programming*, Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg (Eds.). Springer, Berlin, Heidelberg, 385–397. doi:10.1007/978-3-642-39212-2_35

[12] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. *SIGPLAN Not.* 49, 9 (Aug. 2014), 123–135. doi:10.1145/2692915.2628160

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

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

[15] 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

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

A.1 Basic Definitions

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

Privileges 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 Snap allows us to move along this isomorphism, from left to right. For clarity, in Bundl we split these operations into three: a `split` operation that can split privileges into multiple smaller ones, and a `destruct` operation that directly moves us along the isomorphism, and a `group` that divides our current privilege into equal sized parts.

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

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

We also define division of privileges 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 privileges 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 Typing Rules

A.2.1 Expressions.

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

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

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

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

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

A.2.2 *Statements.*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.3 Complete Bundl 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 partition 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 privilege 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 *Privilege Management Rules.*

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

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

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

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

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

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

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

$$\frac{}{\eta,\sigma,\Sigma,t,b,0,\Psi,\Phi\vdash_m^{\pi} \mathtt{destruct\ in\ skip}\rightsquigarrow \mathtt{skip} \dashv_m \eta,\sigma,\Sigma,\Psi,\Phi}\ \textbf{S-Destruct-Done}$$

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

$$\frac{}{\eta,\sigma,\Sigma,t,b,p,\Psi,\Phi\vdash_m^{\pi} \mathtt{group}\ q\ \mathtt{skip}\rightsquigarrow \mathtt{skip};\dashv_m \eta,\sigma,\Sigma,\Psi,\Phi}\ \textbf{S-Group-Done}$$

A.3.3 Thread Synchronization. We define a **size** operation on privileges to compute the size of a partition (the number of individual threads contained within it). The operation is defined as follows:

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

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

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

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

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

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

A.3.4 Asynchrony.

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

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

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

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

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

A.3.5 Variables and Memory.

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

A.4 Theorems and Proofs

Note that in this section we assume no out of bounds array accesses. In general Snap (and by extension Bundl) 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

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

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

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

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

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

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

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 $\pi \vdash p$, 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 match the premises of the evaluation rules, so these rules are 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.

Privilege 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 $(\texttt{Block},1)$ or $(\texttt{Grid},1)$, 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 $(\texttt{Thread},T)$. 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) privilege with p' as our partition 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 $\Gamma, y :^{(\texttt{Thread},1)} \texttt{async}\ \ \tau[\,]^{l} \vdash \eta',\sigma',\Sigma'$, then we can either step s with $(\texttt{Thread},1)$ privilege 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 $\Gamma \vdash^{\pi}_{m} s$.

Privilege 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 partition id p, or if $(h,n_2) \vdash p$ and s_2 steps with partition 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 partition 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 partition 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 $(\texttt{Thread},1)$ privilege. 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

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

  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

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

  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 $\mathbf{fns}\ \Gamma, x_i :^{\pi_i} \tau_i \vdash^{\pi}_{m'} s$. We can take the union of this with Γ to produce $\Gamma, x_i :^{\pi_i} \tau_i \vdash^{\pi}_{m'} s$, which clearly checks s and the output environments.

□