

1 Modular GPU Programming with Typed Privileges

2 ANONYMOUS AUTHOR(S)

3 Abstract

4 1 Introduction

5 CUDA [15] is a low-level, imperative programming language for NVIDIA GPUs. These GPUs are
6 organized into a hierarchy of compute resources. *Threads* are the basic unit of sequential execution;
7 *blocks* are groups of threads that can cooperate through shared scratchpad memory; and the *grid* is
8 the full collection of blocks launched for a GPU kernel. A CUDA program executes in parallel across
9 this hierarchy, but is written from the *perspective of a single thread*.

10 While operations are specified per-thread, some operations are only semantically valid when ex-
11 ecuted *collectively* by a group of threads. The `__syncthreads()` primitive, for instance, synchronizes
12 all threads within a block, and will cause a deadlock if executed by a subset of threads within that
13 block. There are many such collective operations, including tensor core instructions, warp shuffle
14 operations, and other kinds of synchronization primitives. These operations, therefore, require pro-
15 grammers to carefully marshal compute resources and coordinate which threads execute which lines
16 of code. As a result, such collective operations *break the illusion* of threads executing independently.

17 The conceptual clash between regular statements (executed by a single thread), and collective
18 operations (executed by a group of threads) hinders compositional reasoning. In CUDA, function
19 calls, like all other statements, are per-thread operations. However, a function may contain code that
20 executes a collective operation, creating a gap between the per-thread syntax for its invocation and
21 the cooperative semantics of its execution.

22 This gap creates tension between abstraction and correctness, a tension apparent even in widely
23 used CUDA libraries that package common functionality via function interfaces. Consider the
24 following snippet of documentation taken directly from CUB [14], a library of parallel primitives. The
25 documentation describes the `BlockReduce` function [4], in which all threads in a block collaboratively
26 apply a reduction operation, such as a maximum or prefix sum, over an array:

27 `Computes a block-wide reduction for thread0 using the specified binary reduction functor. The first num_valid threads`
28 `each contribute one input element.`

- 29 • The return value is undefined in threads other than `thread0`.
- 30 • A subsequent `__syncthreads()` threadblock barrier should be invoked after calling this method if the collective's
31 temporary storage (e.g., `temp_storage`) is to be reused or repurposed.

32 Through informal documentation, CUB attempts to convey several assumptions implicit in
33 `BlockReduce`'s implementation. First, the function is only well-defined when invoked by all threads
34 in a block. Second, because it accesses memory shared among threads, any subsequent reuse of that
35 storage requires synchronization to ensure all threads have completed their accesses.

36 In effect, CUB attempts to retrofit CUDA with information about the compute and memory re-
37 quirements of collective operations. In fact, by prefixing functions with identifiers such as `Block`,
38 CUB creates ad-hoc "namespaces" for different functions that assume similar invariants. However,
39 without a type system to statically enforce these requirements in CUDA, correct usage of this function
40 depends on the user carefully reading and interpreting the documentation.

41 In this work, we ask the following question: can we provide low-level access to collective oper-
42 ations while statically guaranteeing that they execute with the necessary configuration of compute
43 resources? By reifying these configurations as type-level *privileges*—so named because they dictate
44 which operations are permitted in which contexts—we find that we can. Our key insight is that GPU
45

50 programmers naturally map computations onto different compute resources, and unlike CUDA,
 51 which obscures this mapping, we can expose it in a type system that tracks privileges. This tracking
 52 enforces that collective operations are executed with the necessary resources, while still allowing
 53 users to access low-level interfaces.

54 Our approach is a departure from previous work, which attempts to resolve the mismatch between
 55 per-thread syntax and collective execution by restricting access to low-level operations. Triton [21],
 56 a tile-based, multi-dimensional array language, limits users to programming only at the block level,
 57 which prevents programmers from writing the highest-performance kernels. A variety of functional
 58 languages, such as Futhark [12], provide compile-time guarantees through their type systems but
 59 are machine-agnostic: they cannot expose low-level control over hardware. Some other efforts,
 60 like Descend [13], aim to provide a low-level, safe GPU systems programming language, but lack
 61 support for collective operations like tensor cores. As a result, CUDA remains the de-facto standard
 62 for writing high-performance kernels on modern GPUs.

63 We introduce Snap, a new low-level GPU programming language that guarantees correct usage
 64 of hardware resources by construction. Inspired by coeffext systems [16] and dependency calculi [1],
 65 Snap statically tracks the configurations of compute and memory resources at the type level. We also
 66 present Bundl, a core calculus underpinning Snap that tracks privileges in both its type system and
 67 operational semantics. We provide formal rules for manipulating the privilege of both code and data,
 68 and prove *type-and-privilege safety*, ensuring that Bundl is sound, and that all threads are statically
 69 privileged to run the operations they attempt at run time.

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

- 74 (1) Snap, a low-level GPU programming language with type-level privileges that track compute
 75 and memory resources (Section 3);
- 76 (2) Bundl, a calculus for Snap that tracks privileges in both its type system and operational se-
 77 mantics, along with a soundness theorem guaranteeing that threads only execute operations
 78 for which they are statically privileged (Section 4); and
- 79 (3) An implementation of Snap (Section 5) that demonstrates that Snap can expose modern
 80 GPU features like tensor cores and asynchronous data-movement, and achieve performance
 81 comparable to hand-optimized CUDA implementations (Section 6).

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99 2 Background & Motivation

100 Before diving into Snap's design, we begin with an overview of the GPU's compute and memory
 101 hierarchies and outline the challenges posed by reasoning about them collectively.
 102

103 2.1 Compute Hierarchy

104 In CUDA, programmers launch computations that run on thousands of threads. These threads are
 105 organized into a *compute hierarchy* that defines how work is distributed and scheduled on the GPU. At
 106 the top of this hierarchy is the *grid*, representing all threads launched as part of a single computation.
 107 The grid is divided into *blocks*, each containing a user-specified number of threads. A *thread*, in turn,
 108 is the finest unit of execution, and it is the machine's basic unit of sequential control.
 109

110 Users define the behavior across different levels of the compute hierarchy by defining the behavior of individual threads that constitute
 111 those levels. CUDA allows programs to read built-in identifiers like
 112 `threadIdx.x`—to determine a thread's position within a block—and
 113 `blockIdx.x`—to determine the block's position within the grid—at
 114 runtime. Combining these two IDs, threads can locate themselves
 115 in their share of the full computation.
 116

117 A natural and tempting way to interpret these built-in identifiers
 118 is to think of them as indices into implicit "parallel for-loops" where
 119 each iteration is executing simultaneously. While this view is suf-
 120 ficient to understand CUDA's programming model when threads
 121 do independent work, it quickly breaks down in the presence of
 122 *collective operations*.
 123

124 Unlike most instructions, which are executed by a single thread,
 125 collective operations must be executed collaboratively by a group
 126 of threads. For example, a "warp-level" tensor-core operation is only *meaningful* when invoked by
 127 a collection of 32 threads—a *warp*—acting together. Consider Figure 1 which shows a tensor-core
 128 operation being invoked by a warp on line 14. For the call, each thread sets up its portion of the
 129 operands, and the operation is performed *once* for the entire warp, with the results scattered across
 130 participating threads. If the condition on line 2 were instead `tid > 0 && tid < 30`, making fewer than
 131 32 threads reach line 14, the instruction would be undefined. To make matters worse, the restriction
 132 is not only on the *number* of threads executing the operation, but also on their *alignment*. In this case,
 133 the starting ID of the group of 32 participating threads must be aligned to a multiple of 32. So, if the
 134 condition on line 2 were instead `tid > 1 && tid < 33`, the instruction would still be invalid, even
 135 though 32 threads would execute it.

136 Collective operations make reasoning about CUDA programs challenging because they force
 137 programmers to track the *convergence behavior* of threads. Specifically, programmers must reason
 138 both about how many threads reach a particular point in the program and how those threads are
 139 arranged, accounting for alignment. This difficulty is further amplified due to two reasons. First,
 140 programs may have multiple *points* of convergence, requiring programmers to mentally track the
 141 relative ID of a thread within a logical group as that group changes over the course of a program's
 142 execution. Second, threads may be participating in multiple *levels* of convergence within the same
 143 program. In our example, we only considered the warp-level tensor-core operation, but there are
 144 other collective operations that require convergence at different granularities like the *warp-group*-
 145 level tensor-core operations, which must be issued collectively by four warps, or the block-level
 146 `_syncthreads()` synchronization primitive, which must be executed by all threads within a block.

```

1 int tid = threadIdx.x;
2 if (tid >= 0 && tid < 31) {
3     float A[4];
4     float B[2];
5     float C[4] = { 0 };
6     # Populate A with unique values
7     for (int i = 0; i < 4; i++)
8         A[i] = tid * 4 + i;
9     # Populate B with unique values
10    for (int i = 0; i < 2; i++)
11        B[i] = tid * 4 + i;
12    # Issue a warp-level tensor-core
13    # operation. D = A * B + C
14    asm("mma.sync.aligned.m16n8k8..." +
15        "%0, %1, %2, %3,    /*D*/\n"
16        "%4, %5, %6, %7,    /*A*/\n"
17        "%8, %9,            /*B*/\n"
18        "%10, %11, %12, %13;" /*C*/\n"
19        : "=r"(C[0]), "=r"(C[1]), ...
20        : "r"(A[0]), "r"(A[1]), ...
21        : "r"(B[0]), "r"(B[1]),
22        : "r"(C[0]), "r"(C[1]), ...);}
```

Fig. 1. Invoking a warp-level tensor-core instruction in CUDA.

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

In 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 2 which shows an equivalent rewrite of the CUDA program in Figure 1 using Snap. In Figure 2, 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.

Furthermore, Snap also tracks the *frequency* at which values diverge in *space*. The @ syntax attached to each variable deceleration denotes the rate at which a value changes across compute resources on the machine. For example, a @ `thread[1]` value can diverge per-thread, while a @ `block[1]` can diverge for every block, but not for threads within that block. Using this information, Snap enforces rules for reading from and writing to data. So, if we had attempted to introduce a condition based on `tid` *within* the group's scope, Snap would reject that program at compile time. This is because code executing at a lower frequency cannot read variables that change at a higher frequency than it. We will 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, 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

```

1  tid : int @ thread[] = 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. 2. Invoking a warp-level tensor-core instruction in Snap.

```

1  int x = threadIdx.x;
2  if (x >= 0 && x < 31){
3    float A[4];
4    float B[2];
5    float C[4] = { 0 };
6
7    // k is A's stride
8    A[0] = a_mem[(x/4)*k+(x%4)];
9    A[1] = a_mem[(x/4)*k+(x%4)+4];
10   A[2] = a_mem[(x/4+8)*k+(x%4)];
11   A[3] = a_mem[(x/4+8)*k+(x%4)+4];
12
13   // n is B's stride
14   B[0] = b_mem[(x%4)*n+(x/4)];
15   B[1] = b_mem[(x%4+4)*n+(x/4)];
16
17   asm("mma.sync...");
18
19   // Write back into c_mem
20   // n is C's stride
21   c_mem[(x/4)*n+2*(x%4)] = C[0];
22   c_mem[(x/4)*n+2*(x%4)+1] = C[1];
23   c_mem[(x/4+8)*n+2*(x%4)] = C[2];
24   c_mem[(x/4+8)*n+2*(x%4)+1] = C[3];

```

Fig. 3. Invoking a warp-level tensor-core instruction in CUDA after loading data from memory.

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.

Let us reconsider the tensor-core example in Figure 1, this time initializing the operands of the tensor-core operation from pointers into memory. In Figure 3, A and B are now populated from data in global memory, `a_mem` and `b_mem`. After the tensor-core operation completes, the result is written back to `c_mem`, also in global memory.

In this program, `c_mem` is logically accessed from two different levels of the compute hierarchy. When entering this code, each thread in a group of 32 threads sees the same value for `c_mem`. Then, each thread within these 32 locates an offset within `c_mem` that it can write to (lines 21-24). To restore `c_mem` back to its 32 thread-level ownership, all 32 threads must first synchronize to ensure all per-thread writes have completed. This is similar to the requirement we saw documented in ??.

Similarly to the compute hierarchy (Section 2.1), CUDA programmers are responsible for tracking the logical owner of a piece of memory as it evolves over the course of a program, and for ensuring that appropriate synchronization occurs whenever that ownership changes.

Snap, by contrast, makes the evolution of a memory object's logical ownership explicit in the type system and *automatically* inserts the synchronization required to restore memory to its original ownership. In Figure 4, we show how memory is lowered through the compute hierarchy for the same tensor-core operation introduced in Figure 3. 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. The operation takes the name of the memory to partition, `c_mem`, and lowers it to a single thread, and assigns it a new name, `c_thrd`; Snap will not allow references to the old name `c_mem` within the partition's scope. The `partition` function also takes an indexing function, and each time `c_thrd` is accessed, this indexing function is implicitly applied. Once the `partition`'s scope ends, Snap will insert a synchronization before the first use of the original name, so that all per-thread writes have completed.

We would like to emphasize that preventing data races is not 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 [13], 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 prior work like Descend cannot provide.

```

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

```

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

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 5. This is a standard computation a GPU program might write and is sufficiently complex to reveal the interesting challenges in GPU programming.

The program in Figure 5 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 20-21). Since matrix multiplication offers ample data reuse, to amortize the cost of loading data, we stage A and B in shared memory (lines 46-59). At this point, each thread within the block diverges and loads different elements of A and B. Finally, the program invokes an Amere warp-level tensor-core instruction to compute the output, requiring threads in a warp to converge (line 71). 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 4.

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 an

```

1 @ccuda("global")
2 # Top-level privilege class and required shared memory usage.
3 @requires(grid[1], block[1], thread[32], smem=1280)
4 def mmaTF32NaiveKernel(A: ptr(const(float)) @ grid[1],
5                         B: ptr(const(float)) @ grid[1],
6                         C: ptr(float) @ grid[1],
7                         M : int @ grid[1],
8                         N : int @ grid[1],
9                         K : int @ grid[1]):
# Replicate this code across 1 grid.
with group(grid[1]):
    # @ grid[1] inferred from compute privilege
    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]):
            # SEM 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. 5. Tensor Float 32 matrix Multiplication in Snap.

295 order defined over them, with `thread < block`
 296 `< grid`.

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

302 These differences are a deliberate design choice. On the hardware, the units of each level are
 303 arranged in a single linear order, and the three-dimensional structures are simply *interpretations* of this
 304 linear ordering, not distinct hardware resources. Similarly, warp and warp group are organizational
 305 constructs defined in terms of existing levels. Namely, a warp is a group of 32 threads whose first
 306 thread ID is aligned to 32. A warp group, which only became a meaningful construct with the release
 307 of the Hopper architecture , consists of 128 threads aligned to 128.

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

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

313

314 3.2 Privileges & Privilege Classes

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

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

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

323 Finally, privileges have a *partial order* defined on them. We say that `level2[n2]` is a *higher privilege*
 324 than `level1[n1]` iff either `level1 < level2`, or, if `level1 = level2`, then `n2 % n1 = 0`. The condition `n2`
 325 `% n1 = 0` may appear somewhat mysterious at first, but we will see why it is required in Section 3.2.2.

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

328 3.2.1 Privileges for Code, or Privilege Classes.

329 Privilege classes capture what compute resources a program has access to. Each function starts
 330 out with a top-level privilege class that defines the total available resources that it assumes are
 331 available to it. As a shorthand, when we refer to a code's privilege, we mean the highest privilege
 332 available to it in its class . In Figure 5, the privilege class is declared on line 2 using the notation
 333 `@requires(grid=..,block=..)` Starting from this top-level declaration, programmers shape the
 334 current privilege class in the function's body through two constructs, `group` and `split`.

335 **Group.** The group construct bundles privileges available in the current privilege class into "groups"
 336 of a particular granularity. Intuitively, group narrows the current privilege class and replicates it
 337 across collections of a finer granularity. For example, Figure 5, line 26 shows a grouping of a single
 338 block, while line 22 in Figure 4 shows a grouping of 32 warps , that will ultimately invoke a tensor-core
 339 operation.

340

344 An invocation of `group(level[n])` is allowed if and only if the
 345 privilege class contains privilege higher than `level[n]`. For example,
 346 `group(warp[32])` on 22 in Figure 4 is allowed because the privilege
 347 class of the function declares that the top level on line 22 starts out
 348 with at least 32 threads.

349 Once `group(level[n])` is invoked, it modifies the current privi-
 350 lege class in two key ways. First, it removes all parents of `level[n]`
 351 from it. Second, the number of available units for `level` is set to
 352 exactly `n`. Combined, these rules eliminate malformed programs that would result in unsatisfiable
 353 requirements . We shown some examples of illegal group statements in Figure 6.

354 **Split.** Unlike `group`, which is used for replication over smaller, equal sized resource sets, `split` is
 355 used for sharding currently available resources into unequal sized sets. For example, line 63 in Figure 5
 356 shows a `split` of a `thread[32]` privilege that is used to execute a warp-level tensor-core operation.

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

362 Therefore, to satisfy a `split` operation, Snap checks that the
 363 *cumulative sum* of the privileges requested by each branch of the
 364 `split` can be satisfied. For example, if we were to change the program
 365 in Figure 5 slightly (on line 63-69), to the one shown in Figure 7,
 366 Snap would throw a compile-time error. Finally, because privileges
 367 assume alignment, every branch of the `split` must also be aligned.
 368 In other words, not all splits whose sizes sum to less than or equal to
 369 the available units are necessarily valid. Figure 8 shows an example
 370 of this constraint. This alignment constraint is also why we compare
 371 privileges by checking whether one quantity is a multiple of the
 372 other's.

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

377 3.2.2 Privileges for Data.

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

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

385 For thread-local data, its privilege is specified in its declaration using the `@ level[n]` syntax.
 386 For example, an integer variable `v` is declared with `thread[1]` privilege with the syntax `v : int @`
 387 `thread[1]`. If not explicitly annotated, Snap infers a variable's privilege to be the privilege of the
 388 code where it was declared. For example, `MMA_N`, `num_blocks_n` are all inferred to be at `grid[1]`
 389 scope on lines 13 and 15 in Figure 5.

```

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

```

Fig. 6. Illegal uses of group.

```

1 # Example 1
2 with group(thread[32]):
3     match split(thread):
4         case 32:
5             simple_mma(..)
6             # Illegal! 32 + 1 > 33
7         case 1:
8             pass

```

Fig. 7. Illegal split that exceeds available privileges.

```

1 # Example 2
2 with group(thread[33]):
3     match split(thread):
4         case 1:
5             pass
6             # Illegal! 32 + 1 <= 33,
7             # but the second branch
8             # is not aligned by 32.
9         case 32:
10            simple_mma(..)

```

Fig. 8. Illegal split that violates alignment.

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

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

402 In Figure 5, we can see rule "read up" in action on lines **45 and 46**.
 403 Meanwhile, "write down" is being used to set up variables on line
 404 **18-22**.

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

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

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

420 3.3 The `id()` Function

421 As opposed to exposing users to special hardware `blockIdx.x`, `threadIdx.x` variables directly, Snap
 422 provides the `id()` function instead. The `id` function returns the relative index, or a "label", for a given
 423 compute privilege. The interpretation of the `id` function depends on both the privilege of the variable
 424 it is being written to, and the privilege class that the `id` function is being invoked in. In our example
 425 in Figure 5, we use a call to `id` in `grid[1]` scope at line **18 and 19** to locate the tile that each block is
 426 ultimately charge of computing. In Figure 2, however, a call to `id` in `thread[32]` scope at line **7** will
 427 not return the global thread ID of the threads launched in the kernel, but rather the *relative* thread ID
 428 within the 32 threads, ranging from 0 to 31.

429 3.4 Collective Operations

430 With privilege classes for code and data in place, we can now turn our attention to Snap's reasoning
 431 about collective operations. By statically tracking privilege classes for code, Snap guarantees that
 432 collective operations are semantically valid by ensuring that they are invoked with sufficient privilege.
 433

434 There are two ways to access collective operations in Snap. The first is to use intrinsics provided
 435 by the compiler. These intrinsics have built-in privilege classes attached to them, which the compiler
 436 can directly enforce. In Figure 2, the tensor-core call at line **22** is made using one of the intrinsics
 437 provided by our compiler. The second, more flexible approach allows users to extend these operations
 438 using the `unsafe` construct, explained in Section 3.8. Using unsafe code, users can inline a collective
 439 operation—often a single assembly instruction—into a Snap function and ensure that the function's
 440 top-level requirements, specified with `@require`, faithfully describe the privileges required to execute
 441

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

Fig. 9. Illegal read of `thread[1]` variable.

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 2 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 we 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 tuning. 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 in advance of the operation (e.g., line 14-20 in Figure 2) 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 and third are to obtain offsets into existing data by using `partition` or `claim`. These 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 28-30 of Figure 5. 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 5 declares the amount of shared memory it will require on 2. We use standard techniques to conservatively bound memory usage in Snap's type system.

3.5.2 Privileges for Memory.

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

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

500
 501 **Partition.** The `partition` construct plays the same role for memory that `group` plays for compute.
 502 It is used to lower a memory object from a higher privilege to a lower one by assigning each lower
 503 privilege a distinct offset into the original pointer. It takes the original name of a memory object,
 504 `base_name`, an indexing function `f`, and a target privilege `level[n]` to which the memory is being
 505 lowered, and produces a new name, `new_name`, for the partitioned memory. In Figure 5, for example,
 506 we partition the C buffer to `block[1]` privilege on line 25 first, and then lower it further to `thread[1]`
 507 privilege on line 78 so that we can actually write to it.

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

516 At this point, it is necessary to consider how different indexing functions affect the possibility
 517 of data races. If the indexing function is injective, each lower-privilege privilege receives a distinct
 518 offset into the underlying array, and the resulting partition is free of data races. When the indexing
 519 function is not injective, multiple privileges may write to the same location, introducing a potential
 520 data race. Out-of-bounds accesses are undefined behavior.

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

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

528
 529 **Claim.** The `claim` construct is the analogue of `split` for memory . Using `claim`, the entire
 530 memory region can be moved to a lower privilege.

531 Unlike `partition`, `claim` does not take an indexing function. The `claim` construct takes the
 532 original memory’s name, the target privilege to lower to, and a new name to assign to the lowered
 533 memory. The `new_name` is accessible only within a single, explicit `split` branch; sibling branches are
 534 not permitted to read or write from this memory. In Figure 5, we use the `claim` operation on line 63
 535 to lower the memory to `thread[32]` privilege.

536 As with `partition`, a `split` construct introduces its own scope, and the original name is inacces-
 537 sible within this scope.

540 **3.6 Asynchrony**

541 **3.7 Function Composition**

542 To check whether a function call is valid, Snap needs to ensure that the call site can provide the set of
 543 privileges that a function will attempt to use over its execution, and that the arguments passed to the
 544 function also respect the privileges declared for its input and output types. Both requirements are
 545 represented explicitly via the `@require` construct.
 546

547 As mentioned in Section 3.2, each function carries a privilege class. We call this top-level privilege
 548 class the function's *privilege signature*. The signature specifies the top-level privilege class the
 549 function *assumes* will be provided to it. This signature represents the *minimum amount* of compute
 550 and memory resources the function must be called with. In the example above in Figure 5, line 2
 551 defines the privilege signature. Up to this point, we have described programs written *inside* functions
 552 under the assumption that the resource signature can be satisfied. In Section 3.2, for example, we
 553 used this signature to set up the top-level privilege class for the function.

554 **Compute privileges.** At a function's call site, then, we need to ensure that the function's privilege
 555 signature can be satisfied. Snap compares the current compute privilege with the one required by the
 556 function, and verifies that only functions whose privilege signatures can be satisfied are called.
 557

558 **Per-argument checks.** In Snap, functions have pass-by-value semantics. For arguments, we
 559 distinguish between primitive data types and pointers to memory. For primitive data types, like `int`,
 560 `bool`, and stack allocations, users can pass either arguments that live at or above the data privilege
 561 specified by the function's signature. For pointers, we allow passing pointers that live at higher data
 562 privilege if the pointer is marked as `const`, indicating that it will only be used for reading (line 4 and
 563 5 in Figure 2). Otherwise, we require that pointers match the exact privilege of the function signature.
 564

565 **3.8 Unsafe**

566 Snap also has an unsafe construct that lets users inline collective operations that have not been
 567 explicitly added as intrinsics in the compiler. After using `unsafe`, these operations can be wrapped
 568 in a function whose signature declares the compute privileges required to call it.
 569

570 **3.9 Compiler Internal: Synchronization**

571 As described in Section 3.5, once data has been partitioned, Snap is responsible for
 572 inserting the appropriate synchronization when the partition's scope ends.
 573

574 To determine where these synchronization points need to be inserted, Snap
 575 constructs a *data-control-flow graph* from the program. Each node corresponds to
 576 a partition, and each edge captures program-order precedence: the parent partition
 577 must complete before the child begins. The graph can have backedges introduced
 578 through loops. In this graph, each partition is categorized as a *read* or a *write*
 579 partition. A synchronization point is inserted before entering a partition if either
 580 of two conditions holds:
 581

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

585 Figure 10 shows an example graph with the synchronization points derived from
 586 these two conditions. The inferred synchronization points in Figure 5 have also
 587 been marked on lines **TODO**.

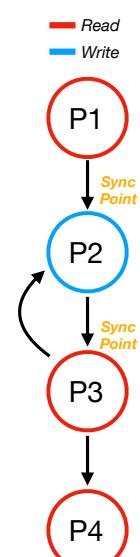


Fig. 10. An example data-

Using this information, Snap emits *wait* operations before a partition begins (to wait for its parent) and *arrive* operations when a partition ends (to signal completion). CUDA exposes a general split-barrier primitive that we use to implement these waits and arrives. For special cases, such as synchronizing an entire block or warp, Snap instead uses primitives like `__syncthreads()` or `__syncwarp()`.

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

Given this setup, naively inserting synchronization immediately after each partition scope would be correct but prohibitively slow. To avoid this, Snap optimizes the inferred waits and arrives in two passes. A wait-motion pass pushes waits downward toward the first use of the partitioned name, and an arrive-motion pass pulls arrives upward toward its last use, reducing unnecessary stalls while preserving correctness.

4 Formalism

Having introduced the full Snap language, we now describe Bundl, a core calculus that formalizes its most fundamental aspects by statically tracking compute and data privileges. We use Bundl to prove that well-typed Snap programs are not only type-safe, but will also never get stuck trying to execute operations for which they do not have the required compute resources.

In this section, we describe 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. By instrumenting Bundl’s operational semantics with privileges, our safety theorem can guarantee that dynamically-realized privileges match the ones inferred by the type system.

4.1 Bundl Type System

The core idea in Bundl is to track, at the type-level, the program’s current compute and data privilege. To achieve this, we borrow techniques from the literature on *coeffects* [16] and *dependency tracking* [1]. In particular, the 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 π s themselves are pairs of levels h of the GPU hierarchy at which each privilege lives (either Thread, Block or Grid), and numbers tracking the size of the logical group denoted by each privilege . Bundl’s π s directly correspond to privileges in Snap.

Each variable in the typing context has a data privilege attached to it; data can only be read or written from a variable when its privilege matches that of the code interacting with it. This requirement is made manifest in the `T-VAR` rule, which can be found in Figure 11. 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.

Figure 11 also shows other key rules, and these rules 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: `group` and `split`. In Bundl, to better model the details of privilege attenuation , we introduce a third construct, `destruct`.

638	$\boxed{\Gamma \vdash^\pi e : \tau}$	<i>(Expression typing)</i>
639		
640	$\frac{x :^\pi \tau \in \Gamma}{\Gamma \vdash^\pi x : \tau}$ T-VAR	
641		
642	$\boxed{\Gamma \vdash^\pi s}$	<i>(Statement typing)</i>
643		
644	$\frac{\Gamma \vdash^{(h,n_1)} s_1 \quad \Gamma \vdash^{(h,n_2)} s_2 \quad n_1, n_2 \text{ align to } n}{\Gamma \vdash^{(h,n)} \text{split}(n_1, n_2)\{s_1\}\{s_2\}}$ T-SPLIT	
645		
646		where $n_1, n_2 \text{ align to } n ::= (n_1 + n_2 \leq n) \text{ and } (n_1 n) \text{ and } (n_2 n) \text{ and } (n_2 n_1 + n)$
647		
648	$\frac{\Gamma \vdash^\pi s}{\Gamma \vdash^{q \cdot \pi} \text{group } q \ s}$ T-GROUP	$\frac{\Gamma, y : \downarrow^\pi \tau[]^l \vdash^\pi s \quad l \neq \text{Local}}{\Gamma, x :^\pi \tau[]^l \vdash^\pi \text{lower } x \text{ into } y \text{ in } s}$ T-LOWER
649		
650		
651		
652	$\frac{\Gamma, y :^{(h,n/c)} \tau[]^l \vdash^{(h,n)} s \quad c n \quad l \neq \text{Local}}{\Gamma, x :^{(h,n)} \tau[]^l \vdash^{(h,n)} \text{partition } x \text{ into } y \text{ by } c \text{ in } s}$ T-PARTITION	$\frac{\Gamma \vdash^\pi s}{\Gamma \vdash^\pi \text{destruct in } s}$ T-DESTRUCT
653		
654		
655	$\frac{\Gamma, y :^{(h,n')} \tau[]^l \vdash^{(h,n')} s \quad n' \leq n \quad l \neq \text{Local}}{\Gamma, x :^{(h,n)} \tau[]^l \vdash^{(h,n)} \text{claim } x \text{ into } y \text{ at } n' \text{ in } s}$ T-CLAIM	
656		
657		
658		

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.

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 $\text{group } q \ s$ will check with privilege $q \cdot \pi$ provided that s itself checks with privilege π . The starting privilege is of the form $q \cdot \pi$, and encodes Snap’s alignment requirement.

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 π s “destructs” the privilege into a many smaller privileges at a lower level; $\downarrow(\text{Grid}, 1) = (\text{Block}, B)$ and $\downarrow(\text{Block}, 1) = (\text{Thread}, T)$, where B and T are parameters to a particular instantiation of Bundl to describe the number of blocks per grid and threads per block.¹ Because \downarrow is only defined on (Grid,1) and (Block,1), the rule enforces that one can only destruct their compute privilege when they have privileges for exactly one grid or block.

4.1.2 Data Privilege Management. The mechanism for managing data privileges mirrors that of compute privileges, with each operation for data corresponding to an operation for code.

The partition operation is analogous to group-ing a compute privilege. The typing rule for this operation, T-PARTITION, requires that the data privilege of the variable to be partitioned, x , is the same as the current compute privilege. After partitioning, a fresh variable y is introduced with a new privilege π/c . We use π/c as a shorthand to denote a division of a privilege π into c equally-sized,

¹Bundl is abstracted over these B and T values, so instead of tracking classes of privileges the way that Snap does, it only tracks the top-level privilege described in Section 3.2.

$$\begin{array}{l}
 \frac{L(t), S(b), \Sigma, t, b, 0 \vdash^{(\text{Grid}, 1)} s \rightsquigarrow s' \dashv \eta', \sigma', \Sigma' \quad P(t, b) = s}{L, S, \Sigma, P \rightsquigarrow L[t \mapsto \eta'], S[b \mapsto \sigma'], \Sigma', P[(t, b) \mapsto s']} \text{ S-PROGRAM} \\
 \\
 \frac{p < n_1 \quad n_1, n_2 \text{ align to } n \quad \eta, \sigma, \Sigma, t, b, p \vdash^{(h, n_1)} s_1 \rightsquigarrow s'_1 \dashv \eta', \sigma', \Sigma'}{\eta, \sigma, \Sigma, t, b, p \vdash^{(h, n)} \text{split}(n_1, n_2)\{s_1\}\{s_2\} \rightsquigarrow \text{split}(n_1, n_2)\{s'_1\}\{s_2\} \dashv \eta', \sigma', \Sigma'} \text{ S-SPLIT-LEFT} \\
 \\
 \frac{p \geq n_1 \quad p < n_1 + n_2 \quad n_1, n_2 \text{ align to } n \quad \eta, \sigma, \Sigma, t, b, p - n_1, \vdash^{(h, n_2)} s_2 \rightsquigarrow s'_2 \dashv \eta', \sigma', \Sigma'}{\eta, \sigma, \Sigma, t, b, p \vdash^{(h, n)} \text{split}(n_1, n_2)\{s_1\}\{s_2\} \rightsquigarrow \text{split}(n_1, n_2)\{s_1\}\{s'_2\} \dashv \eta', \sigma', \Sigma'} \text{ S-SPLIT-RIGHT} \\
 \\
 \frac{\eta, \sigma, \Sigma, t, b, p \text{ mod } n \vdash^{(h, n)} s \rightsquigarrow s' \dashv \eta', \sigma', \Sigma'}{\eta, \sigma, \Sigma, t, b, p \vdash^{(h, q \cdot n)} \text{group } q \ s \rightsquigarrow \text{group } q \ s' \dashv \eta', \sigma', \Sigma'} \text{ S-GROUP} \\
 \\
 \frac{\eta, \sigma, \Sigma, t, b, t \text{ mod } T \vdash^{(\text{Thread}, T)} s \rightsquigarrow s' \dashv \eta', \sigma', \Sigma'}{\eta, \sigma, \Sigma, t, b, 0 \vdash^{(\text{Block}, 1)} \text{destruct in } s \rightsquigarrow \text{destruct in } s' \dashv \eta', \sigma', \Sigma'} \text{ S-DESTRUCT-BLOCK}
 \end{array}$$

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.

smaller privileges. Within the body of the `partition` statement, we disallow any references to the original variable x . Then, we continue checking the body of the `partition` statement against the original compute privilege π because the `partition` has no effect on the current compute privilege.

Unlike `partition`, which divides up a piece of data equally into lower privileges, the `claim` operation, mirroring the `split` construct, gives the entire variable to exactly one, smaller privilege. Accordingly, Bundl needs to ensure that only one branch of a `split` operation, with the appropriate π , can refer to the claimed variable. To ensure that this is the case, the `T-CLAIM` rule links the data privilege of the variable to the compute privilege of the code claiming it by changing *both* at the same time. This represents a minor difference from Snap, which implements additional static analysis to ensure that a claimed variable is only accessed in a single `split` branch.

Lastly, the `T-LOWER` rule mirrors the `T-DESTRUCT` rule; it uses the \downarrow operator to move a variable from one level of the hierarchy to another, distributing it equally among all the child privileges in the same manner as `T-PARTITION`.

4.2 Bundl Semantics

Having explained the key rules of the type system, we can move on to discuss Bundl’s operational semantics. To reflect the fact that GPU programs execute in parallel across numerous threads, we model the semantics of Bundl in the style of Turon et al. [22], 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.²

The thread-level judgment has the shape $\eta, \sigma, \Sigma, t, b, p \vdash^\pi s \rightsquigarrow s' \dashv \eta', \sigma', \Sigma'$. The η s represent the thread’s local memory, the σ s represent the shared memory, the Σ s 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 runtime privilege is present in Bundl, but is erased by Snap during compilation; we will later use it to prove that well-privileged programs will always execute with the same privilege that the type system checked them against.

Other variables in the judgment track the positions of threads within their scopes. The t variable represents the thread’s id, b is the id of the block in which the thread lives, and 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 privileges involve manipulating p 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 rule used.

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 [10] for thread syncs and a stack of effect handlers for modeling 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 Soundness Theorem

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

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 $s', \eta', \sigma', \text{ and } \Sigma'$ such that $\eta, \sigma, \Sigma, t, b, p \vdash^\pi s' \rightsquigarrow s' \dashv \eta', \sigma', \Sigma'$ where $\eta', \sigma', \text{ and } \Sigma'$ 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. \square

It is worth noting that this soundness theorem is a safety theorem, not a liveness theorem. The theorem does 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

²In reality, the GPU’s thread scheduler guarantees that warps execute their threads in lockstep, but this modeling every thread as completely independent is both simpler and a conservative overestimate of the nondeterministic behavior of the GPU.

785 that same privilege π . It does not, however, guarantee that all threads that *can* reach the point will
 786 eventually do so. Indeed, in the presence of nontermination, liveness does not hold: a subset of the
 787 threads could split off and loop forever. While we believe the liveness version of this theorem holds
 788 for a terminating fragment of Bundl, it is beyond the scope of this paper—such proofs are notoriously
 789 challenging and are often research contributions [3, 11, 22] in and of themselves. We hope to tackle
 790 this proof for Bundl in future work.

791 5 Compilation

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

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

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

808 6 Evaluation

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

811 **RQ1** Can Snap express a diversity of CUDA programs?

812 **RQ2** Can Snap express programs that use advanced GPU features?

813 **RQ3** Can Snap match the performance of existing, speed-of-light CUDA code?

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

818 **RQ4** Can Snap help build compositional libraries that users can use with confidence?

820 To perform this evaluation, we use two GPUs. The first is the NVIDIA H100 SXM5, a server-grade
 821 chip that supports tensor-core operations and a dedicated hardware copy engine, the Tensor Memory
 822 Accelerator. Notably, the H100 introduces a new *logical* level called the *warp group* (collection of
 823 4-aligned warps), and we show that our programming model can accommodate this new level.
 824 Moreover, because the H100 has historically served as the primary GPU for large-scale AI training,
 825 many CUDA kernels on this hardware are already highly optimized and achieve near-speed-of-
 826 light performance, providing a rigorous baseline for comparison. To ensure our results generalize
 827 beyond the H100, we additionally test programs on a second GPU, the NVIDIA 4070 SuperTi GPU, a
 828 consumer-grade chip.

829 Like mentioned in Section 5, when name typechecks a program, it produces a CUDA file. We
 830 compile the CUDA file with nvcc version with flags . To measure performance of our benchmark,
 831 we set up the values from an input using a random number generator. We use the average running
 832 time sampled across 10 iterations, with a warm-up of 5 iterations.

834 6.1 RQ1: Can Snap express a diversity of CUDA programs?

835 To answer this research question, we evaluate Snap on two benchmarks: a sequence of matrix
 836 multiplication kernels that grow in complexity and a single-pass parallel prefix scan with decoupled
 837 look-back that requires programmers to think carefully about multiple points of convergence.
 838

839 *Matrix Multiplication.* We chose matrix multiplication as our first benchmark for two main reasons.
 840 First, there exist several implementations that achieve near-peak performance, providing a strong
 841 baseline. Second, matrix multiplication allows for a range of increasingly sophisticated implementa-
 842 tions that stress different parts of the language, making it ideal for evaluating expressiveness.
 843

844 To undertake this exploration, we adapt the codebase from to implement a matrix multiplication
 845 on the `f16` datatype, also known as `sgemm`, in Snap. As this is a `f16` benchmark, it *does not need*
 846 to use advanced GPU features like asynchrony or tensor cores to achieve speed-of-light performance.
 847 We will discuss these advanced features in Section 6.2.

848 We implement several variants of the `sgemm` benchmark, reproduced in Appendix . These include a
 849 naive baseline, a version that exploits memory coalescing, 2D blocking with shared-memory staging,
 850 2D blocking with vectorized loads, and a warp-tiling strategy that effectively introduces an additional
 851 level of tiling from the perspective of a warp. We find that the runtime performance of our Snap
 852 implementations closely matches that of the original versions. This is expected, since the generated
 853 code is nearly identical to the hand-written versions, aside from small differences such as hoisted
 854 expressions and a few index calculations. The downstream compiler (`nvcc`) readily inlines hoisted
 855 expressions that are used only once, so the performance-critical inner loops remain effectively
 856 unchanged.

857 *Single-pass Parallel Prefix Scan with Decoupled Look-Back.* We also implement scan, a widely used
 858 parallel primitive, in Snap. We focus on the prefix-sum scan, which computes, for each position in an
 859 array, the sum of all elements up to that position. Prefix-sum sits in a different corner of the GPU
 860 design space from matrix multiplication: it is memory intensive, requires careful attention to the
 861 convergence behavior of different threads, and traditionally requires multiple passes over data.
 862

863 We implement the single-pass parallel prefix scan with decoupled look-back, introduced by Merrill
 864 and Garland , an elegant algorithm that does not require multiple passes over the input data. In
 865 this algorithm, each block computes a local prefix sum over its assigned region of the array. Once
 866 finished, it writes the final element of its region into a global array. Each block then *looks back* to
 867 accumulate the contributions of prior blocks, allowing them to independently determine the global
 868 prefix without a full sweep over memory. This decoupled look-back mechanism lets blocks progress
 869 at different speeds while still producing correct global results.

870 The algorithm involves several distinct points of convergence. Within each block, work is de-
 871 composed into fine-grained thread-level and warp-level scans. After producing the local result,
 872 blocks publish their partial prefix to global memory. Finally, each block waits until enough prefix
 873 information from earlier blocks becomes available, at which point it accumulates the value and
 874 completes its section of the global scan.

875 We implement this full strategy in Snap, available in Appendix . Our implementation uses the
 876 unsafe feature to implement a global-memory spinlock that lets blocks check when the preceeding
 877 block's data is ready.

878 6.2 RQ2: Can Snap express programs that use advanced GPU features?

879 To test whether Snap can express programs that use advanced features of modern GPUs, we write a
 880 matrix multiplication for the `bf16` datatype for the H100.
 881

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

The implementation works as follows. First, we assign each block on our machine a logical tile of the output to compute. The block is then divided into a producer warp-group and a set of consumer warp-groups, where the producer loads data for multiple consumers, and both must signal to each other when one is done loading and the other is ready to compute. In this way, the benchmark overlaps computation with data movement by pipelining loads.

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

Notably, for this benchmark, to get the `wmma` instruction to work we did not need to add a special privilege in our language; we could simply use `thread[128]`. We did, however, need to add a special TMA asynchronous data-movement construct, since Snap will eventually need to synchronize these transfers.

6.3 RQ3: Can Snap match the performance of existing, speed-of-light CUDA code?

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

The performance of the matrix multiplication benchmarks is shown in , where we demonstrate that Snap is competitive with cuBLAS [5].

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

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

6.4 Case Study: Can Snap help with library design?

Over the course of our evaluation, we found ourselves developing a small library of functions—similar in spirit to the CUB library—that we could call to execute our operations. Based on our experience, we would like to study, qualitatively, if Snap can help programmers design libraries that they can compose with confidence.

As mentioned in Section 1, CUB occupies a unique design point in the GPU library ecosystem. Unlike many other libraries such as cuBLAS [5], cuDNN [6], and cuSPARSE [7], which provide global interfaces that users can call and configure, CUB provides a device-side library organized into different levels. It makes these levels apparent by prefixing each of its functions with `Device`, `Block`, and `Thread`. The single-scan prefix sum in Section 6.1 used these functions in Snap already.

Let's turn our attention to a particular CUB function—the store function—and examine how it is equivalently expressed in Snap, and how Snap's type system reifies the assumptions implicit in CUB. In CUB, the store function is implemented as a class, as shown below:

```
932
933
934
935
936 1 template<typename T, int BlockDimX,
937   int ItemsPerThread, BlockLoadAlgorithm Algorithm = BLOCK_LOAD_DIRECT,
938   int BlockDimY = 1, int BlockDimZ = 1>
939
940
941
942
943
944
```

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

```
945
946
947
948 1 using BlockLoad = cub::BlockLoad<int, 128, 4, BLOCK_LOAD_DIRECT>;
949 2 // Allocate shared memory for BlockLoad
950 3 __shared__ typename BlockLoad::TempStorage temp_storage;
951 4 int thread_data[4]; // Thread local data
952
953
954
955
956
957
958
959
```

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

- (1) The CUB documentation needs to specify the number of threads that the function can assume to be available, because within the function, each thread must locate itself in the computation and use its `threadIdx.x` accordingly. If the starting `threadIdx.x` is not 0, the function must compute its relative ID internally.
- (2) The “item per thread” design is interesting and serves two purposes. The first is related to performance: if loops have constant bounds, they can be unrolled, enabling downstream optimizations. The second is related to correctness: the function relies on the assumption that all threads call the function with an equal number of values to load.
- (3) The CUB documentation also needs to specify that `thread_data` is local; if the same pointer to shared memory is passed, threads may overwrite each other's values.
- (4) Finally, the CUB documentation specifies that if shared memory is being overwritten, a `__syncthreads()` call must be made to ensure that all reads have completed.

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

```
960
961
962 1 @cuda("device")
963 2 @require(block[1], thread[1])
964 3 def load(src: ptr(const(shared(int)))) @ block[1],
965   item_per_thread : int @ block[1],
966   total_size: int @ block[1],
967   thread_data : ptr(int) @ thread[1]:
```

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

- (1) We do not need to pass in the number of threads at all. Whenever Snap calls a function, it threads the relative ID through, so each function can be written locally as if it were running alone, rather than having to determine where the thread resides in the global array.
- (2) We do not need to make `item_per_thread` a template argument for *correctness*. Its frequency is set at the function signature, so Snap will never allow a function to be called with a lower-privilege value.
- (3) In our interface, `thread_data` is explicitly set to a thread-local value. Since it is not marked as `const`, Snap conservatively assumes it may be written to, and enforces at compile time that only `thread[1]` values are passed in.
- (4) Finally, using our synchronization pass outlined in Section 3.9, a `__syncthreads()` call will be inserted automatically if `src` is going to be used for writing.

981 **7 Related Work**

982 *GPU Programming Languages.*

984 *Theoretic Foundations.* The design of Bundl is heavily inspired by existing work on coeffect systems
 985 [16, 17] and dependency tracking [1]. Coeffects allow type systems to talk about how programs
 986 depend on their environments, and have achieved widespread use in Rust [20] in the form of linear
 987 types. In Snap and Bundl, privileges act like coeffects by describing what compute resources are
 988 necessary for programs to execute. Dependency tracking calculi, meanwhile, allow type systems to
 989 track how data and code depend on each other, and have commonly been used to implement secure
 990 information flow analyses [9]. In Bundl, low privilege data is unable to flow into high privilege contexts,
 991 and we use dependency tracking to capture this restriction in Bundl’s type system.

992 **8 Conclusion, Limitations, and Future Work**

994 We have presented Snap, a new low-level GPU language that statically guarantees safe usage of
 995 compute resources by construction, and have demonstrated that it is possible to achieve this safety
 996 without sacrificing performance. We hope our design can inform the implementation of existing
 997 libraries like CUB that attempt to achieve the same goals without static enforcement.

998 We believe Snap is ripe for extension with additional features and capabilities to further improve its
 999 expressivity and usefulness to GPU programmers. Currently, Snap does not have support for explicit
 1000 pointers; programmers must use built-in constructs in order to interact with memory. However, prior
 1001 work like Descend [13] provides a promising template for extending Snap with more permissive
 1002 pointer usage. In particular, were we to restrict Snap’s partition construct to a specific set of
 1003 injective functions identified by Descend, we believe we could guarantee data-race freedom in Snap.

1004 Another promising area for future work is expanding the set of architectures on which Snap and
 1005 its design principles can be used. While Snap currently only supports programming the GPU, we
 1006 believe we can extend the core ideas cleanly to the boundary with the CPU and in general to other
 1007 heterogeneous hardware architectures. Indeed, we do not think that the idea of compute privileges
 1008 is useful only on the GPU, and we would like to investigate how we might adapt Snap to other
 1009 hierarchical programming systems. Conversely, we also believe it is possible for Snap to support
 1010 more GPU architectures featuring even coarser-grained tensor-core operations like Blackwell [8].

1011 We also would like to improve the ergonomics of programming with Snap, in particular by
 1012 improving the pipelining experience and providing users with more control over synchronization.
 1013 As we saw in Section 6.2, Snap requires pipeline slots to be explicitly named and all pipelines to
 1014 be statically sized. We believe we can improve upon this by adding explicit language constructs
 1015 for organizing pipelines to reduce boilerplate code in this case. Similarly, all syncs are currently
 1016 handled by Snap’s compiler, but we would like to extend the language with user-level synchronization
 1017 operations, leveraging the existing privilege system to avoid deadlocking.

1018 On the theoretical side, we are interested in further exploration of Bundl. In particular, we would
 1019 like to examine a terminating fragment of the calculus and prove the liveness property discussed in
 1020 Section 4.3: that every thread in the logical grouping denoted by privilege π will always reach the
 1021 parts of a program with that code privilege. We believe this is equivalent to showing that all threads
 1022 with the same privilege always execute the same code and observe the same data, and are optimistic
 1023 that this can be proved via logical relation, following in the footsteps of Turon et al. [22] and Spies
 1024 et al. [18, 19].

1026 **References**

- 1027 [1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A core calculus of dependency. In *Proceedings*
 1028 *of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (San Antonio Texas USA, 1999-01).

- 1030 ACM, 147–160. [doi:10.1145/292540.292555](https://doi.org/10.1145/292540.292555)
- 1031 [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](https://doi.org/10.1145/3434305)
- 1032 [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](https://doi.org/10.4230/LIPIcs.CSL.2012.107)
- 1033 [4] NVIDIA Corporation. 2025. cub::BlockReduce. https://nvidia.github.io/ccl/cub/api/classcub_1_1BlockReduce.html
- 1034 [5] NVIDIA Corporation. 2025. cuBLAS. <https://docs.nvidia.com/cuda/cublas/index.html>
- 1035 [6] NVIDIA Corporation. 2025. cuDNN. <https://docs.nvidia.com/deeplearning/cudnn/latest/>
- 1036 [7] NVIDIA Corporation. 2025. cuSPARSE. <https://docs.nvidia.com/cuda/cusparse/index.html>
- 1037 [8] NVIDIA Corporation. 2025. NVIDIA Blackwell Architecture Technical Brief. (2025). <https://resources.nvidia.com/en-us-blackwell-architecture>
- 1038 [9] 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](https://doi.org/10.1145/359636.359712)
- 1039 [10] Edsger Wybe Dijkstra. 1963. Over de sequentialiteit van procesbeschrijvingen. (1963). <https://training-ir7.tdl.org/handle/123456789/594>
- 1040 [11] 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](https://doi.org/10.1145/2933575.2935310)
- 1041 [12] 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](https://doi.org/10.1145/3062341.3062354)
- 1042 [13] 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](https://doi.org/10.1145/3656411)
- 1043 [14] NVIDIA Corporation. 2025. CUB: CUDA Unbound. <https://docs.nvidia.com/cuda/cub/index.html> CUDA Toolkit Documentation.
- 1044 [15] NVIDIA Corporation. 2025. CUDA C++ Programming Guide. NVIDIA. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/> Accessed: 2025-11-06.
- 1045 [16] 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](https://doi.org/10.1007/978-3-642-39212-2_35)
- 1046 [17] 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](https://doi.org/10.1145/2692915.2628160)
- 1047 [18] Simon Spies, Lennard Gähler, Daniel Gratzner, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 80–95. [doi:10.1145/3453483.3454031](https://doi.org/10.1145/3453483.3454031)
- 1048 [19] Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021. Transfinite step-indexing for termination. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 1–29. [doi:10.1145/3434294](https://doi.org/10.1145/3434294)
- 1049 [20] Rust Team. 2025. Rust Programming Language. <https://rust-lang.org/>
- 1050 [21] Philippe Tillet. 2025. Introducing Triton: Open-source GPU programming for neural networks. <https://openai.com/index/triton/>
- 1051 [22] 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](https://doi.org/10.1145/2429069.2429111)
- 1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078

1079 **A Complete Bundl Type System, Semantics, and Syntactic Soundness Proofs**

1080 **A.1 Basic Definitions**

1081
 1082 Hierarchy Levels $h ::= \text{Grid} \mid \text{Block} \mid \text{Thread}$
 1083 Memory Kinds $l ::= \text{Local} \mid \text{Shared} \mid \text{Global}$
 1084 Privileges $\pi : h \times \mathbb{N}$
 1085 Base Types $b ::= \text{bool} \mid \text{int} \mid \text{float}$
 1086 Types $\tau ::= b \mid b[]^l \mid \text{Fun}(\Gamma, \pi, m) \mid \text{async } \tau$
 1087 Contexts $\Gamma ::= \cdot \mid \Gamma, x :^\pi \tau$
 1088
 1089 Shared Memory Remaining $m : \mathbb{N}$

1090
 1091
 1092 Privileges π are part of an algebra parameterized over some constant values T (the number of threads
 1093 per block) and B (the number of blocks per grid). With these values, we have two isomorphisms:
 1094

$$(1095) \quad (\text{Block}, 1) \cong (\text{Thread}, T)$$

1096 and

$$(1098) \quad (\text{Grid}, 1) \cong (\text{Block}, B)$$

1099 The group and split operations of Snap allows us to move along this isomorphism, from left
 1100 to right. For clarity, in Bundl we split these operations into three: a split operation that can split
 1101 privileges into multiple smaller ones, and a destruct operation that directly moves us along the
 1102 isomorphism, and a group that divides our current privilege into equal sized parts.

1103 Privileges π are also lexicographically ordered in the obvious way. hs are ordered such that
 1104 $\text{Thread} \leq \text{Block} \leq \text{Grid}$, and $(h_1, n_1) \leq (h_2, n_2)$ iff $n_1 \leq n_2$ and $h_1 \leq h_2$.

1105 We define scalar multiplication $i \times h$ of natural numbers with hs : $i \times (h, n) = (h, in)$.

1106 We also define division of privileges and hierarchy levels of type $\pi \times \pi \rightarrow \mathbb{N}$. $\text{Grid}/\text{Block} = B$ and
 1107 $\text{Block}/\text{Thread} = T$. We lift this to privileges like so: $(h_1, n_1)/(h_2, n_2) = ((h_1/h_2) \cdot n_1)/n_2$.

1108 Lastly we define a partial \downarrow operator on hs such that $\downarrow \text{Block} = \text{Thread}$ and $\downarrow \text{Grid} = \text{Block}$. Note
 1109 that $\downarrow \text{Thread}$ is undefined. This operator lifts to π s whose second component is 1 and encodes the
 1110 leftward component of the isomorphism above: $\downarrow (\text{Block}, 1) = (\text{Thread}, T)$ and $\downarrow (\text{Grid}, 1) = (\text{Block}, B)$.

1111 Note also that the presentation of these rules in the main body of the paper elide the m portion,
 1112 which tracks the maximum amount of memory a given computation is allowed to use. In the full
 1113 system presented here, both the typing rules and the operational semantics carry an additional piece
 1114 of information tracking allocated memory.

1116 **A.2 Typing Rules**

1117 **A.2.1 Expressions.**

$$1118 \quad \frac{x :^\pi \tau \in \Gamma}{\Gamma \vdash^\pi x : \tau} \text{-Var} \quad \frac{}{\Gamma \vdash^\pi n : \text{int}} \text{-Int} \quad \frac{}{\Gamma \vdash^\pi f : \text{float}} \text{-Float}$$

$$1121 \quad \frac{}{\Gamma \vdash^\pi b : \text{bool}} \text{-Bool} \quad \frac{\pi < (\text{Grid}, 1)}{\Gamma \vdash^\pi \text{partition_id} : \text{int}} \text{-Partition-Id}$$

$$1124 \quad \frac{\Gamma \vdash^{\pi'} e_1 : \tau[]^l \quad \Gamma \vdash^\pi e_2 : \text{int} \quad l = \text{Global} \text{ or } l = \text{Local} \quad \pi \leq \pi'}{\Gamma \vdash^\pi e_1[e_2] : \tau} \text{-Arr-Access}$$

1128	$\frac{\Gamma \vdash^{\pi'} e_1 : \tau[]^{\text{Shared}} \quad \Gamma \vdash^{\pi} e_2 : \text{int} \quad \pi \leq (\text{Block}, 1) \quad \pi \leq \pi'}{\Gamma \vdash^{\pi} e_1[e_2] : \tau}$	T-Arr-Access-Shared
1129		
1130		
1131	$\frac{\Gamma \vdash^{\pi} e_1 : \text{int} \quad \Gamma \vdash^{\pi} e_2 : \text{int}}{\Gamma \vdash^{\pi} e_1 \text{ bop } e_2 : \text{int}}$	T-Bop
1132	$\frac{\Gamma \vdash^{\pi} e_1 : \text{int} \quad \Gamma \vdash^{\pi} e_2 : \text{int}}{\Gamma \vdash^{\pi} e_1 \text{ cmp } e_2 : \text{bool}}$	T-Cmp
1133		
A.2.2 Statements.		
1134		
1135	$\frac{f : \pi' \mathbf{Fun}(x_i : \tau_i, \pi, m') \in \Gamma \quad \Gamma \vdash^{\pi} e_i : \tau_i \quad m' \leq m}{\Gamma \vdash_m^{\pi} f(e_1, \dots, e_n)}$	T-Function-Call
1136		
1137		
1138	$\frac{\Gamma \vdash_m^{(h,n_1)} s_1 \quad \Gamma \vdash_m^{(h,n_2)} s_2 \quad n_1, n_2 \text{ align to } n}{\Gamma \vdash_m^{(h,n)} \text{split}(n_1, n_2)\{s_1\}\{s_2\}}$	T-Split
1139		
1140	$\frac{\Gamma \vdash_m^{\downarrow\pi} s}{\Gamma \vdash_m^{\pi} \text{destruct in } s}$	T-Destruct
1141		
1142	$\frac{\Gamma \vdash_m^{(h,n)} s}{\Gamma \vdash_m^{(h,q \cdot n)} \text{group } q \ s}$	T-Group
1143	$\frac{x : \pi' \tau \in \Gamma \quad \Gamma \vdash^{\pi} e : \tau \quad \pi' \leq \pi \quad \pi' \pi}{\Gamma \vdash_m^{\pi} x = e}$	T-Assn
1144		
1145	$\frac{\Gamma \vdash_m^{\pi} \text{init}_{\psi}}{\Gamma \vdash_m^{\pi} \text{Sync-Init}}$	T-Sync-Init
1146	$\frac{\Gamma \vdash_m^{\pi} \text{dec}_{\psi}}{\Gamma \vdash_m^{\pi} \text{Sync-Dec}}$	T-Sync-Dec
1147		
1148	$\frac{\Gamma \vdash_m^{\pi} \text{wait}_{\psi}}{\Gamma \vdash_m^{\pi} \text{Sync-Wait}}$	T-Sync-Wait
1149	$\frac{\Gamma \vdash_m^{\pi} \text{skip}}{\Gamma \vdash_m^{\pi} \text{Skip}}$	T-Skip
1150	$\frac{n \leq m}{\Gamma \vdash_m^{\pi} \text{free } n}$	T-Free
1151		
1152	$\frac{\Gamma \vdash^{\pi'} e_1 : \tau[]^l \quad \Gamma \vdash^{\pi} e_2 : \text{int} \quad \Gamma \vdash^{\pi} e_3 : \tau \quad l = \text{Global or } l = \text{Local} \quad \pi' \leq \pi \quad \pi' \pi}{\Gamma \vdash_m^{\pi} e_1[e_2] = e_3}$	T-Arr-Assn
1153		
1154		
1155	$\frac{\Gamma \vdash^{\pi'} e_1 : \tau[]^{\text{Shared}} \quad \Gamma \vdash^{\pi} e_2 : \text{int} \quad \Gamma \vdash^{\pi} e_3 : \tau \quad \pi \leq (\text{Block}, 1) \quad \pi' \leq \pi \quad \pi' \pi}{\Gamma \vdash_m^{\pi} e_1[e_2] = e_3}$	T-Arr-Assn-Shared
1156		
1157		
1158	$\frac{\Gamma \vdash^{\pi} e : \text{bool} \quad \Gamma \vdash_m^{\pi} s_1 \quad \Gamma \vdash_m^{\pi} s_2}{\Gamma \vdash_{\max(m_1, m_2)}^{\pi} \text{if } e \text{ then } s_1 \text{ else } s_2}$	T-If
1159		
1160		
1161	$\frac{\Gamma \vdash^{\pi} e : \text{bool} \quad \Gamma \vdash_m^{\pi} s}{\Gamma \vdash_m^{\pi} \text{while } e \text{ do } s}$	T-While
1162	$\frac{\Gamma \vdash_{\max(m_1, m_2)}^{\pi} s_1; s_2}{\Gamma \vdash_m^{\pi} s}$	T-Seq
1163		
1164	$\frac{\Gamma, x : \pi' \tau[]^l \vdash_m^{\pi} s \quad l = \text{Global or } l = \text{Local}}{\Gamma \vdash_{m+n \cdot \text{size}(\tau)}^{\pi} x := \text{alloc } l \ \tau \ n \text{ in } s}$	T-Alloc
1165		
1166		
1167	$\frac{\Gamma, x : (\text{Block}, 1) \tau[]^{\text{Shared}} \vdash_m^{\pi} s}{\Gamma \vdash_{m+n \cdot \text{size}(\tau)}^{\pi} x := \text{alloc Shared } \tau \ n \text{ in } s}$	T-Alloc-Shared
1168		
1169		
1170	$\frac{\Gamma, y : (h, n/c) \tau[]^l \vdash_m^{(h, n)} s \quad c n \quad l \neq \text{Local}}{\Gamma, x : (h, n) \tau[]^l \vdash_m^{(h, n)} \text{partition}_{\psi} x \text{ into } y \text{ by } c \text{ in } s}$	T-Partition
1171		
1172		
1173		
1174	$\frac{\Gamma, y : (h, n') \tau[]^l \vdash_m^{(h, n')} s \quad n' \leq n \quad l \neq \text{Local}}{\Gamma, x : (h, n) \tau[]^l \vdash_m^{(h, n)} \text{claim}_{\psi} x \text{ into } y \text{ at } n' \text{ in } s}$	T-Claim
1175		
1176		

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

$$\frac{\Gamma, y : \tau[]^l \vdash_m^{(\text{Thread},1)} s}{\Gamma, x : \tau[]^l \vdash_m^{(\text{Thread},1)} \text{async_partition}_\phi x \text{ into } y \text{ in } s} \text{-T-Async-Partition}$$

$$\frac{\Gamma, x:(\text{Thread},1) \text{ async } \tau[]^l, y:(\text{Thread},1) \tau[]^{l'} \vdash_m^{(\text{Thread},1)} \text{async_memcpy}(x,y)}{\Gamma, x:\pi \tau[]^l, y:\pi \tau[]^{l'} \vdash_m^\pi \text{memcpy}(x,y)} \text{ T-Memcpy}$$

A.3 Complete Bundl Semantics

A.3.1 Definitions.

Global Memory $\Sigma ::= \cdot | \Sigma, n \mapsto^\pi v$

Shared Memory $\sigma ::= \cdot | \sigma, n \mapsto^\pi v$

Local Memory $\eta ::= \cdot | \eta, n \mapsto^\pi v$

Block Memory Map $S ::= \forall n \in B, n \mapsto \sigma$

Thread Memory Map $L ::= \forall n \in T, n \mapsto \eta$

Synchronization Map $\Psi: \psi \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

Deferred Computations Map $\Phi: \phi \rightarrow \{s\}$

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, $t_{\text{real}} = t_{\text{simplified}} \bmod T$ 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 ψ s and $\lambda .\{\}$ for ϕ s not explicitly initialize.

The shape of the judgment for a single thread is $\eta, \sigma, \Sigma, t, b, p, \Psi \vdash_m^\pi s \rightsquigarrow s' \dashv_m \eta', \sigma', \Sigma', \Psi'$. 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 π s 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: $(h,n)/c = (h,n)/(h,c)$.

1226 The overall evaluation of a program is expressed as

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

1229 In this judgment P serves as a *thread pool*, mapping pairs of thread and block ids (which don't change)
 1230 to statements and memory (which can be updated by stepping). One can think of P as tracking which
 1231 program is running on each thread. This steps according to the following rule:

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

1235 For simplicity of notation, we define an **update** operation that searches the three environments
 1236 for the one that contains the variable being used (by convention, there is no conflict between the
 1237 environments, as in reality they exist in three separate address spaces). We also define a similar **get**
 1238 operation that retrieves a variable from memory, and a **rename** operation that remaps a variable
 1239 with the same value but under a different name.

$$1241 \quad \text{update}(\eta, \sigma, \Sigma, x, v) = (\eta[x \mapsto^\pi v], \sigma, \Sigma) \text{ when } x \in^\pi \eta$$

$$1242 \quad \text{update}(\eta, \sigma, \Sigma, x, v) = (\eta, \sigma[x \mapsto^\pi v], \Sigma) \text{ when } x \in^\pi \sigma$$

$$1243 \quad \text{update}(\eta, \sigma, \Sigma, x, v) = (\eta, \sigma, \Sigma[x \mapsto^\pi v]) \text{ when } x \in^\pi \Sigma$$

$$1244 \quad \text{get}(\eta, \sigma, \Sigma, x) = \eta(x) \text{ when } x \in^\pi \eta$$

$$1245 \quad \text{get}(\eta, \sigma, \Sigma, x) = \sigma(x) \text{ when } x \in^\pi \sigma$$

$$1246 \quad \text{get}(\eta, \sigma, \Sigma, x) = \Sigma(x) \text{ when } x \in^\pi \Sigma$$

$$1247 \quad \text{rename}(\eta, \sigma, \Sigma, x, y, \pi') = (\eta[y \mapsto^\pi \eta(x)], \sigma, \Sigma) \text{ when } x \in^\pi \eta$$

$$1248 \quad \text{rename}(\eta, \sigma, \Sigma, x, y, \pi') = (\eta, \sigma[y \mapsto^\pi \sigma(x)], \Sigma) \text{ when } x \in^\pi \sigma$$

$$1249 \quad \text{rename}(\eta, \sigma, \Sigma, x, y, \pi') = (\eta, \sigma, \Sigma[y \mapsto^\pi \Sigma(x)]) \text{ when } x \in^\pi \Sigma$$

1256 A.3.2 Privilege Management Rules.

$$1258 \quad p < n_1 \quad n_1, n_2 \text{ align to } n \quad \eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(h, n_1)} s_1 \rightsquigarrow s'_1 \dashv_{m'} \eta', \sigma', \Sigma', \Psi', \Phi' \\ 1259 \quad \eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(h, n)} \text{split}(n_1, n_2)\{s_1\}\{s_2\} \rightsquigarrow \text{split}(n_1, n_2)\{s'_1\}\{s_2\} \dashv_{m'} \eta', \sigma', \Sigma', \Psi', \Phi' \quad \text{S-Split-Left}$$

$$1261 \quad p < n_1 \quad n_1, n_2 \text{ align to } n \quad \eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(h, n)} \text{split}(n_1, n_2)\{\text{skip}\}\{s_2\} \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi \quad \text{S-Split-Left-Done}$$

$$1265 \quad p \geq n_1 \quad p < n_1 + n_2 \quad n_1, n_2 \text{ align to } n \quad \eta, \sigma, \Sigma, t, b, p - n_1, \Psi, \Phi \vdash_m^{(h, n_2)} s_2 \rightsquigarrow s'_2 \dashv_{m'} \eta', \sigma', \Sigma', \Psi', \Phi' \\ 1266 \quad \eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(h, n)} \text{split}(n_1, n_2)\{s_1\}\{s_2\} \rightsquigarrow \text{split}(n_1, n_2)\{s_1\}\{s'_2\} \dashv_{m'} \eta', \sigma', \Sigma', \Psi', \Phi' \quad \text{S-Split-Right}$$

$$1268 \quad p \geq n_1 \quad p < n_1 + n_2 \quad n_1, n_2 \text{ align to } n \quad \eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(h, n)} \text{split}(n_1, n_2)\{s_1\}\{\text{skip}\} \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi \quad \text{S-Split-Right-Done}$$

$$1272 \quad p \geq n_1 + n_2 \quad n_1, n_2 \text{ align to } n \quad \eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(h, n)} \text{split}(n_1, n_2)\{s_1\}\{s_2\} \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi \quad \text{S-Split-None}$$

1275			
1276	$\eta, \sigma, \Sigma, t, b, t \text{ mod } T, \Psi, \Phi \vdash_m^{(\text{Thread}, T)} s \rightsquigarrow s' \dashv_m \eta', \sigma', \Sigma', \Psi', \Phi'$		
1277		$\eta, \sigma, \Sigma, t, b, 0, \Psi, \Phi \vdash_m^{(\text{Block}, 1)} \text{destruct in } s \rightsquigarrow \text{destruct in } s' \dashv_m \eta', \sigma', \Sigma', \Psi', \Phi'$	S-Destruct-Block
1278			
1279			
1280	$\eta, \sigma, \Sigma, t, b, b \text{ mod } B, \Psi, \Phi \vdash_m^{(\text{Block}, B)} s \rightsquigarrow s' \dashv_m \eta', \sigma', \Sigma', \Psi', \Phi'$		
1281		$\eta, \sigma, \Sigma, t, b, 0, \Psi, \Phi \vdash_m^{(\text{Grid}, 1)} \text{destruct in } s \rightsquigarrow \text{destruct in } s' \dashv_m \eta', \sigma', \Sigma', \Psi', \Phi'$	S-Destruct-Grid
1282			
1283			
1284	$\eta, \sigma, \Sigma, t, b, 0, \Psi, \Phi \vdash_m^\pi \text{destruct in skip} \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi$		S-Destruct-Done
1285			
1286	$\eta, \sigma, \Sigma, t, b, p \text{ mod } n, \Psi, \Phi \vdash_m^{(h, n)} s \rightsquigarrow s' \dashv_m \eta', \sigma', \Sigma', \Psi', \Phi'$		
1287		$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(h, q \cdot n)} \text{group } q \ s \rightsquigarrow \text{group } q \ s'; \dashv_m \eta', \sigma', \Sigma', \Psi', \Phi'$	S-Group
1288			
1289	$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{group } q \ \text{skip} \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi$		S-Group-Done
1290			

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:

1293			
1294	$\text{size(Thread}, n) = n$		
1295		$\text{size(Block}, n) = n \cdot T$	
1296			$\text{size(Grid}, n) = n \cdot B \cdot T$
1297			
1298			
1299			
1300	$\Psi(\psi)(p) = 0$		
1301	$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{wait}_\psi \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi$		S-Sync-Wait-Done
1302			
1303	$\Psi(\psi)(p) \neq 0$		
1304	$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{wait}_\psi \rightsquigarrow \text{wait}_\psi \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi$		S-Sync-Wait-Spin
1305			
1306	$\Psi' = \Psi(\psi)[p \mapsto \Psi(\psi)(p) - 1]$		
1307	$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{dec}_\psi \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi', \Phi$		S-Sync-Dec
1308			
1309	$\Psi(\psi)(p) = 0 \quad \Psi' = \Psi(\psi)[p \mapsto \text{size}(\pi)]$		
1310	$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{init}_\psi \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi', \Phi$		S-Sync-Init-Zero
1311			
1312	$\Psi(\psi)(p) \neq 0$		
1313	$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{init}_\psi \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi$		S-Sync-Init-Nonzero
1314			
1315	A.3.4 Asynchrony.		
1316	$\text{rename}(\eta, \sigma, \Sigma, x, y, (\text{Thread}, 1)), t, b, p, \Psi, \Phi \vdash_m^{(\text{Thread}, 1)} s \rightsquigarrow s' \dashv_m \eta', \sigma', \Sigma', \Psi', \Phi'$		S-Async-Partition-Congr
1317		$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(\text{Thread}, 1)} \text{async_partition}_\phi x \text{ into } y \text{ in } s \rightsquigarrow$	
1318		$\text{async_partition}_\phi x \text{ into } y \text{ in } s' \dashv_m \eta', \sigma', \Sigma', \Psi', \Phi'$	
1319			
1320	$\Phi = \Phi'[\phi \mapsto \Phi'(\phi) \cup \{s\}]$		
1321	$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(\text{Thread}, 1)} \text{async_partition}_\phi x \text{ into } y \text{ in skip} \rightsquigarrow$		S-Async-Partition-Unwind
1322		$(\text{async_partition}_\phi x \text{ into } y \text{ in } s) \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi'$	
1323			

1324		
1325	$\Phi(\phi) = \emptyset$	
1326	$\frac{}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(\text{Thread}, 1)} \text{async_partition}_\phi x \text{ into } y \text{ in } \text{skip} \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi}$	S-Async-Partition-Done
1327		
1328		
1329		
1330	$\frac{}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(\text{Thread}, 1)} \text{async_memcpy}(x, y) \rightsquigarrow \text{skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi[\phi \mapsto \Phi(\phi) \cup \{\text{memcpy}(x, y)\}]}$	S-Async-Memcpy
1331		
1332		
1333	$\frac{(\eta', \sigma', \Sigma') = \text{update}(\eta, \sigma, \Sigma, x, \text{get}(\eta, \sigma, \Sigma, y))}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{memcpy}(x, y) \rightsquigarrow \text{skip} \dashv_m \eta', \sigma', \Sigma', \Psi, \Phi}$	S-Memcpy
1334		
1335		
1336	A.3.5 <i>Variables and Memory.</i>	
1337	$\frac{\eta, \sigma, \Sigma \vdash_{\pi'}^\pi e \Downarrow v}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi x : \tau @ \pi' := e \text{ in } s \rightsquigarrow s \dashv_m \eta[x \mapsto^{\pi'} v], \sigma, \Sigma, \Psi, \Phi}$	S-Decl
1338		
1339		
1340	$\frac{}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{free } n \dashv_{m-n} \eta, \sigma, \Sigma, \Psi, \Phi}$	S-Free
1341		
1342		
1343	$\frac{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi x := \text{alloc Local } \tau n \text{ in } s \rightsquigarrow s; \text{ free } (n \cdot \text{size}(\tau)) \dashv_{m+n \cdot \text{size}(\tau)} \eta[x \mapsto^\pi \langle x, n \rangle], \sigma, \Sigma, \Psi, \Phi}{}$	S-Alloc-Local
1344		
1345		
1346	$\frac{\pi = (\text{Block}, 1)}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi x := \text{alloc Shared } \tau n \text{ in } s \rightsquigarrow s; \text{ free } (n \cdot \text{size}(\tau)) \dashv_{m+n \cdot \text{size}(\tau)} \eta[x \mapsto^\pi \langle x, n \rangle], \sigma, \Sigma, \Psi, \Phi}$	S-Alloc-Shared
1347		
1348		
1349		
1350	$\frac{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi x := \text{alloc Global } \tau n \text{ in } s \rightsquigarrow s; \text{ free } (n \cdot \text{size}(\tau)) \dashv_{m+n \cdot \text{size}(\tau)} \eta, \sigma, \Sigma[x \mapsto^\pi \langle x, n \rangle], \Psi, \Phi}{}$	S-Alloc-Global
1351		
1352		
1353	$\frac{x \in^{\pi'} \eta, \sigma, \Sigma \quad \eta, \sigma, \Sigma \vdash_{\pi'}^\pi e \Downarrow v \quad (\eta', \sigma', \Sigma') = \text{update}(\eta, \sigma, \Sigma, x, v) \quad \pi' \mid \pi}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi, \Phi \vdash_m^\pi x = e \rightsquigarrow \text{skip} \dashv_m \eta', \sigma', \Sigma', \Psi, \Phi}$	S-Assn
1354		
1355		
1356		
1357	$\frac{\eta, \sigma, \Sigma, t, b, p \vdash_\pi^\pi e_1 \Downarrow \langle l, n \rangle \qquad i < n \quad \pi' \mid \pi \\ \eta, \sigma, \Sigma, t, b, p \vdash_\pi^\pi e_2 \Downarrow i \qquad x \in^{\pi'} \eta, \sigma, \Sigma \\ \eta, \sigma, \Sigma \vdash_{\pi'}^\pi e_3 \Downarrow v \qquad (\eta', \sigma', \Sigma') = \text{update}(\eta, \sigma, \Sigma, l+i, v)}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi e_1[e_2] = e_3 \rightsquigarrow \text{skip} \dashv_m \eta', \sigma', \Sigma', \Psi, \Phi}$	S-Arr-Assn
1358		
1359		
1360		
1361		
1362	$\frac{s' = s[(y + c \cdot p)/y] \quad (\eta', \sigma', \Sigma') = \text{rename}(\eta, \sigma, \Sigma, x, y, \pi/c)}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{partition}_\psi x y c s \rightsquigarrow \text{init}_\psi; s'; \text{dec}_\psi; \text{wait}_\psi \dashv_m \eta', \sigma', \Sigma', \Psi, \Phi}$	S-Partition
1363		
1364		
1365	$\frac{(\eta', \sigma', \Sigma') = \text{rename}(\eta, \sigma, \Sigma, x, y, (h, n_1))}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{(h, n_1+n_2)} \text{claim}_\psi x \text{ into } y \text{ at } n_1 \text{ in } s \rightsquigarrow \text{init}_\psi; \text{split}(n_1, n_2) \{s'\} \{\text{skip}\}; \text{dec}_\psi; \text{wait}_\psi \dashv_m \eta', \sigma', \Sigma', \Psi, \Phi}$	S-Claim
1366		
1367		
1368		
1369		
1370	$\frac{(\eta', \sigma', \Sigma') = \text{rename}(\eta, \sigma, \Sigma, x, y, \downarrow \pi)}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi \text{lower}_\psi x \text{ into } y \text{ in } s \rightsquigarrow \text{init}_\psi; s; \text{dec}_\psi; \text{wait}_\psi \dashv_m \eta', \sigma', \Sigma', \Psi, \Phi}$	S-Lower
1371		
1372		

1373 A.3.6 Control Flow.

$$\begin{array}{c}
 \frac{\eta, \sigma, \Sigma \vdash_{\pi}^{\pi} e \Downarrow \text{true}}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{\pi} \text{if } e \text{ then } s_1 \text{ else } s_2 \rightsquigarrow s_1 \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi} \text{ S-If-True} \\
 \\
 \frac{\eta, \sigma, \Sigma \vdash_{\pi}^{\pi} e \Downarrow \text{false}}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{\pi} \text{if } e \text{ then } s_1 \text{ else } s_2 \rightsquigarrow s_2 \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi} \text{ S-If-False} \\
 \\
 \frac{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{\pi} \text{while } e \text{ do } s \rightsquigarrow \text{if } e \text{ then } (s; \text{while } e \text{ do } s) \text{ else skip} \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{\pi} s_1 \rightsquigarrow s'_1 \dashv_{m'}^{\pi'} \eta', \sigma', \Sigma', \Psi', \Phi'} \text{ S-While} \\
 \\
 \frac{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{\pi} s_1 \rightsquigarrow s'_1; s_2 \rightsquigarrow s'_2 \dashv_{m'}^{\pi'} \eta', \sigma', \Sigma', \Psi', \Phi'}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{\pi} \text{skip}; s_2 \rightsquigarrow s_2 \dashv_m \eta, \sigma, \Sigma, \Psi, \Phi} \text{ S-Seq-Done} \\
 \\
 \frac{\Sigma(f) = \{[x_1 : \tau_1, \dots, x_n : \tau_n], s\} \quad \sigma, \Sigma \vdash_{\pi}^{\pi} e_i \Downarrow v_i \quad m' \leq m}{\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^{\pi} f(e_1, \dots, e_n) \rightsquigarrow s \dashv_m \eta[x_i \mapsto v_i], \sigma, \Sigma, \Psi, \Phi} \text{ S-Function-Call}
 \end{array}$$

1387 A.3.7 Expressions.

$$\begin{array}{c}
 \frac{\pi < (\text{Grid}, 1)}{\eta, \sigma, \Sigma \vdash_{\pi}^{\pi}, \text{partition_id} \Downarrow \pi / \pi' - 1} \text{ E-Partition-Id} \\
 \\
 \frac{}{\eta, \sigma, \Sigma \vdash_{\pi}^{\pi}, x \Downarrow \text{get}(\eta, \sigma, \Sigma, x)} \text{ E-Var} \\
 \\
 \frac{\eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, e_1 \Downarrow \langle l, n \rangle \quad \eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, e_2 \Downarrow i \quad i < n \quad \pi' \leq \pi}{\eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, e_1[e_2] \Downarrow \text{get}(\eta, \sigma, \Sigma, l + i)} \text{ E-Arr-Access} \\
 \\
 \frac{\eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, \vdash n \Downarrow n}{\eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, \vdash n \Downarrow n} \text{ E-Int} \quad \frac{}{\eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, \vdash b \Downarrow b} \text{ E-Bool} \\
 \\
 \frac{\eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, \vdash v_1 \quad \eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, \vdash v_2 \quad v = v_1 \text{ bop } v_2}{\eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, \vdash e_1 \text{ bop } e_2 \Downarrow v} \text{ E-Bop} \\
 \\
 \frac{\eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, \vdash v_1 \quad \eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, \vdash v_2 \quad v = v_1 \text{ cmp } v_2}{\eta, \sigma, \Sigma \vdash_{\pi'}^{\pi}, \vdash e_1 \text{ cmp } e_2 \Downarrow v} \text{ E-Cmp}
 \end{array}$$

1407 A.4 Theorems and Proofs

1408 Note that in this section we assume no out of bounds array accesses. In general Snap (and by extension
 1409 Bundl) makes no guarantees about array out of bounds.
 1410

1411 A.4.1 More definitions. As a premise to our type safety theorems, we need to assume we have a
 1412 well-typed environment, written $\Gamma \vdash \eta, \sigma, \Sigma$. We define what this means inductively
 1413

$$\begin{array}{c}
 \frac{}{\eta, \sigma, \Sigma \vdash n : \text{int}} \text{ V-Int} \quad \frac{}{\eta, \sigma, \Sigma \vdash b : \text{bool}} \text{ V-Bool} \quad \frac{}{\eta, \sigma, \Sigma \vdash f : \text{float}} \text{ V-Float} \\
 \\
 \frac{\forall i < n, \eta, \sigma, \Sigma \vdash \text{get}(\eta, \sigma, \Sigma, x + i) : \tau}{\eta, \sigma, \Sigma \vdash \langle x, n \rangle : \tau[]^l} \text{ V-Array} \quad \frac{\Gamma, x_i : \pi \tau_i \vdash_m^{\pi} s \quad \Gamma \vdash \cdot, \cdot, \text{fns } \Sigma}{\eta, \sigma, \Sigma \vdash \{x_i : \tau_i, s\} : \text{Fun}(x_i : \tau_i, \pi, m)} \text{ V-Function} \\
 \\
 \frac{\cdot \vdash \eta, \sigma, \Sigma}{\cdot \vdash \eta, \sigma, \Sigma} \text{ G-Empty} \quad \frac{\eta(x) =^{\pi} v \quad \eta, \sigma, \Sigma \vdash v : \text{int}}{\Gamma, x : \pi \text{ int} \vdash \eta, \sigma, \Sigma} \text{ G-Int}
 \end{array}$$

$$\begin{array}{c}
 \frac{\eta(x) =^\pi v \quad \eta, \sigma, \Sigma \vdash v : \text{bool}}{\Gamma, x :^\pi \text{bool} \vdash \eta, \sigma, \Sigma} \text{ G-Bool} \quad \frac{\eta(x) =^\pi v \quad \eta, \sigma, \Sigma \vdash v : \text{float}}{\Gamma, x :^\pi \text{float} \vdash \eta, \sigma, \Sigma} \text{ G-Float} \\
 \\
 \frac{\eta(x) =^\pi v \quad \eta, \sigma, \Sigma \vdash v : \tau[]}{\Gamma, x :^\pi \tau[]^{\text{Local}} \vdash \eta, \sigma, \Sigma} \text{ G-Local} \quad \frac{\sigma(x) =^\pi v \quad \eta, \sigma, \Sigma \vdash v : \tau[]}{\Gamma, x :^\pi \tau[]^{\text{Shared}} \vdash \eta, \sigma, \Sigma} \text{ G-Shared} \\
 \\
 \frac{\Sigma(x) =^\pi v \quad \eta, \sigma, \Sigma \vdash v : \tau[]}{\Gamma, x :^\pi \tau[]^{\text{Global}} \vdash \eta, \sigma, \Sigma} \text{ G-Global} \quad \frac{\Sigma(x) =^\pi v \quad \eta, \sigma, \Sigma \vdash v : \mathbf{Fun}(\Gamma', \pi, m)}{\Gamma, x :^\pi \mathbf{Fun}(\Gamma', \pi, m) \vdash \eta, \sigma, \Sigma} \text{ G-Function}
 \end{array}$$

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 $\eta, \sigma, \Sigma \vdash \mathbf{get}(\eta, \sigma, \Sigma, x) : \tau$.

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

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

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:

$$\Gamma \vdash \Phi ::= \forall \phi, s \in \Phi(\phi), \Gamma \vdash_m^{(\text{Thread}, 1)} s$$

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. \square

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. \square

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. \square

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

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 (Block, 1) or (Grid, 1), so we only need to consider the cases where π is one of those.

In the former case p becomes $t \text{ mod } T$ while $\downarrow \pi$ is (Thread, T). $t \text{ mod } T < T$ 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 $b \text{ mod } B < B$ 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 $p \text{ mod } n$. 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 $\Gamma, y : (\text{Thread}, 1) \text{ async } \tau[]^l \vdash \eta', \sigma', \Sigma'$, then we can either step s with $(\text{Thread}, 1)$ privilege under environments η' , σ' , and Σ' , or s is skip.

1520 We have by assumption that $\Gamma, x:(\text{Thread}, 1) \text{async } \tau[]^l \vdash \eta, \sigma, \Sigma$. By our environment renaming
 1521 lemma, this gives us what we need to use our IH with $(\eta', \sigma', \Sigma')$ as **rename** $(\eta, \sigma, \Sigma, x, y, (\text{Thread}, 1))$.
 1522 Thus s either steps or is skip. In the former case we can step with **S-Async-Partition-Congr**,
 1523 and in the latter we can use either **S-Async-Partition-Unwind** or **S-Async-Partition-Done**
 1524 depending on whether $\Phi(\phi)$ is empty or not.

- Case **T-Async-Memcpy**

1526 Immediate via use of the **S-Async-Memcpy** rule.

- Case **T-Memcpy**

1528 Immediate via use of the **S-Memcpy** rule.

1529 *Memory Rules.*

- Case **T-Decl**

1532 Via our lemma about expression type safety and our hypothesis that e is well-typed, we obtain
 1533 the premises necessary to use the **S-Decl** rule to step.

- Case **T-Arr-Assn**

1535 Each of e_1 , e_2 and e_3 must evaluate to a well-typed value by the expression type safety lemma.
 1536 In particular, both e_1 evaluates to some $\langle l, n \rangle$ and e_2 evaluates to some i . We assume all array
 1537 accesses are in bounds, so this is sufficient to use the **S-Arr-Assn** rule to step.

- Case **T-Arr-Assn-Shared**

1539 Same as previous case.

- Case **T-Free**

1541 Trivial via the **S-Free** rule.

- Case **T-Partition**

1543 Trivial via the **S-Partition** rule.

- Case **T-Claim**

1545 Trivial via the **S-Claim** rule.

- Case **T-Lower**

1547 Trivial via the **S-Lower** rule.

- Case **T-Alloc**

1549 We assume that l is not Shared, so we can use the **S-Alloc-Local** or **S-Alloc-Global** rule,
 1550 depending on whether l is Local or Global.

- Case **T-Alloc-Shared**

1552 Trivial via the **S-Alloc-Shared** rule.

1553 *Control Rules.*

- Case **T-Skip**

1556 Trivial

- Case **T-While**

1558 Trivial, all while loops step via the **S-While** rule

- Case **T-If**

1560 By our proof of expression type safety, the expression e steps to either the boolean value true
 1561 or false. We can thus use either the **S-If-True** or **S-If-False** rules to step.

- Case **T-Seq**

1563 In this case we know by our IH that s_1 is either skip or can step. In the former case we use
 1564 the **S-Seq-Done** rule and in the latter we use the **S-Seq-First** rule.

- Case **T-Function-Call**

1566 In this case we know by our expression safety lemma that each of the arguments will evaluate to
 1567 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 $\Gamma \vdash_m^\pi s$ and $\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_{m'}^\pi s \rightsquigarrow s' \dashv_{m''} \eta', \sigma', \Sigma, \Psi', \Phi'$ and $\pi \vdash p$ and $m \geq m'$ and $\Gamma \vdash \Phi$, then there is some Γ' such that $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash_m^\pi s'$ and $\Gamma' \vdash \eta', \sigma', \Sigma'$ and $m \geq m''$ and $\Gamma' \vdash \Phi'$.*

PROOF. We proceed by induction on the derivation of $\Gamma \vdash_m^\pi s$.

Privilege Management Rules.

- Case **T-Split**

In this case we have by assumption that $(h, n) \vdash p$, $\Gamma \vdash \eta, \sigma, \Sigma$, and $\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_{m'}^\pi \text{split}(n_1, n_2)\{s_1\}\{s_2\} \rightsquigarrow s' \dashv_{m''} \eta', \sigma', \Sigma, \Psi', \Phi'$. 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 $\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_{m'}^\pi \text{destruct}$ 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 $t \bmod T$. $(\text{Thread}, T) \vdash t \bmod T$ 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

$\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_{m'}^{(h, q \cdot n)} \text{group } q \ s \rightsquigarrow s' \dashv_{m''} \eta', \sigma', \Sigma, \Psi', \Phi'$. 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 $p \bmod n$. It is always the case that $(h, n) \vdash p \bmod n$ 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 $\text{group } q \ s'$ is well-typed, and to finish the case.

1618 **Thread Synchronization Rules.** These rules are all trivial: with one exception all thread synchronization
 1619 primitives step to skip without changing environment or memory, and are thus obviously well-typed.
 1620 **S-Sync-Wait-Spin** does not produce skip, but it steps to the same statement as we already assumed
 1621 typechecks in the premise of the lemma, so is straightforward nonetheless.

1622 If we wanted to say something about deadlock freedom we'd have more work here, but we aren't
 1623 doing that, so these rules are easy.

1624 **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 $(\text{Thread},1)$ privilege. We also have that $\Gamma, x:(^{\text{Thread},1}) \tau[]^l \vdash \eta, \sigma, \Sigma$ and $\Gamma, x:(^{\text{Thread},1}) \tau[]^l \Phi$.

1629 By inversion, we are in one of three cases.

1630 In the **S-Async-Partition-Done** case, we are done.

1631 In the **S-Async-Partition-Congr** case, our inductive hypothesis gives us that there is some
 1632 Γ' such that $\Gamma, y:(^{\text{Thread},1}) \text{async } \tau[]^l \subseteq \Gamma'$ and $\Gamma' \vdash \Phi'$ and $\Gamma' \vdash \text{rename}(\eta, \sigma, \Sigma, x, y, (\text{Thread},1))$
 1633 via our well-typed renaming lemma. This lets us use the **T-Async-Partition** rule to check
 1634 this case, with a choice of Γ' as $\Gamma, x:(^{\text{Thread},1}) \tau[]^l$.

1635 In the **S-Async-Partition-Unwind** case, our assumption that Φ is well-typed tells us that
 1636 $\Gamma, y:(^{\text{Thread},1}) \vdash \{ (\text{Thread},1)_m s$. Thus, we can use the **T-Async-Partition** rule to type this case.

- Case **T-Async-Memcpy**

1638 In this case the statement and environment typing are trivial, we need only to show that the
 1639 asycn stack remains well typed.

1640 In this case we have that x and y have the same type at $(\text{Thread},1)$. This is sufficient for us
 1641 to check $x = y$ at $(\text{Thread},1)$, meaning that adding that instruction to the stack maintains its
 1642 well-typedness.

- Case **T-Memcpy**

1644 Immediate via use of the **S-Memcpy** rule. We just need to show that the environment remains
 1645 well typed, which we know via our lemmas about **update** and **get**.

1646 **Memory Rules.**

- Case **T-Decl**

1649 By inversion, we are using **S-Decl** rule for evaluation. Our well-typed expression lemma gives
 1650 us that v is well-typed, so it follows from our assumptions and our lemmas about extending
 1651 environments that the extended η and s are well-typed by $\Gamma, x:\pi \tau$.

- Case **T-Free**

1653 Trivial.

- Case **T-Alloc**

1655 By inversion we are either in the **S-Alloc-Local** or **S-Alloc-Global** rules. In either case, we
 1656 assume that $\Gamma, x:\pi \tau[]^l$ checks s , meaning we can use our extended environment lemmas and
 1657 the **T-Seq** and **T-Free** rules to check these cases.

- Case **T-Alloc-Shared**

1659 Same as previous case.

- Case **T-Partition**

1661 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
 1662 inversion on our step relation we must be in the **S-Partition** case, so we have $\eta, \sigma, \Sigma, t, b, p, \Psi, \Phi \vdash_m^\pi$
 1663 $\text{partition}/xycs \sim \text{init}_\psi; s'; \text{dec}_\psi; \text{wait}_\psi \dashv_m \eta', \sigma', \Sigma', \Psi, \Phi$ where $s' = s[(y + c \cdot p)/y]$ and
 1664 $(\eta', \sigma', \Sigma') = \text{rename}(\eta, \sigma, \Sigma, x, y, \pi/c)$. Via our well-typed renaming lemma contexts we know
 1665 that we can check the renamed environments in the extended environment $\Gamma, y:\pi/c \tau[]^l, x:\pi$

1667 $\tau[]^l$, and this context is also sufficient to check s' (via a substitution-preserves-typing lemma
 1668 that is obvious). Using the **T-Skip** rule this is exactly what we need to show to complete this
 1669 case, as the thread sync primitives check trivially via their typing rules.

- 1670 • Case **T-Claim**

1671 Essentially the same as **T-Partition**.

- 1672 • Case **T-Lower**

1673 Essentially the same as **T-Partition**.

1674 *Control Rules.*

- 1675 • Case **T-If**

1676 We have by assumption that **if** e then s_1 else s_2 steps to some s' , and by inversion we
 1677 know that either e evaluates to **true** and s' is s_1 , or e evaluates to **false** and s' is s_2 .

1678 In either case, our inductive hypotheses is sufficient to tell us that these are well-typed. In
 1679 particular, in both cases our IHs tell us that the amount of memory used by stepping each
 1680 branch of the **if** is less than the amount of memory computed by the type system for each
 1681 branch. Because the whole **if** expression checks using the greater of the memory usage of m_1
 1682 or m_2 (i.e., the memory usage on each branch), the resulting usage for the whole conditional
 1683 is also bounded by the type system.

- 1684 • Case **T-Skip**

1685 Trivial: **skip** does not step

- 1686 • Case **T-Seq**

1687 In this case we have that s_1 and s_2 are both well-typed (with m_1 memory and m_2 memory
 1688 respectively), and $m = \mathbf{max}(m_1, m_2)$. We also have by inversion that s_1 either steps to **skip** or
 1689 s'_1 , and our inductive hypothesis tells us that s'_1 is well-typed.

1690 In the former case we can use the **S-Seq-Done** rule to trivially finish the case. In the latter,
 1691 our IH allows us to finish the case, since m_1 is always $\leq \mathbf{max}(m_1, m_2)$

- 1692 • Case **T-While**

1693 By inversion, we have that

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

1694 We also have by assumption that e and s are well-typed. With this information, through use
 1695 of the **T-If**, **T-Seq**, **T-While**, and **T-Skip** rules, we can conclude that the result of this rule is
 1696 also well-typed.

- 1697 • Case **T-Function-Call**

1698 By inversion we have that

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

1699 and also that $\Sigma(f) = \{[x_1 : \tau_1, \dots, x_n : \tau_n], s\}$, and $\eta, \sigma, \Sigma, t, b, p \vdash^\pi e_i \Downarrow v_i$, and $m' \leq m$.

1700 We also have from the premises of our case that $f : \pi \mathbf{Fun}(x_1 : \pi_1 \tau_1, \dots, x_n : \pi_n \tau_n, \pi, m') \in \Gamma$, and
 1701 $\Gamma \vdash^\pi e_i : \tau_i$, and $m' \leq m$.

1702 Our lemma for expression well-typedness tells us that that each v_i is a well-typed value,
 1703 and our assumption that $\Gamma \vdash \eta, \sigma, \Sigma$ tells us that $\Sigma(f)$ is a well-typed function and thus that
 1704 $\mathbf{fns} \Gamma, x_i : \pi_i \tau_i \vdash_m^\pi s$. We can take the union of this with Γ to produce $\Gamma, x_i : \pi_i \tau_i \vdash_m^\pi s$, which
 1705 clearly checks s and the output environments.

□