

SIMT-Step Execution: A Flexible Operational Semantics For GPU Subgroup Behavior

ANONYMOUS AUTHOR(S)

GPU hardware implements a SIMT execution model, where small groups of threads, called subgroups (or warps in CUDA), execute synchronously. Languages expose this through high-performance subgroup-level APIs. However, providing precise subgroup semantics in languages is challenging, as compilers may transform the program, potentially disrupting source-level synchronous behavior even if the hardware is synchronous. As a result, no GPU programming language provides rigorous semantics for subgroup execution.

In this work, we present SIMT-Step, a formal and flexible operational semantics for subgroup execution. At its core is a new semantic object, dynamic basic blocks, which enables precise specification of converged subgroup execution. SIMT-Step then provides flexibility for the execution of instructions, which can be collective, synchronous, or independent. We propose several candidate instantiations of SIMT-Step and design a suite of idiomatic tests to distinguish them, highlighting counter-intuitive behavior that arises under relaxed variants. We implement SIMT-Step in TLA+ and validate the behavior of the tests. To investigate how closely SIMT-Step models real-world GPU behavior, we conduct a fuzzing campaign, spanning ten GPUs and eight vendors. Our empirical study shows that non-synchronous behaviors are rare and appear on only a small number of devices; however, detailed investigation into these behaviors was inconclusive as to whether they are intentional or not. Combined, these contributions provide both a theoretical foundation and practical tools for reasoning about subgroup semantics in GPU programming languages.

1 Introduction

GPUs are now essential to modern computing, driving workloads in machine learning, graphics, and scientific simulation. Their impressive efficiency stems from a massively parallel execution model, but comes at a cost: GPU programming models are complex, evolve rapidly, and often leave important behaviors underspecified. As GPUs grow in importance, the need for precise, formal semantics becomes critical. Specification designers must balance abstraction, portability, and implementation flexibility. Without rigorous foundations, the community is left to rely on fragile assumptions, risking subtle bugs and missed optimizations. Modern GPU programming frameworks can be vendor-specific, e.g., CUDA [43] for NVIDIA hardware, or cross-platform, e.g., Vulkan [27], which uses the portable SPIR-V intermediate shader (kernel) language.

Over the years, components of the GPU programming model have been formalized. For example, given the complexity of the GPU hierarchy, much attention has been given to the memory consistency model. This includes formalizations of existing language specifications [2, 52], new model proposals [18], and empirical investigations [29]. Notably, both PTX (NVIDIA's IR for CUDA) and SPIR-V now include formal specifications [34, 51]. Other works have formalized control flow properties [30] and forward progress guarantees [17, 49] for GPU systems. Still, many aspects of the GPU programming model remain underspecified. One increasingly critical example is *subgroup behavior*. Subgroups (warps in CUDA) are small groups of threads that execute together in a SIMT (Single Instruction, Multiple Threads) manner ([44, §7.1]). Due to their close proximity in hardware, subgroup threads can efficiently cooperate; these capabilities are exposed to programmers through subgroup APIs in GPU languages, and they are widely used in performance-critical code, e.g., [8].

Although subgroup threads nominally execute together under the SIMT model, two compounding issues complicate their specification. First, subgroup operations may appear in divergent control flow, where threads within a subgroup take different control flow paths, making it unclear which threads are required to participate. Second, the degree of implicit synchronization provided within a subgroup, both in regular execution and during subgroup operations, is not well specified, and compiler transformations might further obscure these behaviors.

1.1 Examples: Intuition and Nuance

We highlight some of these difficulties with two examples. We use a generic pseudo GPU kernel language that resembles modern GPU kernel languages. To more precisely highlight the issues around subgroup synchronization, we will assume a sequentially consistent memory module throughout our examples. Later, we will show how to extend our model with fragments of a more realistic relaxed memory model (Sec. 4.5). Global memory (accessible across all threads) is denoted with g .

Converged, Diverged, and Merged. We start by providing intuition about subgroup operations with the example in Fig. 1, which is executed by two threads in the same subgroup. The `subgroupAdd` operation simply adds the provided values across the subgroup from participating threads. The operation on line 3 is executed by both threads and returns 2 (as each thread contributes 1). However, on line 5, the program contains divergent behavior, where threads from the same subgroup take different paths. The subgroup operations on lines 7 and 10 are described as *non-uniform*, i.e., they are executed by a subset of a subgroup. In this case, the operations on lines 7 and 10 are executed by threads 0 and 1, respectively, resulting in the value 1 for each thread. Finally, control flow merges for the subgroup operation on line 12, which is executed by both threads and returns 2.

This simple example already shows some nuances. Intuitively, a compiler should not flatten the `if/else` control flow even though both paths contain the same instruction, as it would impact the participating threads, returning 2 instead of 1. Similarly, the system should ensure that both threads converge before the subgroup operation on line 12.

Lockstep or Independent? Fig. 2 presents another example with less obvious behavior. Similarly, this example is executed by two threads in the same subgroup, each with a unique thread ID (`tid`). Each thread performs local initializations before reconverging at line 9, where both execute two memory stores (at locations $g[0]$ and $g[1]$), potentially separated by a subgroup operation. First, consider the program *without* the subgroup operation (line 10). Many prior works, including formal specifications [7, 33], modeled *lockstep* subgroup execution, with all converged threads advancing together instruction by instruction. This assumption seems to remain common in developers, where lockstep execution is often taken as the default mental model. Under this constrained model, threads 0 and 1 would execute line 9 together, storing the value 0 to $g[0]$ and $g[1]$, respectively. After completion, they advance to line 11, where threads 0 and 1 store the value 1 to $g[1]$ and $g[0]$, respectively. This tightly synchronized execution permits only the *Lockstep* behavior.

```

1 int main() {
2   // exec. by {0,1}; result of 2
3   int val_a = subgroupAdd(1);
4   int val_b;
5   if (tid == 0)
6     // exec. by {0}; result of 1
7     val_b = subgroupAdd(1);
8   else
9     // exec. by {1}; result of 1
10    val_b = subgroupAdd(1);
11  // exec. by {0,1}; result of 2
12  int val_c = subgroupAdd(1);
13 }

```

Fig. 1. An example showing different behaviors of subgroup collective operations in converged, diverged, and merged code blocks.

Init: $g[0] = 0, g[1] = 0;$

```

1 int main() {
2   int index_a, index_b;
3   if (tid == 0) {
4     index_a = 0; index_b = 1;
5   }
6   if (tid == 1) {
7     index_a = 1; index_b = 0;
8   }
9   atomicStore(&g[index_a], 0);
10  (opt.) subgroupAdd(0);
11  atomicStore(&g[index_b], 1);
12 }

```

Allowed behaviors

Lockstep: $g[0] == 1 \ \&\& \ g[1] == 1 \ ;$
Interleaving: $g[0] == 1 \ \&\& \ g[1] == 1 \ ||$
 $g[0] == 0 \ \&\& \ g[1] == 1 \ ||$
 $g[0] == 1 \ \&\& \ g[1] == 0 \ ;$

Fig. 2. An example executed with two threads each in the same subgroup. The allowed behaviors explore whether subgroup threads are guaranteed to execute in lockstep and whether subgroup operations provide synchronization.

99 However, GPU specifications have shifted toward allowing *independent* intra-subgroup execution,
 100 where threads do not have to execute synchronously. Thus, interleaving behaviors would be
 101 permitted. Specifically, thread 0 could run first, storing 0 to $g[0]$ and then 1 to $g[1]$, followed by
 102 thread 1 storing 0 to $g[1]$ and then 1 to $g[0]$, resulting in $g[0] == 1 \ \&\& \ g[1] == 0$. Reversing
 103 the thread order yields another valid outcome: $g[0] == 0 \ \&\& \ g[1] == 1$.

104 Even with lockstep hardware, a relaxed specification can still be exploited by an optimizing
 105 compiler via the *as-if* rule and cross-thread reasoning. For example, given a programming language
 106 with an interleaving specification, but a backend with lockstep behavior, a compiler may transform
 107 the program as long as permitted behaviors are preserved. In Fig. 2, it could rewrite line 4 as
 108 $\text{index_a} = 1; \text{index_b} = 0;$, causing both threads to store 0 to $g[0]$ and 1 to $g[1]$ in the final
 109 store (line 11), yielding a legal interleaved behavior despite synchronous hardware. This rewrite
 110 makes both branches identical, enabling flattening and eliminating divergence, which is inefficient
 111 on GPUs. It may also improve memory access patterns (e.g., avoiding bank conflicts [10] or enabling
 112 coalescing [13, 54]). Thus, platform implementers may favor more permissive specifications that
 113 enable such optimizations, even on synchronous hardware.

114
 115 *Subgroup Operation Synchronization.* Now consider the test with the line 10 subgroup operation
 116 included. This test raises the question: does a subgroup operation imply subgroup synchronization?
 117 That is, if the specification permits interleaved behaviors for memory operations, does a subgroup
 118 operation synchronize threads? If so, then including the subgroup operation would guarantee the
 119 lockstep behavior. In this example, the addition is trivial as each thread provides a constant (0),
 120 and thus, an optimizing compiler could be tempted to replace the operation with a constant 0. Is
 121 removing the subgroup operation entirely allowed? If so, and under interleaving semantics, then
 122 one could imagine a sequence of transformations: first, removing the subgroup operation; and
 123 next, inducing an interleaving behavior. Thus, even a subgroup operation may not be sufficient to
 124 provide synchronous behavior.

125 126 1.2 SIMT-Step Execution: A Flexible Operational Semantics for Subgroups

127 In this paper, we introduce *SIMT-Step Execution*, a flexible operational semantics for specifying
 128 subgroup behavior in GPU programming languages. SIMT-Step can be instantiated in multiple ways,
 129 for example, to account for the range of behaviors discussed in Sec. 1.1 and to explore their broader
 130 consequences. SIMT-Step was developed in discussion with official GPU specification groups;
 131 and while there has not yet been any official adoption, SIMT-Step is designed to accommodate a
 132 spectrum of behaviors and optimizations.

133
 134 *Dynamic Blocks.* Subgroup operations are unusual because they are both *collective*, combining
 135 inputs across threads, and *non-uniform*, i.e., not all threads are required to participate, as shown
 136 in Fig. 1. SIMT-Step first provides the semantic foundations to specify the set of participating
 137 threads in subgroup operations, even under divergence. To do this, we extend the notion of a
 138 dynamic instruction instance to basic blocks, defining the *dynamic block* (Sec. 3). All threads begin
 139 execution in the dynamic entry block; threads that take the same branch continue together into a
 140 subsequent shared dynamic block, with designated dynamic blocks where divergent threads will
 141 merge. Threads belonging to the same subgroup that execute the same dynamic block are said to be
 142 *active threads* for that dynamic block (and its constituent instructions). If a dynamic block contains
 143 a subgroup operation, then all of its active threads will participate in that operation.

144
 145 *Flexible Instruction Execution.* Next, SIMT-Step provides flexibility on how instructions are exe-
 146 cuted. Each instruction can be executed as one of the following:

147

Table 1. SIMT-Step models, each with a description and a documenting tests. Tests are given as pairs, where the model allows the relaxed behavior in the first test, and enforces the synchronous behavior in the second. The sg subscript is given to tests where optional subgroup operations are included. The first model (strongest) has no test in which it shows relaxations; conversely, the last model (weakest) has no test in which it guarantees the synchronous behavior. Color shading indicates semantic complexity, with the lighter shaded SSO requires extra guards in semantics, and the darker shaded Spec and Symb requiring advanced semantic concepts, such as speculation and symbolic reasoning.

SIMT-Step Model	Description	Tests
Collective Memory (CM)	Memory/subgroup ops, branches, and labels are collective	(NA, Fig. 3)
Sync. Memory (SM)	Memory is synchronous but not collective	(Fig. 3, Fig. 2)
Sync. Control Flow (SCF)	Synchronizes at basic block entry, exits via collectives	(Fig. 2, Fig. 9 _{sg})
Sync. SG Op (SSO)	Synchronizes at SG ops and associated control dependencies	(Fig. 9 _{sg} , Fig. 9 _{sg})
Speculative SG Op (Spec)	May speculatively run SG ops before synchronization	(Fig. 10, Fig. 11)
Symbolic SG op (Symb)	SG Ops don't synchronize and return symbolic values	(Fig. 11, NA)

- (1) **Independently:** Threads execute these instructions independently under a standard concurrency model, with any allowed interleaving possible.
- (2) **Synchronously:** These instructions implicitly synchronize active threads.
- (3) **Collectively:** The operations execute collectively (atomically) across active threads.

SIMT-Step Instantiations. By specifying different sets of independent, synchronous, and collective instructions, we define several instantiations of SIMT-Step that could be candidates for inclusion in language specifications. These instantiations can be ordered from stronger (more synchronization) to weaker (more independence). The various models, short descriptions, and documenting tests are summarized in Tab. 1, and their definitions are formalized throughout the paper. Our tools and implementations focus on the first four models, which we dub *direct models*, as we deem the last two models too complex for detailed exploration.

We implement the SIMT-Step direct models in TLA+, providing executable semantics (Sec. 5). To facilitate further testing and exploration, we provide a frontend that accepts tests written in a higher-level shading language: GLSL.

Empirical Exploration. One challenge in specifying subgroup behavior is the lack of information about how current GPUs behave, which makes the feasibility of adopting a specification unclear. To investigate this, we conduct a large-scale empirical study across real-world devices (Sec. 7). We generalize our direct model tests across different memory access patterns, yielding 10 base tests. To test for behaviors introduced by compiler transformations, we apply a semantics-preserving fuzzer to each test [11], generating 100k fuzzed tests, which run on ten GPUs spanning eight vendors.

We observe that most GPUs exhibit strongly synchronous subgroup behavior across our tests. We performed detailed follow-up investigation on the rare non-synchronous behaviors, but were unable to conclusively determine whether they reflect intentional behavior or unintended anomalies. A clear exception is that nearly all GPUs do *not* treat memory operations as collective. This behavior is illustrated in Fig. 3, where all non-NVIDIA and non-Apple platforms show the non-collective outcome. This observation is noteworthy because there exist subgroup-select idioms (e.g., from [36]) that rely on collective memory operations.

Scope and Limitations. This work focuses on intra-subgroup behavior; however, we believe our semantics compose straightforwardly across scopes, e.g., by instantiating each rule per subgroup. To separate execution behavior from memory effects, we assume a sequentially consistent (SC) memory model for most of this paper. However, real GPU programming languages have relaxed

memory models, and some, such as SPIR-V, do not support SC. To account for this, Sec. 4.5 discusses extending SIMT-Step with release/acquire memory operations based on [46], and how our examples change under these semantics. Future work could extend SIMT-Step toward more complete relaxed memory models, for example following works such as [21, 41].

Although we utilize many components and syntax from SPIR-V and rely on some of its control flow properties (see Sec. 2.2), SIMT-Step is designed to be general. Similarly, although our semantics utilize SPIR-V instructions, we make no claim of being an official formalization of SPIR-V. We note that CUDA does not provide the structured control flow or merge properties that SIMT-Step requires. Thus, general CUDA cannot be modeled using SIMT-Step; however, there are likely pragmatic subsets that could be targeted. We also note that CUDA's independent thread scheduler (Volta onward) [45], which is documented to significantly relax warp behaviors, still exhibits synchronous behaviors on NVIDIA devices. Nevertheless, specifications like this indicate a growing interest in more relaxed execution models.

While SIMT-Step supports loops (as they appear in SPIR-V), their semantics require additional bookkeeping compared to conditionals. These mechanisms are encoded precisely in our TLA+ semantics, where the dynamic block stack and merge rules handle per-iteration control flow. To keep the presentation accessible, the main paper text focuses on control flow encoded by conditionals.

Contributions. In summary, our contributions are:

- We introduce *SIMT-Step*, a flexible operational semantics for subgroup behavior in GPU programming languages. It is built around a new semantic object, the dynamic block (Sec. 3), and can support varying degrees of independent, synchronous, and collective execution.
- We define a range of SIMT-Step models, ranging from fully lockstep to symbolic execution (Sec. 4). We verify the behavior of a pragmatic subset of the models using TLA+ (Sec. 5).
- To understand real-world behaviors, we conduct a large empirical study. The tested platforms largely showed strongly synchronous behavior, although we also observed rare violations of lockstep execution whose root causes we were unable to determine (Sec. 7).

2 Background

2.1 GPU Architecture and Programming Models

GPUs are high-throughput, massively parallel accelerators composed of many *compute units* (CUs), called *streaming multiprocessors* by NVIDIA. Each CU contains many lightweight *processing elements* (PEs) that execute instructions in parallel. The corresponding programming model is also organized hierarchically: *threads* execute instructions on a PE. Threads are grouped into *subgroups* (*warps* in CUDA), which typically contain 16–128 threads depending on the architecture. Multiple subgroups that execute on a single CU form a *workgroup*, and all workgroups together constitute the *grid*.

Within a subgroup, threads execute according to the *SIMT* (Single Instruction, Multiple Threads) model. Conceptually, all these threads follow the same instruction stream and typically execute instructions together. When control-flow differs among threads, the subgroup *diverges*, some threads take one branch while others take another, and later *reconverge* (or merge) to resume joint execution. The exact rules governing divergence and reconvergence are crucial for reasoning about both

```
Initial: int *loc = 0;
1 atomicStore(loc, tid);
2 int read = atomicLoad(loc);
3 assert(subgroupAllEqual(read));
```

Allowed behaviors

Collective Memory: assertion passes
Non-collective Memory: assertion fails

Fig. 3. This test is executed by many threads across subgroups. Each writes their unique id to `loc` and then reads from the `loc`. Under collective memory, all threads within the subgroup will read the same value. Otherwise, the load can return different values to different threads in the subgroup. The `subgroupAllEqual` checks if the arguments across all threads are all equal.

performance and correctness, as they determine when threads efficiently execute together (and participate in subgroup operations) or when they may execute separately, as discussed in Sec. 2.2.

GPU programming frameworks expose this hardware hierarchy through low-level APIs designed for high-performance computing. These include *CUDA* [43], *DirectX* [39], *Vulkan* [27], and *Metal* [3], each providing different dimensions of portability. We adopt Vulkan terminology in this work, except where more familiar terms (e.g., thread vs. invocation) improve presentation. A typical GPU program consists of host-side code for memory and dispatch, and device-side shaders (or kernels) defining the computation. Shading languages are used to write programs that execute on the GPU. They follow a data-parallel model, where an entry function is dispatched across many threads, each with a unique identifier within the execution hierarchy for indexing and data access. Different frameworks pair with different shading or kernel languages: CUDA uses CUDA C++ (and PTX as an IR), Vulkan uses SPIR-V IR, DirectX uses HLSL, and Metal provides the Metal Shading Language.

Subgroup Operations and Semantics.

Modern GPU languages expose a variety of highly efficient *subgroup operations*, which operate across a subgroup. These operations typically take an argument from each active thread and perform reductions, broadcasts, shuffles, and ballots. While these operations are *collective*, they can be specified to be *non-uniform*: not all threads in a subgroup are required to participate. This work focuses on these non-uniform variants. Language specifications typically leave the participating set loosely defined, with the intuition that only threads reaching the operation, i.e., via the same control flow path, take part. However, these properties have never been formally modeled.

Classically, subgroup execution models have also included various implicit synchronization behavior. It remains unclear whether operations like memory accesses or branches enforce any synchronization among threads. Earlier documentation hinted at implicit synchronization, but this documentation has been omitted or revised in many current specs.

In our supplementary material, we provide a history of how subgroup (warp) semantics have evolved throughout GPU history. We also discuss how formal analysis tools have modeled subgroup execution. It is worth noting that CUDA warp operations take an additional mask that explicitly denotes the participating threads. However, this does not solve the issue, as this mask is often obtained by calling the `__activemask` instruction, which returns a mask of the active threads.

SPIR-V. SIMT-Step is implemented using a subset of SPIR-V, a low-level, SSA-based intermediate representation for GPU programs. It serves as a common compilation target, with several widely used frontend languages, including HLSL [40], GLSL [22], and OpenCL C [26]; it can target a wide range of backend GPUs, including those from NVIDIA, AMD, Intel, Qualcomm, and ARM. In most cases, we present examples in GLSL for clarity and readability, as it resembles C code. Although

Table 2. SPIR-V instructions used in the paper

Instruction	Description
<i>Subgroup Operation</i>	
OpGroupAdd	Returns the sum of values provided by all active threads
OpGroupAllEqual	Return true if the provided values from all participating threads are equal
OpGroupBroadcast	Broadcasts a value from one specified participating thread to all others
<i>Label Instructions</i>	
OpLabel	Marks the beginning of a basic block with a unique ID
<i>Block Terminating Instructions</i>	
OpBranch	Unconditional branch to another label
OpBranchCond	Conditional branch based on a boolean
OpReturn	Return from a function/shader
<i>Merge Instructions</i>	
OpSelectionMerge	Declares the merge block for a conditional construct

our tools support a wider set of instructions (see Sec. 5), Tab. 2 lists the SPIR-V instructions most widely used in this paper, including subgroup and control flow instructions.

2.2 Structured Control Flow and Maximal Reconvergence

SIMT-Step relies on two key features, both provided by SPIR-V: structured control flow and *maximal reconvergence*, which dictates how threads merge after divergence. As a target language, SPIR-V requires any source shading language to support these properties to generate valid code.

```

1 %entry = OpLabel
2   OpSelectionMerge %merge
3   OpBranchConditional %c %then %else
4
5 %then = OpLabel
6   ; computation
7   OpBranch %merge
8
9 %else = OpLabel
10  ; computation
11  OpBranch %merge
12
13 %merge = OpLabel
14  %res = OpGroupAll %arg

```

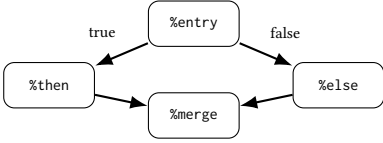


Fig. 4. An example of control flow in SPIR-V and the resulting control flow graph

Structured Control Flow. SPIR-V enforces structured control flow by constraining valid control flow graphs and providing instructions that define both behavior and structure. Each block starts with `OpLabel` and ends with a terminator like `OpBranch`. Branching constructs must declare a reconvergence point using a merge instruction (e.g., `OpSelectionMerge` for conditionals), placed immediately before the branch.

Fig. 4 illustrates a simple `if-then-else` construct in SPIR-V alongside its control flow graph. For clarity, some instructions are simplified, without changing semantics. Each basic block begins with a static instruction `OpLabel` (blue) assigning a unique ID and ends with a terminator (red). When a block branches to multiple successors—e.g., `%entry` to `%then` and `%else`, SPIR-V requires a merge instruction to specify the reconvergence point. Unstructured control flow (e.g., `goto`) is disallowed; all control flow must be statically nested and explicitly merged.

Maximal Reconvergence. SPIR-V specifies a model for handling divergent control flow known as *maximal reconvergence* [24]. Under this model, all subgroup threads that execute a merge instruction must reconverge the first time they execute the merge target. Fig. 4 illustrates this with a subgroup operation, `OpGroupAdd`, placed in the merge block. Under maximal reconvergence, threads that diverge into `%then` and `%else` are guaranteed to reconverge at `%merge`, ensuring that the operation executes over the same set of threads that executed `%entry`.

3 SIMT-Step Foundations: Dynamic Blocks

We build on standard operational semantics, where a program execution E is described as a sequence of *atomic steps*. Each step corresponds to the execution of a static instruction, referred to as a *dynamic instance* (following SPIR-V terminology [16]), denoted by i . An execution is a sequence of such instances: $E = i_0 \rightarrow i_1 \rightarrow \dots$. In a concurrent program, E consists of interleavings of dynamic instances executed by all threads, potentially constrained by synchronization. We use E_t to denote the projection of E to instances executed by a thread t and i_t to denote that i was executed by t .

Dynamic blocks extend the notion of dynamic instances to basic blocks. A static basic block b , when executed, produces a dynamic block instance db . Multiple threads may execute the same dynamic block, which captures convergent thread execution. An execution E is now associated with a dynamic execution graph D , which records the dynamic blocks executed across all threads.

A control flow graph (CFG) can be straightforwardly produced from SPIR-V code given its structured control flow. Each node in a CFG corresponds to a basic block b , which contains the following components:

- **Block Label** ($b.label$): A unique label provided by the code.
- **Children** ($b.C$): The immediate children b in the CFG.
- **Merge Block** ($b.merge$): If b has multiple children, then it must specify the basic block where the children merge.

A dynamic block db extends a basic block with the following:

- **Basic Block** ($db.b$): The associated static basic block.
- **ID** ($db.id$): A unique ID as there can be multiple dynamic blocks for a given basic block.
- **Threads** ($db.\mathcal{T}$): The set of all threads that execute db . To ease presentation, we will assume that all threads in $db.\mathcal{T}$ belong to the same subgroup. If more threads are executing the kernel, then they can simply be partitioned into disjoint sets based on their subgroup.
- **Children** ($db.C$): The children dynamic blocks of db in the dynamic execution graph.
- **Dynamic Block Merge Target** ($db.merge$): If $db.b$ has a merge instruction, then this is the dynamic block where $db.\mathcal{T}$ will reconverge after any potential divergence.

Each dynamic block db must align with the structure of its corresponding basic block ($db.b$) in the CFG. For example, the basic blocks of the dynamic children of db must correspond to basic children of $db.b$. The same condition applies to the merge relationships.

Convergence and Divergence. The dynamic execution graph tracks converged execution through the Threads (\mathcal{T}) set for each dynamic block. The following rules dictate constraints on this set and on the dynamic execution graph structure.

- **Initial:** There exists a single initial dynamic block db_0 corresponding to the first basic block in the shader: $db_0.\mathcal{T}$ contains all the threads.
- **No spurious divergence:** Once a thread starts executing a dynamic block, it will continue executing the same dynamic block until a block terminating instruction.
- **Branching:** If two threads are executing the same dynamic block and then branch to the same basic block, they must also branch to the same dynamic block.
- **Reconvergence:** If db contains a merge block ($db.merge$), then there must exist db' later in the execution that is associated with $db.merge$, which is executed by the same thread set \mathcal{T} . Furthermore, there must *not* exist another db'' such that (1) db'' lies in the execution sequence between db and db' ; and (2) db'' 's associated basic block is the merge target of db . Intuitively, this states that threads must reconverge at a merge block as early as possible. This property captures the maximal reconvergence [24].
- **Non repeating:** A thread t cannot execute the same dynamic basic block twice.
- **Acyclic:** The dynamic execution graph must be acyclic.

These rules allow the dynamic block thread set (\mathcal{T}) to specify which threads are converged in db , and thus, if a subgroup operation appears in db , it must be executed by all threads in $db.\mathcal{T}$. Although this work focuses on subgroups, it can naturally support the more common workgroup-level barrier semantics. The textual alignment requirement of workgroup barriers [20, 33] seems

```

1 for (int i = 0; i < 3; i++) {
2   // loop block
3   if (tid == i || i == 2) {
4     // cond block
5     subgroupOp(...);
6   }
7 }

```

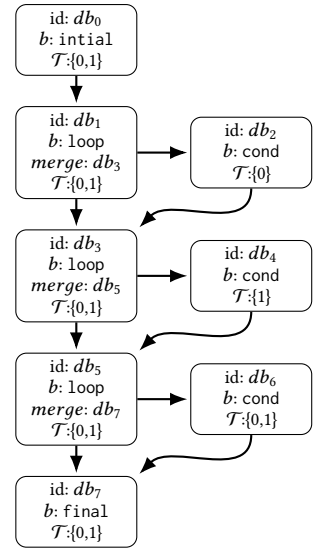


Fig. 5. An example program where a subgroup operation is executed conditionally within a loop. The dynamic execution graph illustrates how threads diverge and reconverge.

393 to correspond with dynamic basic blocks, although with stronger uniformity requirements than
 394 subgroup operations. That is, a db containing a workgroup barrier must be executed by (and thus
 395 synchronizes across) all threads in the workgroup, otherwise it is undefined behavior.

396 *Example.* Figure 5 shows a GLSL loop containing a conditional with a subgroup operation inside.
 397 The conditional is non-uniform and may split the subgroup. The dynamic execution graph below
 398 approximates basic blocks from the high-level code (a SPIR-V version would be more precise but
 399 more complex). For example, the cond dynamic blocks (db_2 , db_4 , and db_6) are shown as one dynamic
 400 block when they would actually be separate blocks for the selection header, conditional body, and
 401 merge block of the conditional. We assume the program is executed with two threads, with tid
 402 values of 0 and 1, both in the same subgroup.

403 The execution begins with both threads in the initial dynamic block (db_0), as noted by \mathcal{T} . Both
 404 threads then enter the first loop iteration block (db_1) where $i = 0$. This dynamic block diverges
 405 depending on the conditional, but the threads must reconverge at the next loop iteration (db_3). The
 406 conditional splits the subgroup, with thread 0 executing the conditional block (db_2). This block
 407 contains a subgroup operation. Given the above rules, this subgroup operation is only executed
 408 by thread 0, as it is the sole thread in \mathcal{T} for (db_2). Similarly, in the next loop iteration ($i = 1$),
 409 the next conditional block db_4 is executed only by thread 1, and its constituent subgroup operation
 410 is only executed by thread 1. Because the dynamic execution graph must be acyclic, db_2 and db_4
 411 must be disjoint dynamic blocks, and thus it is guaranteed that the first two iterations of the loop
 412 must execute two distinct instances of the subgroup operation with thread 0 and thread 1 as the
 413 participating threads, respectively. Merging them is not allowed. In the third and final loop iteration,
 414 $i = 2$, the conditional is true for both threads, so both threads execute db_6 , and thus both participate
 415 in the subgroup operation execution before proceeding to the final block db_7 .

416 *Limitations: Escaping Merge Targets.* We assume that given a dynamic block db with a merge
 417 block $db.merge$, they must both have the same thread set, i.e., $db.\mathcal{T} = db.merge.\mathcal{T}$. However,
 418 SPIR-V provides an exception to this condition: a thread can terminate before executing the merge
 419 target. We have explored candidate semantics under these conditions in our TLA+ implementation,
 420 but more work is needed to understand the impact of such behaviors.

423 4 SIMT-Step Operational Semantics for Subgroup Behavior

424 Utilizing dynamic blocks, we can design a flexible operational semantics for subgroup execution.
 425 We will reference the models of Tab. 1 and detail how they are instantiated. For simplicity, we
 426 restrict our semantics to a single subgroup unless explicitly stated otherwise. Extending to arbitrary
 427 grid configurations involves partitioning \mathcal{T} for each dynamic block, which is supported in our
 428 TLA+ implementation. Our semantics adopt the style of TLA+, including using guarded transition
 429 systems and the ability to construct semantic objects, such as dynamic blocks, during execution.

430 The language shown in Fig. 6 defines a single shader function f composed of at least one
 431 basic block B . Each basic block begins with a label instruction, followed by zero or more normal
 432 (N) instructions, an optional *Merge* instruction, and ends with a block terminator *Term*. The
 433 standard instructions we model include (sequentially consistent atomic) memory loads and stores,
 434 subgroup operations, and an abstract thread-local *Local* instruction. The subgroup operation takes
 435 an additional *op* argument that specifies its operation, e.g., *add*. The instructions are (statically)
 436 partitioned into the following sets: labels (*Label*), block-terminating instructions (*Term*), merge
 437 instructions (*Merge*), and normal instructions (N), as defined in Fig. 6.

438 *State and Initialization.* The system state shown in Tab. 3 captures three levels: individual thread
 439 state (σ), group thread state (Σ), and shared state, including memory M and dynamic blocks D .

Table 3. Components of the SIMT-Step system state.

Name	Definition	Description
σ_t	$\triangleq \langle pc_t, db_t, R_t \rangle$	Thread state: the program counter, current dynamic block, and register file for thread t .
Σ	$\triangleq \langle \sigma_0, \sigma_1, \dots, \sigma_{n-1} \rangle$	Group state: tuple of thread states in a subgroup of size n .
M	$: a \mapsto v$	Global memory mapping each address a to a value v .
D	$= \{db_0, \dots, db_n\}$	Set of dynamic blocks forming the dynamic execution graph.
<i>State</i>	$\triangleq \langle M, \Sigma, D \rangle$	Full SIMT-Step system state.

We extend the dynamic block definition from Sec. 3 with three auxiliary fields: ms is a stack of dynamic block ids, which is used to manage reconvergence in the merge block. Each dynamic block maintains its own merge stack ($db.ms$), where the top element can be accessed via the *top*. The stack records dynamic blocks that were designated as merge targets by previously executed control flow. sis is a dictionary mapping each static synchronous instruction in $db.b$ to a Boolean value indicating whether the subgroup is currently synchronized at that instruction. Unknown threads (U) track threads for which it is unknown whether they will be active in db .

The initial state for a specific shader program S is:

<i>entry</i>	$:=$ label of the entry basic block of S
db_0	$:= \langle id = 0, b = \text{entry}, C = \emptyset, \text{merge} = \text{none}, ms = [],$ $sis = [I \mapsto \text{False} \mid I \in \text{SyncInstr}(\text{entry})], \mathcal{T} = \text{AllThreads}, U = \emptyset \rangle$
$\Sigma(t)$	$:= \langle pc = pc(\text{entry}), db = db_0, R = R_0(t) \rangle \quad \forall t \in db_0.T$
M	$:= M_0$ (initial global memory, from host inputs)
D	$:= \{db_0\}$

The entry dynamic block db_0 corresponds to the entry basic block of S . Each thread t begins with its program counter at the entry label, an empty merge stack, and register state $R_0(t)$. We use σ to denote a single thread state. Given a group state $\Sigma = \langle \sigma_0, \sigma_1, \sigma_2, \dots, \sigma_{n-1} \rangle$, we use $\Sigma(i)$ to denote σ_i , and n is the subgroup size. The initial memory M_0 is provided by the host and shared among all threads. $\text{SyncInstr}(b)$ is the set of synchronous instructions in the basic block *entry* (which is flexible depending on the intended semantics), and *AllThreads* is the thread IDs of all threads.

Transition Rules. We provide two types of transition rules, first a single-thread transition rule with the signature:

$$\rightarrow_t \subseteq (\text{ThreadState} \times M \times D) \times (\text{ThreadState} \times M \times D).$$

These rules operate on a single thread state σ , possibly updating the group state of memory

tid	$\in \{0, \dots, n-1\}$	Thread IDs
ℓ	$\in \text{Label}$	Basic block labels
a	$\in \text{Addr}$	Memory addresses
r	$\in \text{Reg}$	Thread-local registers
v	$\in \text{Val}$	Values (e.g., integers)
S	$::= \text{shader } f \{B^+\}$	A shader function f
B	$::= \text{Label} ; N^* ; \text{Merge?} ; \text{Term}$	A basic block
<i>Label</i>	$::= \text{label}(\ell)$	A label instruction
N	$::=$ Load $r \leftarrow a$ Store $a \leftarrow \text{Operand}$ $r \leftarrow \text{SG_op}(op, \text{Operand})$ Local(Operand^*)	Normal instructions Load from memory Store to memory Subgroup operation An abstract local instr.
<i>Term</i>	$::=$ branch(ℓ) branchCond($\text{Operand}, \ell_1, \ell_2$)	Terminating instruction Unconditional branch Conditional branch
<i>Merge</i>	$::= \text{selectionMerge}(\ell)$	Merge instruction
<i>Operand</i>	$::= r \mid v$	A value or register

Fig. 6. The syntax for a simple GPU shader language (similar to SPIR-V) which we build our operational semantics for.

(through memory operations) or the set of dynamic blocks (through branching). These single-thread transitions define the smallest step relation governing individual thread behavior. At a wider scope, the *group transition rules* execute instructions collectively and have the following signature:
 $\rightarrow_g \subseteq (\text{GroupState} \times M \times D) \times (\text{GroupState} \times M \times D)$.

In this case, execution operates atomically across all active threads in the subgroup. These rules can update all thread local states, memory, and the set of dynamic blocks. We will utilize the guard $\text{Align}(pc, db)$, which checks whether all threads in the dynamic block db are aligned at the same program counter pc .

Modes of Instruction Execution. SIMT-Step can be instantiated in several ways to account for a variety of subgroup behaviors, including the models in Tab. 1. These different instantiations are realized by statically partitioning instructions into three sets: (1) **independent**: these instructions execute independently, as in traditional concurrency semantics; (2) **synchronous**: these instructions require all active threads to align on the instruction, but do not execute the instruction atomically; and (3) **collective**: these instructions synchronize *and* execute atomically across the active threads. The supplementary material provides a comprehensive table detailing the execution mode of each instruction class under the various SIMT-Step semantics model.

Auxiliary Functions. There are several auxiliary functions throughout that we will describe in the text, such as operand evaluation, local instruction execution, and dynamic block creation and retrieval. Their formal definitions and implementations are provided in the supplementary material.

4.1 Executing Within a Dynamic Block

We begin with instructions internal to a basic block, i.e., instructions that are not labels or branches. These are straightforward to execute because they do not require reasoning about divergence. Several variants are summarized in Fig. 7 and described below.

The most relaxed way that these instructions can be executed is independently, and correspond to the rules T-LOCAL, T-LOAD, and T-STORE. For the abstract local instruction in T-LOCAL, the instruction is evaluated by the auxiliary function *Exec*, producing an updated register file R' , after which the thread advances its program counter by one. T-LOAD and T-STORE follow the same pattern but read from and write to global memory, respectively. Allowing memory operations to execute independently within a basic block allows some flexible behaviors; for example, the interleaved behavior in Fig. 2, becomes possible. In the SIMT-Step models, SCF and weaker models permits independent execution around memory operations.

Synchronous. It may be desirable to strengthen the semantics within a basic block. To this end, we allow instructions to be defined as synchronous. These semantics are implemented in two steps. First, T-SYNC-ARRIVE fires once all threads are aligned on the instruction, as indicated by the guard $\exists db \in D (\text{Align}(db, \Sigma))$, which selects a dynamic block in which all active threads share the same program counter. This triggers an update to the synchronous-instruction status in the dynamic block ($db.sis$), setting the entry for the synchronous instruction to *True*. Once this status is set, threads proceed with the T-SYNC-EXECUTE step, where each thread may take a step individually according to the thread-local semantics (\xrightarrow{t}). Our SIMT-Step model SM instantiates memory operations as synchronous.

Collective. Instructions can further be strengthened to execute collectively, by taking a *collective* step, defined by the (G-COLLECTIVE) rules. These rules apply when all active threads in a dynamic block are aligned at the same instruction. In this case, the instruction is executed atomically across the threads, and its semantics (\xrightarrow{g}) are defined over the group state rather than an individual thread.

540	$\frac{I_{[\sigma.pc]} = \text{Local}(\text{Operand}) \quad R' = \text{Exec}(I_{[\sigma.pc]}, \sigma.R)}{\langle \sigma, M, D \rangle \xrightarrow{t} \langle \sigma[pc := pc + 1, R := R'], M, D \rangle}$	(T-LOCAL)
541		
542		
543	$\frac{I_{[\sigma.pc]} = \text{Load } r \leftarrow a \quad v = M(a)}{\langle \sigma, M, D \rangle \xrightarrow{t} \langle \sigma[pc := pc + 1, R := R[r \mapsto v]], M, D \rangle}$	(T-LOAD)
544		
545		
546	$\frac{I_{[\sigma.pc]} = \text{Store } a \leftarrow \text{Operand} \quad v = \text{eval}(\text{Operand}, \sigma.R)}{\langle \sigma, M, D \rangle \xrightarrow{t} \langle \sigma[pc := pc + 1], M[a \mapsto v], D \rangle}$	(T-STORE)
547		
548		
549	-----	
550		
551	$\frac{\exists db \in D \text{ Align}(db, \Sigma)}{\langle \sigma, M, D \rangle \xrightarrow{t} \langle \sigma, M, D[db.sis[I_{[\sigma.pc]}] := \text{True}] \rangle}$	(T-SYNC-ARRIVE)
552		
553		
554	$\frac{\sigma.db.sis[I_{[\sigma.pc]}] = \text{True} \quad \langle \sigma, M, D \rangle \xrightarrow{t} \langle \sigma', M', D' \rangle}{\langle \sigma, M, D \rangle \xrightarrow{t} \langle \sigma', M', D'[db.sis[I_{[\sigma.pc]}] := \text{False}] \rangle}$	(T-SYNC-EXECUTE)
555		
556		
557	-----	
558		
559	$\frac{\exists db \in D (\text{Align}(db, \Sigma) \wedge \forall i \in d.\mathcal{T} \Sigma(i) \xrightarrow{t} \Sigma'(i))}{\langle \Sigma, M, D \rangle \xrightarrow{g} \langle \Sigma', M, D \rangle}$	(G-COLLECTIVE-LOCAL)
560		
561		
562	$\frac{\exists db \in D (\text{Align}(db, \Sigma) \wedge \forall i \in db.\mathcal{T} I = I_{[\Sigma(i).pc]} \wedge (M', \{R'_u\}_{u \in db.\mathcal{T}}) = \text{ExecSG}(db.\mathcal{T}, I, \{\Sigma(u).R\}_{u \in db.\mathcal{T}}, M) \wedge \forall u \in db.\mathcal{T} \Sigma'(u) = \Sigma(u)[R := R'_u])}{\langle \Sigma, M, D \rangle \xrightarrow{g} \langle \Sigma', M', D \rangle}$	(G-COLLECTIVE-SUBGROUP)
563		
564		
565		
566		
567	$\frac{\exists db \in D (\text{Align}(db, \Sigma) \wedge \forall i \in db.\mathcal{T} I = I_{[\Sigma(i).pc]} \wedge (M', \{R'_u\}_{u \in db.\mathcal{T}}) \in \text{ExecMem}_{\text{collective}}(db.\mathcal{T}, I, \{R_u\}_{u \in db.\mathcal{T}}, M) \wedge \forall u \in db.\mathcal{T} \Sigma'(u) = \Sigma(u)[R := R'_u])}{\langle \Sigma, M, D \rangle \xrightarrow{g} \langle \Sigma', M', D \rangle}$	(G-COLLECTIVE-MEM)
568		
569		
570		
571		

Fig. 7. Rules for execution within a basic block.

Intuitively, once all the active threads in a dynamic block reach the same instruction, they execute it together as a single group-level operation. G-COLLECTIVE-LOCAL is the group version of T-LOCAL. Once threads are aligned, each can take the corresponding thread-local step, and the group-level transition updates the thread states to the resulting Σ' . G-COLLECTIVE-SUBGROUP defines collective subgroup operation. The equality $I = I_{[\Sigma(i).pc]}$ extracts the static instruction at the aligned pc . The term ExecSG applies the subgroup semantics, producing updated memory and per-thread register values. For example, if I is *subgroupAdd*, ExecSG sums the input values provided by all active threads and returns the result to each thread. The conclusion distributes these register updates to each thread and performs a single group-level transition to (Σ', M') .

G-COLLECTIVE-MEM handles collective memory operations. $\text{ExecMem}_{\text{collective}}$ invokes the collective memory semantics: given the active thread set, the instruction, their registers, and the global memory, it returns a set of possible outcomes representing the nondeterminism of concurrent memory effects when threads write to the same address. The operational rule chooses one such

outcome $(M', \{R'_u\}_{u \in db.\mathcal{T}})$ and applies it. The premise $\forall u \in db.\mathcal{T} \Sigma'(u) = \Sigma(u)[R := R'_u]$ updates each thread's registers with the values produced by the selected result.

In our SIMT-Step models, all direct variants (CM, SM, SCF, and SSO) execute subgroup operations collectively, as non-collective semantics quickly become complex (see Sec. 4.4). The strongest model (CM) also executes loads collectively, ensuring threads observe the same value when reading the same location. Collective store semantics are less clear, but could be defined for example, by selecting the value from the highest-ID thread. However, some GPU models (e.g., CUDA) leave this undefined, stating that “which thread performs the final write is undefined” [44, §7.1].

4.2 Collective Control Flow

We now describe control flow instruction semantics, which add complexity, as they must (1) account for potential divergence and (2) prepare for reconvergence at merge blocks. It is most straightforward to handle if control flow operations (branches, labels and merges) are executed collectively. That is, collective control flow instructions ensure that all threads within a dynamic block leave and enter basic blocks together, ensuring that the active thread sets for dynamic blocks are known at creation. We show the semantics of the collective conditional branch in G-COLLECTIVE-CBRANCH (Fig. 8). The unconditional branch is strictly simpler and provided in supplemental material along with the label and merge instruction, which are similar to G-COLLECTIVE-LOCAL.

In G-COLLECTIVE-CBRANCH, once an aligned dynamic block is selected and the branch instruction $branchCond(c, \ell_t, \ell_f)$ is identified, the rule calls get_child to obtain the successor dynamic blocks db_t and db_f for the true and false targets. get_child reuses an existing block when the merge stack indicates that the successor is the merge block; otherwise it creates a fresh dynamic block and inherits the parent's merge stack. The active threads are then partitioned into A_t and A_f depending on whether they satisfy the branch condition. Threads in A_t update their program counters to the basic block of db_t and move into that block; threads in A_f do the same for db_f . The group-level transition \xrightarrow{g} commits these updates and incorporates D_t and D_f into the dynamic-block set.

If a thread set diverges to two different dynamic blocks, SIMT-Step conservatively permits the most permissive behavior across the two sets: arbitrary interleavings. However, a specification may choose to prioritize one divergent path over another, with paths synchronizing at the merge block. It would not be difficult to extend SIMT-Step to support such constraints. Indeed, earlier versions of the CUDA documentation state that divergent paths are serialized (§4.1 [42]).

Our SIMT-Step model SCF (along with the stronger models) instantiates control flow instructions to be collective. We note that these semantics require preserving the effects of (control) dependencies at the programming language level, which has been controversial, e.g., in the C++ memory model. However, enforcing synchronization at control flow instructions does not completely prohibit

$$\begin{array}{c}
 I_{[\sigma.pc]} = branch(\ell) \\
 (db', D') = get_child(\sigma.db, \ell, \sigma.db.\mathcal{T}, D) \\
 \hline
 \langle \sigma, M, D \rangle \xrightarrow{t} \langle \sigma[pc := db'.b, db := db'], M, D' \rangle \\
 \text{(T-UBRANCH)} \\
 \\
 I_{[\sigma.pc]} = branchCond(c, \ell_t, \ell_f) \quad b = eval(c, \sigma.R) \\
 (db', D') = get_child(\sigma.db, (b?\ell_t : \ell_f), \sigma.db.\mathcal{T}, D) \\
 \hline
 \langle \sigma, M, D \rangle \xrightarrow{t} \langle \sigma[pc := db'.b, db := db'], M, D' \rangle \\
 \text{(T-CBRANCH)} \\
 \\
 \exists db \in D (Align(db, \Sigma) \wedge \\
 \forall i \in db.\mathcal{T} I_{[\Sigma(i).pc]} = branchCond(c, \ell_t, \ell_f) \wedge \\
 (db_t, D_t) = get_child(db, \ell_t, db.\mathcal{T}, D) \wedge \\
 (db_f, D_f) = get_child(db, \ell_f, db.\mathcal{T}, D) \wedge \\
 A_t = \{u \in db.\mathcal{T} \mid eval(c, \sigma_u.R)\}, A_f = db.\mathcal{T} \setminus A_t \wedge \\
 \forall u \in A_t : \Sigma'(u) = \Sigma(u)[pc := db_t.b, db := db_t] \wedge \\
 \forall v \in A_f : \Sigma'(v) = \Sigma(v)[pc := db_f.b, db := db_f]) \\
 \hline
 \langle \Sigma, M, D \rangle \xrightarrow{g} \langle \Sigma', M, D_t \cup D_f \rangle \\
 \text{(G-COLLECTIVE-CBRANCH)}
 \end{array}$$

Fig. 8. Rules for executing control flow instructions

638 optimizing away control dependencies; they would simply need to be replaced with subgroup
 639 synchronization instructions, which are expected to be lightweight on current hardware.

640 *Example.* We illustrate collective control flow in the two-thread example of Fig. 9. Each thread
 641 writes to opposite locations ($g[0]$ and $g[1]$) in sequence, optionally separated by a subgroup
 642 operation, e.g., `subgroupAdd(1)`. If branch instructions execute collectively, both threads must
 643 complete the first store before leaving the current basic block. They then enter the target block
 644 together and execute the second store, yielding the *synchronized* outcome. We discuss independent
 645 branch execution next.

647 4.3 Independent Branches

648 We now explore independent execution of control flow
 649 instructions. This first investigation focuses on the se-
 650 mantic design space under the following constraints:
 651 (1) we utilize an operational semantics; (2) we avoid
 652 introducing complex mechanisms such as speculative
 653 execution or symbolic values; and (3) subgroup opera-
 654 tions remain collective, since they must combine values
 655 across all threads in a dynamic block. Under those con-
 656 straints, we propose the weakest semantics we are able
 657 to derive. Given its complexity, we outline the formalism
 658 here and refer the reader to our TLA+ implementation
 659 for further details.

661 *Unknown Thread Sets.* If threads are allowed to branch
 662 in a non-collective manner, e.g., T-UBRANCH and T-
 663 CBRANCH from Fig. 8, then dynamic blocks may not
 664 know their complete set of executing threads at creation
 665 time. That is, a thread t could branch out of a dynamic
 666 block db and begin executing a new dynamic block db' ,
 667 while another thread $t' \in db.\mathcal{T}$ may eventually take the
 668 same branch. However, until t' takes the branch, it will
 669 not appear in $db'.\mathcal{T}$. To account for this, we add an *Unknown Thread Set* to dynamic blocks $db.U$
 670 to track the set of threads for which it is not yet known whether they will execute the block.

671 *Updating Align.* As mentioned earlier, we still execute subgroup operations collectively, e.g., as
 672 in G-COLLECTIVE-SUBGROUP from Fig. 7. Given a dynamic block db , recall that the function *Align*
 673 ensures that all threads in db arrive at the same instruction before allowing the collective operation.
 674 However, this constraint cannot be satisfied until db knows all of its threads, that is, the unknown
 675 thread set of db must be empty. This condition must therefore be added as a constraint to *Align*.
 676

677 *Implicit Synchronization.* We view the above rules as weakening synchronization as much as
 678 possible without relying on speculation or symbolic values. However, some implicit synchronization
 679 remains. We illustrate this with the example in Fig. 9. Under these relaxed semantics, including
 680 the subgroup operations changes the behavior of the program. This is surprising because each
 681 subgroup operation collectively involves only a single thread: the operation on line 7 is executed
 682 only by thread 0, with thread 1 executing the operation on line 10. Due to *Align*, threads must wait
 683 to execute a collective operation until any *potential* participating thread has branched to a path
 684 where it is guaranteed not to execute the collective. Thus, while the collective operations do not
 685 individually synchronize threads 0 and 1, they are indirectly synchronized through the branch on
 686

Initial: $g[0] = 0, g[1] = 0;$

```

1 int main() {
2   int index_a, index_b;
3   if(tid == 0) index_a = 0; index_b = 1;
4   if(tid == 1) index_a = 1; index_b = 0;
5   atomicStore(&g[index_a], 0);
6   if (tid == 0){
7     (opt.) subgroupAdd(1);
8     atomicStore(&g[index_b], 1);
9   } else{
10    (opt.) subgroupAdd(1);
11    atomicStore(&g[index_b], 1);
12  }
13 }
```

Allowed behaviors

Synchronized: $g[0] == 1 \ \&\& \ g[1] == 1 \ ;$
Relaxed: $g[0] == 1 \ \&\& \ g[1] == 1 \ ||$
 $g[0] == 0 \ \&\& \ g[1] == 1 \ ||$
 $g[0] == 1 \ \&\& \ g[1] == 0 \ ;$

Fig. 9. An example executed with two threads each in the same subgroup, with optionally included subgroup operations. The allowed behaviors explore the extent to which control flow synchronizes subgroups

line 6. As a result, even with independent branching, the synchronized behavior is guaranteed when subgroup operations are included. In our SIMT-Step models, SSO embodies this behavior. It only provides collective operations around subgroup operations, and implements all other relaxations. It is shaded lightly red in Tab. 1 to indicate its complex semantics.

4.4 Further Relaxations

Lastly, we explore additional relaxations through the use of speculation and symbolic semantics. Similar to the previous section, we find the semantic consequences of such relaxations to be highly complex and likely not useful to practitioners. Accordingly, we do not implement either of these models in TLA+. We call these models *indirect* and do not explore them further than discussing high-level tests. However, we note that they had serious consideration in the specification group discussions, as they allow incredible flexibility in implementations.

Speculative Subgroup Operations. The semantics we have explored so far require all threads in a dynamic block to be known and present before executing a subgroup operation. However, subgroup operations could, in principle, be speculatively executed, assuming that the current view of active threads remains valid. That is, the subgroup operation proceeds with the threads currently in $db.\mathcal{T}$ under the assumption that no unknown threads will later join. If this assumption is later violated, the execution is pruned.

We illustrate the counterintuitive behavior enabled by speculative subgroup operations in Fig. 10. This test is executed by two threads in the same subgroup with IDs 0 and 1. The test examines which threads are allowed to participate in the subgroup operation. Suppose thread 0 reads 0 from x , takes the branch, and enters the conditional block. At that point, the dynamic block sees only thread 0 as present, with thread 1 marked as unknown. Under speculation, the collective operation may proceed, assuming it will only involve thread 0. Thread 0 then stores 1 to x . Later, when thread 1 executes, it reads 1 from x and does not take the branch, and thus never enters the dynamic block containing the collective. In this way, the speculation is validated. This is counterintuitive because the speculation effectively justifies itself. While we did not implement it due to the difficulties around speculative execution, this specification describes the Spec SIMT-Step execution model.

This exact test spurred a lengthy discussion (with no resolution) within the specification groups, with an internal discussion board having 89 comments from nearly all major GPU vendors. Most participants agreed that the behavior was too relaxed, but the flexibility was desirable.

Symbolic Subgroup Operations. As discussed in Sec. 1, there has been ongoing debate about the extent to which compilers should be allowed to optimize by removing subgroup operations. While there are clear cases where such operations could be removed (e.g., when the argument is a constant), this is not always so obvious. It becomes difficult to constrain the types of analysis a compiler might perform to deduce the result of a computation, especially in the presence of advanced techniques, e.g., [47].

Semantically, this could be modeled by relaxing the requirement that subgroup operations execute collectively. Instead, each subgroup operation could execute independently and return a

Initial: `int x = 0;`

```

1 int cond = atomicLoad(x);
2 if (cond == 0) {
3   subgroupAdd(0);
4   atomicStore(&x, 1);
5 }
```

Allowed subgroup_all participants

Non-speculative: {0, 1}

Speculative: {0, 1} || {1} || {0}

Fig. 10. A test, executed with two threads in the same subgroup, showing the implications of collective speculation. Instead of the results asking about the final state, it instead asks about the allowed participants to the subgroup operation. If speculation is allowed, then any subset of threads can independently execute the collective and then make the block unreachable, satisfying the speculation.

687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735

736 symbolic value. This symbolic value would be constrained and concretized after all relevant threads
 737 have completed enough execution.

738 However, allowing symbolic execution introduces thin-
 739 air behaviors, just as it does in relaxed memory models. We
 740 illustrate this with the example in Fig. 11. The test involves
 741 two threads in the same subgroup. Thread 1 first loads an
 742 arbitrary value from $g[1]$. It then executes a subgroup broad-
 743 cast operation, which attempts to read a value b from thread
 744 0. Because thread 0 has not yet executed and initialized this
 745 value, the subgroup operation returns a symbolic value to
 746 thread 1, allowing it to proceed without synchronization.
 747 Thread 1 then stores this symbolic value to $g[0]$. When
 748 thread 0 eventually executes, it loads from $g[0]$, receiving
 749 the symbolic value that originated from thread 1. It then
 750 broadcasts this symbolic value, which is also stored at $g[1]$.

751 At this point, the symbolic value is completely unconstrained and may resolve to any value.

752 This constitutes a thin-air execution, despite all memory operations being sequentially consistent.
 753 Such behaviors are generally viewed as problematic and difficult to reason about [19]. Although we
 754 don't implement the semantics, the SIMT-Step model *Symb* could theoretically execute subgroup
 755 operations symbolically, following the above description. At this point, there are *no more* possible
 756 relaxations. There are no operations that are executing collectively or synchronously. Thus *Symb*
 757 represents the extreme end of independent subgroup execution.

758 To the best of our knowledge, thin-air behaviors are primarily a specification concern and have
 759 not been observed in practice [6]. As a sanity check, we implemented the test in Fig. 11 and evaluated
 760 it across our devices (detailed in Tab. 5). We observed no out-of-thin-air values. To avoid such
 761 issues (even at the specification level), one approach is to allow a compiler to replace a subgroup
 762 operation with a subgroup synchronization barrier, omitting the collective operation.

764 4.5 Release/Acquire Relaxed Memory Model Fragment

765 We outline how SIMT-Step can be extended to support a practical fragment of relaxed memory,
 766 where stores use release ordering and loads use acquire ordering from the C/C++ memory model [5].
 767 GPU languages such as CUDA [34], OpenCL [25, §3.3.7], and SPIR-V [23] support this release/acquire
 768 ordering. Our discussion follows prior operational work on relaxed memory [46], conceptually
 769 similar to vector-clock reasoning [35]. Even this fragment raises additional design questions in
 770 the SIMT setting, and a full treatment is left to future work. SIMT-Step provides a foundation for
 771 exploring these questions.

772
 773 *State.* We begin by defining a per-location *history*, which records the order in which values
 774 are written to shared memory. Formally, each address maps to a sequence of writes. Each SIMT-
 775 Step thread state σ_t is extended with a *viewfront* (vf), which records the thread's current view of
 776 memory by mapping each address to the index of the latest visible write in that address's history.
 777 The SIMT-Step system memory M is then replaced by an augmented history H , whose entries have
 778 the form (v, vf_w) , where v is the written value and vf_w is the viewfront of the thread at the time of
 779 the write. This stored viewfront can later be used by acquire reads to implement synchronization.

780 An *initial history* maps each address to a singleton history containing its initial value. The *initial*
 781 *viewfront* maps each address to this initial history position. In the initial SIMT-Step state, each
 782 thread-local viewfront $\sigma_t.vf$ is set to the initial viewfront, and the system memory is initialized to
 783 the augmented history obtained by pairing each initial value with the initial viewfront.

Initial: $g[0] = 0;$

```

1 int index_a, index_b;
2 if(tid == 0) index_a = 0; index_b = 1;
3 if(tid == 1) index_a = 1; index_b = 0;
4 int b = atomicLoad(&g[index_a]);
5 int a = subgroupBroadcast(b, 0);
6 atomicStore(&g[index_b], a);

```

Allowed behaviors

Symbolic (thin-air): $g[0]==Any$

Fig. 11. A test, executed with two threads in the same subgroup, showing the thin-air execution that arises when allowing non-collective (i.e., symbolic) execution of subgroup operations.

785 An important operation on viewfronts is *join*, which takes n viewfronts and returns a new
 786 viewfront vf_{result} . For each address a , $vf_{result}[a]$ is the maximum of the corresponding positions in
 787 the input viewfronts. Intuitively, joining two viewfronts synchronizes their views of memory.
 788

789 *Independent Memory Operations.* We begin with the simplest integration with SIMT-Step seman-
 790 tics, in which threads execute memory operations independently (e.g., T-LOAD and T-STORE). These
 791 semantics closely mirror the prior operational semantics for release/acquire memory [46] and only
 792 require updates to two rules:

- 793 • (T-STORE-RA): replaces (T-STORE); instead of updating M , append $(v, \sigma.vf)$ to $H[a]$.
- 794 • (T-LOAD-RA): replaces (T-LOAD). Instead of reading the value stored at address a in M , it
 795 may read any entry in $H[a]$ whose position is at or after $\sigma.vf[a]$. If the selected entry is
 796 $(v, \sigma'.vf)$, the load returns v and updates $\sigma.vf$ to $join(\sigma.vf, \sigma'.vf)$.

797 Intuitively, a release stores the issuing thread's viewfront in the history, and an acquire read
 798 synchronizes with that write by joining its viewfront with the one in the selected history entry.
 799

800 *Collective Operations.* Integrating relaxed memory with collective execution in SIMT-Step intro-
 801 duces additional design choices. In the independent case, each thread performs loads and stores
 802 using its own viewfront. In contrast, a collective step may advance multiple participating threads
 803 together, raising the question of how their viewfronts should interact during the operation. This
 804 applies both to subgroup operations (e.g., subgroupAdd) and, in SIMT-Step CM model. We outline
 805 three candidate designs that augment any (G-COLLECTIVE) rule:

- 806 • **Synchronize completely** Before executing the collective operation, all participating
 807 threads join their viewfronts. Intuitively, the collective acts as a synchronization point
 808 for memory views. This ensures that all participating threads observe a consistent view of
 809 memory: colliding stores attach the same viewfront to H , and colliding loads have the same
 810 set of available history entries. This design is simple, but may be too strong, especially for
 811 collectives that do not directly access memory, such as subgroup operations.
- 812 • **Synchronize on collision** Only participating threads whose accesses collide on the same
 813 address join their viewfronts. This preserves consistency for colliding memory operations,
 814 while avoiding unnecessary synchronization between threads accessing disjoint addresses.
 815 However, it introduces a more subtle semantics, since only a subset of threads participating
 816 in the collective may synchronize their viewfronts.
- 817 • **No persistent synchronization** Participating threads do not update their thread-local
 818 viewfronts during the collective. However, to ensure a consistent interpretation of collective
 819 memory effects, colliding accesses must still be evaluated using a joined viewfront for the
 820 accessed address. For example, this joined view can determine the valid load candidates and
 821 the viewfront stored with writes in H . Under this design, the collective does not permanently
 822 align the participating threads' views, but it may still require temporary joining to define a
 823 coherent memory effect. Variants of this kind are likely to admit highly counterintuitive
 824 behaviors, particularly because a thread may be unable to reason about its own memory
 825 view from the outcome of the collective operation.
 826

827 Overall, this simple relaxed memory model extension exposes a substantial semantic design space.
 828 Careful future work will be needed to develop executable formal models, design distinguishing
 829 litmus tests, and identify design points that avoid both counterintuitive behaviors and unnecessarily
 830 strong synchronization constraints. Our empirical results in Sec. 7 suggest that these questions
 831 may be difficult to study experimentally, as we did not observe relaxed memory behaviors within a
 832 subgroup on any tested platform when using a robust memory model testing framework.
 833

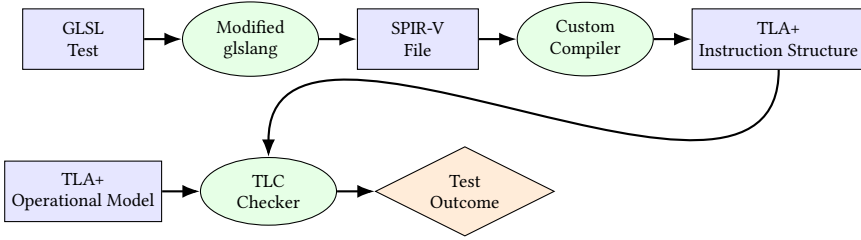


Fig. 12. End-to-end toolchain for executing GLSL test cases in the TLA+ model checker

5 Tool Suite and Implementation Details

We now describe the implementation of our SIMT-Step tool suite, which includes: (1) an executable semantics written in TLA+, covering the four of the models described in §1.2 under the SC memory model; (2) a curated set of idiomatic tests designed to distinguish between these models; and (3) a large, fuzzed test suite used to empirically evaluate how closely real GPU devices align with the SIMT-Step models. All material described in this section is submitted in the supplementary material.

5.1 Executable Semantics

We implement the semantics from §4 in TLA+, mirroring the specified states and transitions. Our TLA+ semantics implement over 100 SPIR-V instructions to support a GLSL frontend. For SIMT-Step, we model merge instructions and both unconditional and conditional branches. We also implement six subgroup operations, including subgroup barriers, which are treated as collective operations without computation.

To support SIMT-Step models, our implementation provides several configurations. Memory operations, branches, and labels can be configured as independent or collective. Additionally, any instruction can be marked for synchronous execution, which we use to implement the SM model.

An Accessible Front End. To ease testing, we provide two translation layers that enable writing test shaders in GLSL. Our execution pipeline is shown in Fig. 12. First, we developed a custom compiler that translates SPIR-V into TLA+, including full lexing and parsing. We extend SPIR-V and glslang (through custom `#pragma` directives) to include grid-configuration directives, synchronization annotations, and intra-shader assertions.

5.2 Generalizing Tests

Throughout, we have shown example tests, i.e., see Tab. 1. We now discuss how these test cases can be generalized in two ways: the types of memory accesses and the number of executing threads. This can provide more coverage when testing actual devices. First, we categorize four test patterns, which serve as distinguishing and documenting tests for the four direct models of Tab. 1.

- **CM**: This test corresponds to Fig. 3 and tests whether memory loads are executed collectively.
- **SM**: This test corresponds to Fig. 2 without the subgroup operation and tests whether memory operations execute synchronously.
- **SCF**: This test corresponds to Fig. 9 without the subgroup operations and tests whether threads have synchronous control flow.
- **SSO**: This test corresponds to Fig. 9 with the subgroup operations included. It tests if subgroup operations synchronize with associated control flow.

Table 4. Summary of executing tests under SIMT-Step semantics in TLA+. Each model is tested in two configurations: one allowing relaxed behavior (to detect a witness) and one assuming strong semantics (to verify correctness). The configuration is given as (x, y) , where x is the number of subgroups and y is the subgroup size. We report lines of code (LoC) for both GLSL and SPIR-V as pairs, which correspond to the test pair, along with execution metrics. The first model has no relaxed variant.

Test Pair	Model	Config	GLSL LoC	SPIR-V LoC	Relaxed Witness		Strong Verif.	
					States	Time (s)	States	Time (s)
(NA, Fig. 3)	CM	$\langle NA, (2, 2) \rangle$	(NA, 21)	(NA, 74)	NA	NA	5065	1
(Fig. 3, Fig. 2)	SM	$\langle (2, 2), (1, 2) \rangle$	(21, 24)	(74, 93)	3796	1	203	1
(Fig. 2, Fig. 9 \circ sg)	SCF	$\langle (1, 2), (1, 2) \rangle$	(24, 28)	(93, 104)	211	1	227	1
(Fig. 9 \circ sg, Fig. 9 \oplus sg)	SSO	$\langle (1, 2), (1, 2) \rangle$	(28, 30)	(104, 109)	308	1	233	1

These tests follow a common pattern: they construct a racy (scheduler-dependent) program that, under fully independent execution, admits all interleavings, then examine which behaviors are allowed (or observed) to infer synchronization. The exception is the CM test, which checks atomicity, e.g., whether all participating threads observe the same value.

The tests presented so far were instantiated with two stores per thread. However, race conditions can also occur between write-read and read-write pairs (although not read-read). Thus, the above test patterns can be generalized across write-write (W-W), read-write (R-W), and write-read (W-R) pairs. This gives us 10 possible base tests. While SIMT-Step generally doesn't distinguish between memory access types (loads and stores), these different test instantiations could provide different coverage when executing across real systems.

To support varying subgroup sizes and the large parallelism of GPUs, we generalize each test to run with an arbitrary number of threads (and an arbitrary subgroup size greater than one). Instead of using fixed addresses (e.g., x and y), the tests now use a buffer. To scale to arbitrary thread and subgroup sizes, tests use a buffer indexed by thread IDs. Each thread accesses its own slot, then the next slot in the subgroup (wrapping at the end), distributing race behavior across all threads.

Fuzzing Tests. To more thoroughly evaluate whether systems relax subgroup synchronization, we apply compiler fuzzing techniques to our core suite of 10 tests, generating 100k variants (10K each). We use GraphicsFuzz [50], which applies semantics-preserving transformations to the input code. That is, given a test such as the one in Fig. 3, the fuzzer inserts additional instructions that leave the original behavior intact. While GraphicsFuzz is unaware of subgroup semantics, manual inspection confirms that it preserves the actual operations (memory and subgroup operations) and the control-flow relationships between them, which are the critical components of our test.

6 Executable Semantics

We run our simple idiomatic tests, configured with the smallest number of threads, through our TLA+ semantics. This consists of the 10 tests described in Sec. 5.2, written in GLSL and passed through our pipeline. We run these tests on a modest device: a mini PC running Linux kernel version 6.15.2-arch1-1, with 64 GB of RAM, a 14-core, 20-thread i9-13900H CPU (5.40 GHz), and execute TLC in parallel using all 20 threads. Our results are shown in Tab. 4. Each test behaves as expected under each implemented model (i.e., the direct models from Sec. 1.2). Note that the smallest configuration for the Collective Mem test is $(2,2)$ as opposed to $(1,2)$. This is larger because it requires two subgroups to potentially disrupt atomicity violations in collective memory operations.

All tests run in under one second. This is likely because of how small the tests are, the small number of threads, and the heuristics to only interleave at shared state changing actions. We start

to see the state space explosion in the CM test (with over $10\times$ more states), as it requires four total threads instead of just two. Despite this, our tests still execute in under one second.

7 Empirical Investigation

Tests are written in GLSL (via GraphicsFuzz) and executed using the Amber framework [15]. GLSL is well supported and provides a mature subgroup API, contrary to other portable platforms like WebGPU, where subgroup support is still evolving [53]. Our tests require the `VK_KHR_shader_subgroup` and `VK_KHR_shader_maximal_reconvergence` extensions, and SPIR-V version 1.3+.

We evaluate subgroup behavior on a diverse set of real-world GPUs by running our fuzzed test suite across ten devices from eight vendors, covering mobile, integrated, and discrete hardware summarized in Tab. 5 with additional details provided in supplementary material. We exclude Imagination GPUs due to trivial subgroup size (1) on our available devices, though newer devices (e.g., Pixel 10) report a meaningful subgroup size of 128; we hope to test these devices soon. Although Apple devices do not natively support GLSL, we were able to include an Apple M1 GPU by using the Asahi Linux Vulkan driver (which only supports the older M1 and M2 GPUs at this time) [4], allowing us to extend coverage to a widely deployed GPU platform that is typically excluded from Vulkan-based studies. Given that GLSL (and SPIR-V) do not support sequentially consistent atomics, our tests use the strongest available operations: release for stores and acquire for loads. Under this configuration, the tests remain well-defined, although technically more permissive than intended (see Sec. 7.2). Each test runs with the largest portable supported grid: 65K workgroups, each with 128 threads [28]. Total runtime exceeds 700 hours, with mobile GPUs generally slower than discrete GPUs.

As with any large empirical study, some generated tests failed to execute. A substantial fraction arose from invalid programs produced by GraphicsFuzz after introducing atomic operations, e.g., atomics targeting thread-local memory, which are rejected by GLSL. We filtered out these invalid tests prior to analysis; they accounted for roughly 14% and do not correspond to valid programs. In addition, less than 1% of tests per device failed due to other compiler errors that we were unable to diagnose, often involving large shaders that could not be reduced. Finally, on the Samsung device, fewer than 1% of cases encountered runtime errors (e.g., out of memory) that we could not reliably reproduce. Only successfully executed tests were included in our results.

7.1 Empirical Observations

Our results are presented in Fig. 13, with rows representing GPUs and columns representing test patterns. Tests are grouped by pattern (as described in Sec. 5.2) and memory access sequence (**W** for write, **R** for read). Each cell reports the number of relaxed outcomes: green indicates none, while red indicates one or more (darker denotes higher frequency). Results aggregate 10K fuzzed variants per base test, each executed once.

Collective Memory Observations. Non-collective memory behavior was widely observed across many tested devices. The exceptions were NVIDIA and Apple, which showed no such behaviors in our experiments, and the Samsung device, which exhibited only non-collective memory in one

GPU	Vendor	SG Size
Radeon RX 7900	AMD	64
RTX 4070 Super	NVIDIA	32
Tegra Orin	NVIDIA	32
Iris Xe Graphics	Intel	16
Mali-G710 MP7	ARM	16
Xclipse 940	Samsung	32
Adreno 740	Qualcomm	64
Adreno 730	Qualcomm	64
Adreno X1-85	Qualcomm	64
M1	Apple	32

Table 5. Devices used in our empirical study. Some GPUs, e.g., from Intel, have dynamic subgroup size which can vary across different kernels. The subgroup (SG) size was obtained by querying `gl_SubgroupSize` with `varying_subgroup_size` enabled [31].

932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980

981 test. These results suggest that the strongest SIMT-Step model, Collective Memory (CM), is too
982 strict. This is notable because collective memory behavior has been assumed by several classic GPU
983 optimization techniques, such as leader election [37]. Thus, these idioms are unlikely to be reliable
984 across diverse GPUs; luckily modern subgroup APIs provide safer and more portable alternatives.

985 7.2 Investigating Rare Observations and Threats to Validity

986 Next, we observe that non-synchronous behaviors appear only on the Qualcomm and Samsung
987 devices (which contains a Qualcomm GPU) and are extremely rare. We devoted substantial effort
988 to understanding whether these observations reflect intended relaxations or unintended anomalies,
989 such as compiler or driver bugs. Rare outcomes can correspond to legitimate relaxed behaviors [1],
990 but GPU software stacks are also known to exhibit framework and compiler issues [11], especially
991 in under-explored areas (e.g., subgroup execution). We summarize our investigation below.

992 *Test Case Reduction.* We attempted to reduce the corresponding fuzzed tests using both the
993 GraphicsFuzz reducer and manual reduction, but succeeded only for one test (SM R-W) on the
994 Samsung device (included in the supplementary material). The reduced source shows no obvious
995 pattern suggesting an aggressive compiler transformation, and we were unable to inspect the
996 generated assembly (as is typical for GPUs). We shared this test with Qualcomm partners, though
997 they could not comment at this time. The remaining tests resisted reduction: simplifying the kernel
998 caused the non-synchronous behavior to disappear.

1000 *Hardware Relaxed Memory.* Another explanation is that some observations arise from relaxed
1001 GPU memory behavior [1], though prior work focuses on inter-workgroup interactions. As noted
1002 earlier, due to framework limitations, our tests use release/acquire (RA) atomics. Under RA, the
1003 W-R and W-W variants correspond to store buffering and 2+2W, respectively [5], where weak
1004 outcomes are permitted, while the R-W variants correspond to load buffering, whose weak outcome
1005 is disallowed under RA.

1006 To test whether relaxed memory behavior explains our observations, we adapted the GPUHarbor
1007 memory model testing framework [32] to focus on store buffering and 2+2W within a single
1008 subgroup. We modified the pairing logic so threads communicated within the same subgroup,
1009 mirroring our tests. Across Qualcomm and Samsung devices, we observed no relaxed outcomes.
1010 Thus, relaxed memory behavior does not explain the observed non-synchronous subgroup behavior.

1011 *Differential Testing.* To further investigate, we performed differential testing on the Adreno 740
1012 using an alternative driver stack (Mesa) [38]. Notably, all 14 tests that showed non-synchronous
1013 behavior on the native stack instead exhibited synchronous behavior under Mesa.

1014 *Putting it Together.* Despite this investigation, we cannot determine the root cause of these rare
1015 non-synchronous observations. They may arise from legal reorderings under relaxed memory (e.g.,
1016 W-W and W-R under RA), from complex hardware or software stack behaviors, or from unintended
1017 anomalies (e.g., framework bugs).

1018 This uncertainty affects interpretation: The programming model must permit the weakest
1019 observed behavior. Thus, our results suggest that subgroup (and memory) semantics may be
1020 weak: empirically, all tested devices are consistent with the weakest direct SIMT-Step model (SSO)
1021 under RA semantics (the only all-green column in Fig. 13). However, if these rare observations are
1022 unintended anomalies, current GPUs may instead align with a stronger model closer to sequential
1023 consistency under SM (though CM still appears too strong). This highlights the difficulty of inferring
1024 subgroup semantics from empirical data and the need for clearer guidance from vendors and
1025 specification groups.

1030	Radeon 7900										8968
1031	RTX 4070										
1032	Tegra Orin										
1033	Iris XE										8981
1034	Mali-G710										7769
1035	Xclipse 940	7	1	2	4	2		5	1		1
1036	Adreno 740	9	1		2		1	1			9048
1037	Adreno 730	1		4	1	14	9	12	8		9037
1038	Adreno X1-85					1					9199
1039	M1										
1040		R-W	W-R	W-W	R-W	W-R	W-W	R-W	W-R	W-W	W-R
1041		SSO			SCF			SM			CM.
1042		Relaxation Count									

Fig. 13. Observed relaxed behaviors: Green cells show no relaxed behaviors

8 Related Work

To our knowledge, there is little prior work that focuses on specifying the semantics of subgroup (or warp) behavior, particularly regarding degrees of independent execution. For example, [48] addresses producer-consumer synchronization across subgroups, but their model assumes lockstep execution and therefore does not capture independent thread progress within a subgroup. We detailed prior work on GPU verification in this area in Sec. 1 and provide a comprehensive history in supplementary material. Here we note a few hardware and software (e.g., compiler transformations) optimizations around SIMT execution that relax synchronous execution.

Subgroups correspond to sets of GPU threads executing on SIMT hardware, often with strong synchronous behavior. This tight hardware coupling historically informed the synchronous semantics exploited by programmers. Since synchronization incurs cost if not supported natively, the SIMT implementation directly constrains feasible programming models. Several works have proposed new hardware SIMT execution. For example, [14] introduced dynamic warp formation, regrouping threads based on control flow to improve utilization under divergence. Another work, [12] tackled synchronization bottlenecks by proposing BOWS, a scheduler that deprioritizes spinning threads within warps, and DDOS, a hardware mechanism for detecting such spin loops.

On the software side, [9] proposed *speculative reconvergence*, a user-guided compiler annotation that delays the execution of common code regions until enough threads have arrived, improving SIMT efficiency. Such techniques may influence subgroup semantics, altering fairness guarantees or allowing subgroup composition to change during execution. As a result, specifications should consider the flexibility to accommodate evolving, and potentially radical, SIMT hardware designs.

9 Conclusion

This paper introduced SIMT-Step, a flexible operational semantics for modeling subgroup execution in GPU programming. SIMT-Step utilizes a new semantic object, the dynamic block, which tracks convergence and divergence across subgroups. Different models can then be instantiated by specifying whether instructions execute collectively, synchronously, or independently. We instantiated several SIMT-Step models and implemented them in TLA+, validating their behavior with a suite of idiomatic tests. We also evaluated thousands of fuzzed GLSL tests across a wide range of GPUs, finding that most tested platforms exhibited behavior consistent with strong subgroup synchronization, while a small number of rare relaxed observations remain inconclusive. Together, these contributions provide a foundation and tooling support for exploring subgroup semantics.

References

- [1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 577–591. doi:10.1145/2694344.2694391
- [2] Jade Alglave and Luc Maranget. 2015. Towards a formalisation of the HSA memory model in the cat language.
- [3] Apple Inc. 2024. Metal Developer Resources. <https://developer.apple.com/metal/resources/> Accessed: 2025-07-06.
- [4] Asahi Linux Project. 2024. Asahi Linux. <https://asahilinux.org/>. Accessed: 2026-03-25.
- [5] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. *SIGPLAN Not.* 46, 1 (Jan. 2011), 55–66. doi:10.1145/1925844.1926394
- [6] Mark Batty and Peter Sewell. 2014. The Thin-air Problem. <https://www.cl.cam.ac.uk/~pes20/cpp/notes42.html> Working note.
- [7] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 113–132. doi:10.1145/2384616.2384625
- [8] NVIDIA Corporation. 2024. CUB 1.16.0: Warp-, Block-, and Device-Wide Collective Primitives. *NVIDIA Research* (online documentation). <https://nvlabs.github.io/cub/> See “Warp-Wide Collective Primitives” section.
- [9] Sana Damani, Daniel R. Johnson, Mark Stephenson, Stephen W. Keckler, Eddie Yan, Michael McKeown, and Olivier Giroux. 2020. Speculative re-convergence for improved SIMT efficiency. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20)*. Association for Computing Machinery, New York, NY, USA, 121–132. doi:10.1145/3368826.3377911
- [10] Wei Ding, Diana Guttman, and Mahmut Kandemir. 2014. Compiler Support for Optimizing Memory Bank-Level Parallelism. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, USA, 571–582. doi:10.1109/MICRO.2014.34
- [11] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. doi:10.1145/3133917
- [12] Ahmed ElTantawy and Tor M. Aamodt. 2018. Warp Scheduling for Fine-Grained Synchronization. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 375–388. doi:10.1109/HPCA.2018.00040
- [13] Naznin Fauzia, Louis-Noël Pouchet, and P. Sadayappan. 2015. Characterizing and enhancing global memory data coalescing on GPUs. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, USA, 12–22.
- [14] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE Computer Society, 407–420. doi:10.1109/MICRO.2007.30
- [15] Google. 2025. Amber. <https://github.com/google/amber>. Accessed: 2025-04-24.
- [16] The Khronos Group. 2025. SPIR-V Specification, Section “Dynamic Instance”. <https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html#DynamicInstance>. Version 1.6, Revision 6.
- [17] Thomas Haas, Roland Meyer, Hernán Ponce de León, and Andrés Lomeli. 2026. Recurrence Sets for Proving Fair Non-termination under Axiomatic Memory Consistency Models. *Proc. ACM Program. Lang.* (2026).
- [18] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free memory models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 427–440. doi:10.1145/2541940.2541981
- [19] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, POPL, Article 54 (Jan. 2022), 30 pages. doi:10.1145/3498716
- [20] Amir Ashraf Kamil and Katherine A. Yelick. 2006. *Concurrency Analysis for Parallel Programs with Textually Aligned Barriers*. Technical Report UCB/ECS-2006-41. EECS Department, University of California, Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-41.html>
- [21] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 175–189. doi:10.1145/3009837.3009850
- [22] Khronos Group. 2017. OpenGL Shading Language, Version 4.50. <https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.50.pdf> Revision 7, May 9, 2017.

- 1128 [23] Khronos Group. 2018. Comparing the Vulkan SPIR-V Memory Model to C++'s. [https://www.khronos.org/blog/
1129 comparing-the-vulkan-spir-v-memory-model-to-cs](https://www.khronos.org/blog/comparing-the-vulkan-spir-v-memory-model-to-cs) Blog post.
- 1130 [24] Khronos Group. 2024. SPV_KHR_maximal_reconvergence Extension Specification. [https://github.khronos.org/SPIRV-
1131 Registry/extensions/KHR/SPV_KHR_maximal_reconvergence.html](https://github.khronos.org/SPIRV-Registry/extensions/KHR/SPV_KHR_maximal_reconvergence.html) Revision 2, Last Modified: 2024-04-18.
- 1132 [25] Khronos Group. 2025. OpenCL 3.0 API Specification. [https://registry.khronos.org/OpenCL/specs/3.0-unified/html/
1133 OpenCL_API.html](https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html) Version 3.0.18, Last accessed: May 12, 2025.
- 1134 [26] Khronos Group. 2025. OpenCL™ SPIR-V Environment Specification. [https://registry.khronos.org/OpenCL/specs/3.0-
1135 unified/html/OpenCL_Env.html](https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_Env.html) Version 3.0.18, April 3, 2025.
- 1136 [27] Khronos Group. 2025. Vulkan API Specification. <https://docs.vulkan.org/spec/latest/index.html> Version 1.3, Last
1137 accessed: May 12, 2025.
- 1138 [28] Khronos Group. 2025. Vulkan® 1.3.283 - Limits. <https://docs.vulkan.org/spec/latest/chapters/limits.html>. Accessed:
1139 2025-07-10.
- 1140 [29] Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. 2020. Foundations of empirical memory consistency testing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 226 (Nov. 2020), 29 pages. doi:10.1145/3428294
- 1141 [30] Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. 2023. Taking Back Control in an Intermediate Representation for GPU Computing. *Proc. ACM Program. Lang.* 7, POPL, Article 60 (Jan. 2023), 30 pages. doi:10.1145/3571253
- 1142 [31] Raph Levien. 2020. Prefix sum on Vulkan. <https://raphlinus.github.io/gpu/2020/04/30/prefix-sum.html>. Blog post.
- 1143 [32] Reese Levine, Mingun Cho, Devon McKee, Andrew Quinn, and Tyler Sorensen. 2023. GPUHarbor: Testing GPU Memory Consistency at Large (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 779–791. doi:10.1145/3597926.3598095
- 1144 [33] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: concolic verification and test generation for GPUs. *SIGPLAN Not.* 47, 8 (Feb. 2012), 215–224. doi:10.1145/2370036.2145844
- 1145 [34] Daniel Lustig, Sameer Sahasrabudde, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 257–270. doi:10.1145/3297858.3304043
- 1146 [35] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.
- 1147 [36] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/2145816.2145832
- 1148 [37] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.* 1, 2, Article 14 (Feb. 2015), 30 pages. doi:10.1145/2717511
- 1149 [38] Mesa developers. 2024. Mesa 3D Graphics Library. <https://www.mesa3d.org/>. Accessed: 2026-03-25.
- 1150 [39] Microsoft. 2025. DirectX-Specs. <https://microsoft.github.io/DirectX-Specs/>. Accessed: 2025-07-06.
- 1151 [40] Microsoft Corporation. 2019. Reference for HLSL. [https://learn.microsoft.com/en-us/windows/win32/direct3dhls/dx-
1152 graphics-hlsl-reference](https://learn.microsoft.com/en-us/windows/win32/direct3dhls/dx-graphics-hlsl-reference) Accessed: 2025-05-12.
- 1153 [41] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An operational semantics for C/C++11 concurrency. *SIGPLAN Not.* 51, 10 (Oct. 2016), 111–128. doi:10.1145/3022671.2983997
- 1154 [42] NVIDIA Corporation. 2016. *CUDA C Programming Guide, Version 8.0*. NVIDIA Corporation. [https://docs.nvidia.com/
1155 cuda/archive/8.0/pdf/CUDA_C_Programming_Guide.pdf](https://docs.nvidia.com/cuda/archive/8.0/pdf/CUDA_C_Programming_Guide.pdf) Section §4.1, p. 69.
- 1156 [43] NVIDIA Corporation. 2024. *CUDA C Programming Guide: SIMT Architecture*. NVIDIA Corporation. [https://docs.
1157 nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture](https://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture) Accessed: 2025-05-12.
- 1158 [44] NVIDIA Corporation. 2025. *CUDA C++ Programming Guide* (release 12.9 ed.). NVIDIA Corporation, Santa Clara, CA. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf Section 7.1, p. 140.
- 1159 [45] NVIDIA Corporation. 2025. *CUDA Volta Tuning Guide* (v12.9 ed.). NVIDIA, Santa Clara, CA. [https://docs.nvidia.com/
1160 cuda/volta-tuning-guide/index.html](https://docs.nvidia.com/cuda/volta-tuning-guide/index.html) Last updated May 31, 2025.
- 1161 [46] Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. 2016. Operational Aspects of C/C++ Concurrency. arXiv:1606.01400 [cs.PL] <https://arxiv.org/abs/1606.01400>
- 1162 [47] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2018. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422 [cs.PL] <https://arxiv.org/abs/1711.04422>
- 1163 [48] Rahul Sharma, Michael Bauer, and Alex Aiken. 2015. Verification of producer-consumer synchronization in GPU programs. *SIGPLAN Not.* 50, 6 (June 2015), 88–98. doi:10.1145/2813885.2737962
- 1164 [49] Tyler Sorensen, Lucas F. Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F. Donaldson. 2021. Specifying and testing GPU workgroup progress models. *Proc. ACM Program. Lang.* 5, OOPSLA, 1175
- 1176

- 1177 Article 131 (Oct. 2021), 30 pages. doi:10.1145/3485508
- 1178 [50] The GraphicsFuzz Authors. 2025. GraphicsFuzz testing framework. <https://github.com/google/graphicsfuzz>. Accessed
- 1179 June 30, 2025.
- 1180 [51] The Khronos Group. 2018. *Vulkan has just become the world's first graphics API with a formal memory model. So what*
- 1181 *is a memory model and why should I care?* The Khronos Group. [https://www.khronos.org/blog/vulkan-has-just-](https://www.khronos.org/blog/vulkan-has-just-become-the-worlds-first-graphics-api-with-a-formal-memory-model.-so-what-is-a-memory-model-and-why-should-i-care)
- 1182 [become-the-worlds-first-graphics-api-with-a-formal-memory-model.-so-what-is-a-memory-model-and-why-](https://www.khronos.org/blog/vulkan-has-just-become-the-worlds-first-graphics-api-with-a-formal-memory-model.-so-what-is-a-memory-model-and-why-should-i-care)
- 1183 [should-i-care](https://www.khronos.org/blog/vulkan-has-just-become-the-worlds-first-graphics-api-with-a-formal-memory-model.-so-what-is-a-memory-model-and-why-should-i-care) Accessed 6 July 2025.
- 1184 [52] Haining Tong, Natalia Gavrilenko, Hernan Ponce de Leon, and Keijo Heljanko. 2025. Towards Unified Analysis of
- 1185 GPU Consistency. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming*
- 1186 *Languages and Operating Systems, Volume 4 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA,
- 1187 329–344. doi:10.1145/3622781.3674174
- 1188 [53] W3C GPU for the Web Working Group. 2024. WebGPU Shading Language (WGSL) Specification — Subgroups.
- 1189 <https://www.w3.org/TR/WGSL/#subgroups>.
- 1190 [54] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU compiler for memory optimization and
- 1191 parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and*
- 1192 *Implementation (PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 86–97. doi:10.1145/1806596.
- 1193 1806606
- 1194
- 1195
- 1196
- 1197
- 1198
- 1199
- 1200
- 1201
- 1202
- 1203
- 1204
- 1205
- 1206
- 1207
- 1208
- 1209
- 1210
- 1211
- 1212
- 1213
- 1214
- 1215
- 1216
- 1217
- 1218
- 1219
- 1220
- 1221
- 1222
- 1223
- 1224
- 1225