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

ANONYMOUS AUTHOR(S)

GPU hardware typically implements a SIMT execution model, where small groups of threads, called subgroups (or warps in CUDA), synchronously execute instructions. GPU programming languages expose this concept through subgroup-level APIs that enable high-performance optimizations. 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 eventually executes synchronously. As a result, no existing GPU programming language provides a rigorous semantic model for subgroup execution.

In this work, we present SIMT-Step, a formal and flexible operational semantics for GPU subgroup execution. At its core is a new semantic object, dynamic blocks, which generalizes dynamic instruction instances to basic blocks, enabling precise specification of converged subgroup execution. SIMT-Step then provides flexibility around 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 and document them, highlighting surprising and counter-intuitive behavior that arises under more relaxed variants. We implement SIMT-Step in TLA+ and validate the behavior of the idiomatic tests, all of which can be verified in under one second. Finally, to investigate how closely SIMT-Step models real-world GPU behavior, we conduct a large-scale fuzzing campaign, executing for over 700 hours spanning nine GPUs and seven vendors; our results show that, despite the hesitancy to provide strict guarantees in official specifications, most GPUs exhibit strongly synchronous subgroup execution. Combined, these contributions provide both a theoretical foundation and practical tools for reasoning about subgroup semantics in GPU programming languages.

1 Introduction

GPUs have become essential to modern computing, driving workloads in machine learning, graphics, and scientific simulation. Their impressive efficiency stems from a massively parallel execution model, but this efficiency 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. Programmers must reason about their code, compiler writers must understand which transformations are sound, and 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.

Over the years, components of the GPU programming model have been examined and formalized. Much attention has focused on the memory consistency model, especially given the complexity introduced by the GPU memory hierarchy. These efforts have led to formalizations of existing language specifications [1, 51], proposals for new models [15], and empirical investigations [26]. Notably, both PTX (NVIDIA's IR for CUDA) and SPIR-V (the shading language for Vulkan) now include formal memory model specifications [32, 50]. Other works have formalized control flow properties [27] and forward progress guarantees [2, 48].

Still, many aspects of the GPU programming model remain underspecified. One increasingly critical example is subgroup behavior. Subgroups (or warps in CUDA), are small groups If a dynamic blockof threads that execute together in a SIMT (Single Instruction, Multiple Threads) manner (§7.1 [44]). Due to their close proximity in hardware, subgroup threads can cooperate with extremely low overhead; these capabilities are exposed to programmers through subgroup-level APIs in modern GPU languages, and are widely used in performance-critical code [6].

Formally specifying subgroup execution is challenging. Although subgroup threads nominally execute together under the SIMT model, the degree of provided implicit synchronization is unclear and sparsely documented. Compiler transformations can further weaken or obscure hardware-level synchronous behavior. Moreover, subgroup operations do not require all threads to participate in collective operations. Precisely specifying which threads are required to participate, especially when synchronization is relaxed, is non-trivial. As a result, subgroup semantics remain a critical gap in our understanding of GPU execution models.

1.1 Example: Lockstep and Collective Synchronization

Initial: `int *x = 0, *y = 0;`

```
 1  int *addr1, *addr2;
 2  int st1, st2;
 3  if (tid == 0) {
 4    addr1 = x; addr2 = y;
 5    st1 = 1; st2 = 2;
 6  }
 7  if (tid == 1) {
 8    addr1 = y; addr2 = x;
 9    st1 = 1; st2 = 2;
10  }
11  atomicStore(addr1, st1);
12  // (opt.) int a=subgroup_all(0);
13  atomicStore(addr2, st2);
```

Allowed behaviors

```
Lockstep:     *x == 2 && *y == 2 ;
Interleaving: *x == 2 && *y == 2 ||
              *x == 1 && *y == 2 ||
              *x == 2 && *y == 1 ;
```

Fig. 1. 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.

Consider the GPU program in Fig. 1, executed by two threads in the same subgroup, each with a unique thread ID (`tid`). Each thread performs some local initializations before reconverging at line 11, where both execute two memory stores (across shared locations `x` and `y`), potentially separated by a subgroup operation. We assume a sequentially consistent memory model, as indicated by the atomic functions.

Lockstep Behavior. First, consider the program without the subgroup operation (line 12). Under lockstep execution, threads 0 and 1 would execute line 11 together, thread 0 and thread 1 storing the value 1 to `x`, and `y`, respectively. After completion, they advance to line 13, where thread 0 and 1 store the value 2 to `y` and `x`, respectively. This tightly synchronized execution permits only the one *Lockstep* behavior: `*x == *y == 2`.

Historically, subgroup lockstep execution was often assumed; however, GPU specifications have shifted toward allowing *independent* intra-subgroup behavior, where threads do not have to execute synchronously. Thus, interleaving behaviors would be permitted. Specifically, thread 0 could run first, storing 1 to `x` and then 2 to `y`, followed by thread 1 storing 1 to `y` and then 2 to `x`, resulting in `*x == 2 && *y == 1`. Reversing the thread order yields another valid outcome: `*x == 1 && *y == 2`.

Even when hardware implements lockstep execution, compilers would still be allowed to utilize interleaving behaviors in transformations. For example, given a system with lockstep hardware, but an interleaving specification, a compiler could transform the program by rewriting line 5 as `int st1 = 2, st2 = 1;`. This would cause thread 0 to store 2 to `y` and thread 1 to store 1 to `x` in the final store (line 13), producing an interleaved behavior. While this transformation may seem contrived, it reflects a broader class of compiler optimizations that adjusts arguments to memory accesses. Such transformations can greatly impact performance, e.g., by avoiding bank conflicts [7] or enabling memory coalescing [10, 53]. As a result, platform implementers may prefer more permissive specifications that allow these kinds of optimization, even on lockstep hardware. However, completely removing synchronous execution yields complex semantics and surprising behaviors (see Sec. 4.4 and 4.5), especially when determining which threads will participate in a subgroup operation.

Subgroup Operation Synchronization. Now consider the test with the line 12 subgroup operation included. This operation performs a collective reduction (`and`) across subgroup threads and their arguments, and stores the result in `a`. This test raises the question: does a subgroup operation imply subgroup synchronization? That is, if the specification permits interleaved behaviors for memory operations, does a subgroup operation synchronize threads? If so, then including the subgroup operation would guarantee the lockstep behavior. In this example, the reduction is trivial as each thread provides a constant (0), and thus, an optimizing compiler could be tempted to replace the operation with 0. Is removing the subgroup operation entirely allowed? If so, and under interleaving semantics, then one could imagine a sequence of transformations: first, removing the subgroup operation; and next, inducing an interleaving behavior. Thus, even a subgroup operation may not be sufficient to provide synchronous behavior. Alternatively, transformations might be constrained to maintain some type of synchronization, e.g., by transforming collective reductions to simpler barriers. However, these rules remain unclear.

1.2 SIMT-Step Execution: A Flexible Operational Semantics For Subgroups

In this paper, we introduce *SIMT-Step Execution*, a flexible operational semantics for specifying subgroup behavior in GPU programming languages. SIMT-Step can be instantiated in multiple ways, for example, to account for the range of behaviors discussed in Sec. 1.1 and to explore their broader semantic consequences. SIMT-Step was developed in discussion with official GPU specification groups; and while there has not yet been any official adoption, SIMT-Step is designed to accommodate a spectrum of behaviors and optimizations proposed during these discussions. Our investigations have already informed ongoing conversations, and we intend for this work to continue to influence the evolution of GPU language specification.

Dynamic Blocks. Subgroup operations are unusual because they are both *collective*, combining inputs across threads, and *non-uniform*, i.e., not all threads are required to participate. That is, subgroup operations are allowed in *divergent* control flow, i.e., control flow that has different targets across the subgroup; in this case only the threads that execute that path will participate.

SIMT-T first provides the semantic foundations to specify the set of participating threads in subgroup operations, even under divergent control flow. To do this, we extend the notion of a dynamic instruction instance, (sometimes called an event or action) to basic blocks, defining the *dynamic block* (Sec. 3). An execution can then be described as the a trace of dynamic instructions, but also as a graph of dynamic blocks. A single dynamic block can be executed by many threads. All threads begin execution in the dynamic entry block; threads that take the same branch continue together into a shared dynamic block. This structure allows us to distinguish between different dynamic instances of the same static basic block across loop iterations or call sites. Each dynamic block can query its threads, i.e., the set of threads that executed it. Unless otherwise noted, the threads in a dynamic block belong to the same subgroup; although the concept can generalize simply by partitioning the threads according to any level in the GPU hierarchy. If a dynamic block contains a subgroup operation, then its threads threads will participate in that operation.

Dynamic blocks can be used to express the *textual alignment* property used to formalize (GPU) barriers in prior work [17, 29]. However, subgroup operations may be executed non-uniformly and therefore require a more expressive semantic construct, which are provided by dynamic blocks.

Flexible Synchronization. SIMT-Step contains flexibility along two semantic dimensions:

(1) **Collective Operations**: The operations execute collectively (atomically) across, and therefore synchronize, participating threads. These include subgroup operations, but may also contain memory accesses and control flow, depending on the intended semantics.
(2) **Implicit Synchronization**: Any implicit synchronization across subgroup threads.

SIMT-Step Instantiations. By specifying different sets of collective operations and implicit synchronization properties (Sec. 4), we define several instantiations of SIMT-Step that could be candidates for inclusion in language specifications. These instantiations can be ordered from stronger, more collective operations and implicit synchronization, to weaker, fewer collective operations and reduced synchronization. The various models, short descriptions, and documenting tests are summarized in Tab. 1, and their definitions formalized throughout the paper. Many of our tools and

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 violating the converged basic block property (Sec. 4.2), and the darker shaded SC and SymC 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. 2)
Synchronous Memory (`SM`)	Memory is synchronous but not collective	(Fig. 2, Fig. 1)
Synchronous Control Flow (`SCF`)	Synchronizes at basic block entry, exits via collectives	(Fig. 1, Fig. 9⊘sg)
Synchronous SG Op (`SSO`)	Synchronizes at SG ops and associated control dependencies	(Fig. 9⊘sg, Fig. 9⊕sg)
Speculative Collectives (`SC`)	May speculatively run collectives before synchronization	(Fig. 10, Fig. 11)
Symbolic Collectives (`SymC`)	Collectives don't synchronize and return symbolic values	(Fig. 11, NA)

implementations focus on the first four models, which we dub *direct models*, as we deem the last two models too complex for further consideration. We implement the SIMT-Step direct models in TLA+, providing executable semantics (Sec. 5). All models verify correctly against our tests, with each test completing in under one second. To facilitate further testing and exploration, we provide a front-end 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 is unknown. Many GPU ISAs are undocumented, and low-level compiler toolchains are often proprietary. To investigate this, we conduct a large-scale empirical study across real-world devices (Sec. 6.2).

We generalize our direct model tests across different memory access patterns, yielding 10 base tests. To account for relaxed behaviors introduced by compiler transformations, we apply a semantics-preserving fuzzer to each test [8], generating 100k fuzzed tests, which run on nine GPUs spanning seven vendors.

Our results show that, although specification groups have been reluctant to endorse strong subgroup semantics, most GPUs empirically exhibit strong subgroup behavior. We observe two notable exceptions. First, Qualcomm GPUs occasionally fail all test variants; however we suspect this may be due to compiler bugs rather than intentional subgroup semantic relaxation. Second, we find that nearly all GPUs, except those from NVIDIA, do *not* treat memory operations as collective. This behavior is illustrated in Fig. 2, where all non-NVIDIA platforms show the non-collective outcome. This observation is noteworthy because there exists performance-critical patterns, such as subgroup-elect idioms (e.g., from [33]), that rely on collective memory operations.

Initial: `int *loc = 0;`

```
atomicStore(loc, tid);
int read = atomicLoad(loc);
assert(subgroupAllEqual(read));
```

Allowed behaviors

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

Fig. 2. This test is executed by many threads across many subgroups. Each thread write a unique value (their id) to `loc`. Each thread then reads from the location. Under collective memory, all threads within the subgroup will read the same value. Otherwise, the load is allowed to return different values to different threads in the subgroup. The `subgroupAllEqual` operation checks if the arguments across all threads are all equal.

Scope and Limitations. This work focuses on formalizing relaxed subgroup execution semantics, which have been relatively unexplored until this point. To cleanly separate execution behavior from memory effects, we assume a sequentially consistent (SC) memory model, even though it is not technically supported in many GPU languages (e.g., SPIR-V, WebGPU, or Metal). We believe SIMT-Step can be extended to relaxed memory models, e.g., by having symbolic values in the operational semantics, as in [18].

Although we utilize many components from Vulkan (and its IR, SPIR-V) and rely on some of its control flow properties (see Sec. 2.2), SIMT-Step is designed to be general. In the same vein, although our executable semantics utilize SPIR-V instructions, we make no claim of being an official formalization of SPIR-V: we simply utilize the a subset of SPIR-V instructions to explore subgroup behavior. We note that CUDA does not provide the structured control flow or *reconvergence* properties that SIMT-Step requires. Here, reconvergence refers to the property that diverged threads in a subgroup are guaranteed to eventually arrive at a designated point in the control flow. Thus, it would likely be difficult to model all of CUDA using SIMT-Step; however, there are likely pragmatic subsets that could be targeted.

We also briefly address NVIDIA's independent thread scheduler (Volta onward) [45], which, although it claims to significantly relax synchronous (warp) behaviors, we still observe strongly synchronous behaviors on NVIDIA devices (see Sec. 6.2). Nevertheless, features like this may indicate a growing interest among vendors in exploring more relaxed execution models.

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, and can support varying degrees of collective and synchronous instruction execution.
- We define a range of SIMT-Step models, ranging from fully lockstep to symbolic execution. We verify the behavior of a pragmatic subset of the models using TLA+.
- To understand if real-world GPUs adhere to any SIMT-Step model, we conduct a large-scale empirical study, running over 100K tests across nine GPUs from seven vendors. Our results show that, in practice, most hardware exhibits strong synchronization behavior.

2 Background

2.1 GPU Architecture and Programming Models

Modern GPUs are high-throughput, massively parallel accelerators that are composed of multiple compute units (CUs), called streaming multiprocessors (SMs) by NVIDIA. Each CU contains many processing elements (PEs) that execute instructions in parallel. Threads are grouped into subgroups (or warps in CUDA) that follow the SIMT (Single Instruction, Multiple Threads) model, where threads execute the same instruction simultaneously but are also allowed to follow divergent control flow paths. Subgroup size is hardware-dependent—for instance, it is 32 threads on NVIDIA GPUs and can be either 32 or 64 on AMD GPUs.

GPU programming frameworks expose this hardware hierarchy through low-level APIs designed for high-performance computing. These include *CUDA* [43], *DirectX* [36], *Vulkan* [24], and *Metal* [3], each balancing portability and control differently. CUDA and Metal are vendor-specific (NVIDIA and Apple, respectively), while Vulkan and DirectX support multiple hardware platforms. 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 a function is launched across many threads, each with a unique id that can be used to access different data. The GPU's architectural hierarchy is reflected in the programming model: threads are grouped into subgroups, which execute in SIMT style and communicate via subgroup-level APIs; subgroups form workgroups, which are mapped to a single compute unit; and the full set of threads forms a grid. This close alignment between language and hardware enables low-level optimizations but also introduces semantic complexity, especially when taking into account allowed compiler transformations.

Subgroup Operations and Semantics. Modern GPU languages expose a variety of highly efficient *collective subgroup operations*, which operate across a subgroup. These operations include reductions, broadcasts, shuffles, ballots, and predicates like `All` and `Any`. While these operations are *collective*, they are intentionally *non-uniform*: not all threads in a subgroup are required to participate. Language specifications typically leave the participating set loosely defined, with the general 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, which we provide in this work.

Subgroup semantics also involve implicit synchronization behavior. It remains unclear whether operations like memory accesses or branches enforce any ordering or coordination among subgroup threads. Earlier documentation hinted at implicit synchronization (see Sec. 2.3), but this documentation has been omitted in many current specs. To this end, SIMT-Step provides flexibility on what implicit synchronization is provided.

SPIR-V. SIMT-Step targets a subset of SPIR-V (Standard Portable Intermediate Representation [21]), the intermediate shading language used by Vulkan. SPIR-V is a low-level, SSA-based intermediate representation for GPU programs. Like LLVM IR, SPIR-V is not typically written by hand. Instead, it serves as a common compilation target, allowing GPU vendors to avoid implementing separate backends for each high-level shading language. Today, SPIR-V is the compilation target for several widely used languages, including HLSL [37], GLSL [20], and OpenCL C [23]; it can target a wide range of backend GPUs, including those from: NVIDIA, AMD, Intel, Qualcomm, and ARM.

We express our semantics using a subset of SPIR-V instructions, while presenting examples in GLSL for clarity and readability. The mapping should be straightforward, and can be validated using `glslang`, the compiler for GLSL to SPIR-V, which is also available online through Godbolt [12]. Although our tools implement a wider variety of instructions (see Sec. 5), Tab. 2 lists the SPIR-V instructions used in this paper, including subgroup and control flow instructions (described next).

Table 2. SPIR-V instructions used in this paper, where NU should be expanded to `NonUniform`

Instruction	**Description**
	SPIR-V Instructions
`OpGroupNUAll`	Returns true if all participating threads provide true
`OpGroupNUAllEqual`	Return true if the provided values from all participating threads are equal
`OpGroupNUBroadcast`	Broadcasts a value from one specified participating thread to all others
	Label Instructions
`OpLabel`	Marks the beginning of a basic block with a unique ID
	Block Terminating Instructions
`OpBranch`	Unconditional branch to another label
`OpBranchCond`	Conditional branch based on a boolean value
`OpReturn`	Return from a function/shader
	Merge Instructions
`OpSelectionMerge`	Declares the merge block for a conditional construct

2.2 Structured Control Flow and Maximal Reconvergence

SPIR-V provides two key features that SIMT-Step relies on: structured control flow and a well-defined subgroup reconvergence model known as *maximal reconvergence*. As a target language, SPIR-V requires any source shading language to support these properties to generate valid code.

Structured Control Flow. SPIR-V enforces structured control flow by constraining valid control flow graphs and providing instructions that define both behavior and structure. Tab. 2 lists the instructions that we discuss in this paper examples, though our TLA+ model implements the full set. 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. 3 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 an `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.

```
%entry = OpLabel
          OpSelectionMerge %merge
          OpBranchConditional %c %then %else

%then = OpLabel
         ; comptuation
         OpBranch %merge

%else = OpLabel
         ; computation
         OpBranch %merge

%merge = OpLabel
          %res = OpGroupAll %arg
```

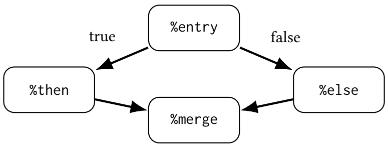


Fig. 3. An example of control flow in SPIR-V and the resulting control flow graph

Maximal Reconvergence. SPIR-V specifies a model for handling divergent control flow in subgroups, known as *maximal reconvergence* [22]. Under this model, all subgroup threads that execute a merge instruction must reconverge the first time they execute the merge target, regardless of potentially divergent paths taken. Fig. 3 illustrates this with a subgroup operation, `OpGroupAll`, 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`. As a result, the compiler cannot legally perform code motion on the subgroup operation—for example, by duplicating it and placing one instance in each branch—since those blocks may be executed by different subsets of the original threads from `%entry`.

2.3 A Brief History of Subgroups in GPU Programming Models

To provide context, we overview the evolution of subgroup (or warp) semantics in GPU programming, focusing on CUDA for its rich documentation and examples. We also comment on other GPU languages, some of which are only beginning to adopt subgroup semantics, making them timely targets for programming language research.

CUDA's Subgroup Evolution. Since the early days of GPU programming, developers have leveraged the lockstep execution of subgroups. A classic CUDA tutorial on reductions notes that once computation is confined to a warp, synchronization can be omitted, as threads execute in lockstep [14]. Subgroup operations like `__all` and `__any` have existed since CUDA 2.0 (2008) [38], however the documentation does not specify if uniform execution is required. Furthermore, these early guides state that subgroups are guaranteed to reconverge after divergence.

2012 was an eventful year for subgroup semantics. CUDA 4.2 [39], released in April, introduced a richer set of subgroup operations, including `__shfl`, `__shfl_up`, and `__shfl_down`, which enabled intra-subgroup register sharing and more efficient reductions without relying on implicit lockstep execution across normal memory operations [31]. Around the same time, programmers continued to exploit strong subgroup memory behavior, assuming collective and synchronous execution in algorithms for irregular applications [33]. Later that year, CUDA 5.0 [40] removed an example that had relied on implicit synchronization from its performance guidelines, without explicitly disallowing lockstep execution. However, examples began to surface showing compiler transformations that broke lockstep behaviors [47].

2017 was another eventful year. CUDA 9.0 [42] introduced the independent thread scheduler, documenting that lockstep execution within subgroups could no longer be assumed. Earlier subgroup operations were deprecated, just to be replaced with the same instructions with an explicit _sync suffix, indicating that they now synchronized threads. CUDA also introduced an explicit subgroup-level barrier, __syncwarp, and allowed subgroup operations to accept a bitmask to specify participating threads, enabling use in non-uniform control paths.

CUDA's subgroup programming model has not had major updates since. This model is difficult to formalize because unlike SPIR-V, CUDA does not provide structured control flow or maximal reconvergence. Furthermore, from our exploration, it appears they provide almost no implicit synchronization, which can lead to extremely counter-intuitive behaviors, as described in Sec. 4.5.

```
// initial block
for (int i = 0; i < 3; i++) {
  // loop block
  if (tid == i || i == 2) {
    // cond block
    subgroupOp(...);
    }
// final block
}
```

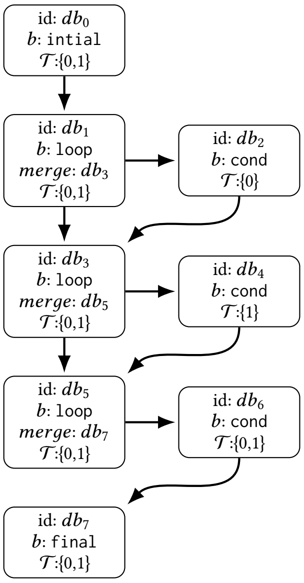


Fig. 4. An example GLSL program with divergent control flow, where a subgroup operation is executed conditionally within a loop. The dynamic execution graph underneath illustrates how threads diverge and reconverge throughout execution.

Other Shader Programming Languages. *OpenCL* added a subgroup API in November 2015 [19], constraining calls to uniform control flow, similar to workgroup barriers. *HLSL* introduced the concept of a *wave* (analogous to a subgroup) and *wave intrinsics* in Shader Model 6.0 (2017)[35], allowing their use under non-uniform control flow. *Metal* added *SIMD-group functions* in Metal 2.0 (June 2017)[4], also permitting use under non-uniform control flow. *WGSL* introduced subgroup operations most recently in January 2025 [52], but explicitly states that their use in non-uniform control flow as undefined and non-portable. In every case, there is almost no documentation on implicit synchronization within a subgroup, nor the consequences of independent execution specifications, which we provide in this work.

Verification Tools. As subgroup semantics have evolved, verification tools have addressed them in various, but never precise or comprehensive, ways. GPUVerify [5] assumed predicated, collective execution for all intra-subgroup instructions, an assumption now unsound under modern semantics and contradicted by our empirical findings in Sec. 6.2. GKLEE [29], a concolic execution engine for CUDA, initially implemented lockstep semantics but later supported interleavings. However, it did not handle non-uniform subgroup operations, and incorporating such support would be extremely challenging given the subtle semantics outlined in our work. More recent tools, such as Faial [30], omit subgroup semantics entirely, limiting their ability to reason about highly optimized kernels. We hope that SIMT-Step can provide a foundation for future verification tools to build upon, making assumptions about subgroup execution explicit and precise.

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, which we refer to as a *dynamic instruction instance*, denoted by i. Assuming instructions execute atomically, an execution is a sequence of such instances: $E = i_0 \rightarrow i_1 \rightarrow \ldots$ In a parallel program, E consists of interleavings of dynamic instruction instances executed by different threads, potentially constrained by synchronization. We write 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 instruction 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 db, 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 in the system. D can similarly be restricted to a thread t, noted D_t.

Basic Block and Dynamic Block Objects. We assume that a control flow graph (CFG) can be straightforwardly produced from SPIR-V code, especially given its structured control flow constraints. 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.
- **Static Children Blocks** ($b.C$): The immediate child blocks of b in the CFG.
- **Static Merge Target Block** ($b.merge$): If the block has multiple children, then under structured control flow, it must specify the basic block where the children merge.

A dynamic block db object naturally extends the basic block object with the following components:

- **Basic Block** ($db.b$): The basic block whose execution gives rise to db.
- **ID** ($db.id$): Since there can be multiple dynamic blocks for a given basic block, a label alone is not sufficient to distinguish them. Each dynamic block therefore carries an additional id.
- **Threads** ($db.\mathcal{T}$): The set of all threads that execute db.
- **Dynamic Block Children** ($db.C$): The dynamic blocks that are immediate children of db in the dynamic execution graph.
- **Dynamic Block Merge Target** ($db.merge$): If $db.b$ has a merge instruction, then this denotes dynamic block associated with $db.b.merge$, i.e., the dynamic block where $db.\mathcal{T}$ will reconverge after any potential divergent control flow.

Each dynamic block db must align with the structure of its corresponding basic block ($db.b$) in the CFG. For example, for the children relationship, this requires the following constraint: $\{db'.b \mid db' \in db.C\} \subseteq db.b.C$ Intuitively, the basic blocks of the dynamic children of d must correspond to the basic children of $db.b$. The same condition applies to the merge relationships.

Each dynamic instruction instance i_t is associated with both a basic block, $i_t.b$ and a dynamic block $i_t.db$. This also implies that the executing thread t belongs to $i_t.db.\mathcal{T}$. We define a *basic block instruction sequence* (denoted $bbis_t$) as the sequence of dynamic instructions that execute the same basic block by thread t. Each $bbis_t$ begins with a label instruction and ends with a branch-terminating instruction (see Tab. 2). As a result, E_t can be partitioned into a sequence of such $bbis_t$, which we denote $BBIS_t$. Each basic block instruction sequence $bbis_t \in BBIS_t$ is uniquely associated with a dynamic block $db \in D_t$, such that $\forall i \in bbis_t$, $i.db = db$. Finally, we define an auxiliary function, $next\_db$, which takes a dynamic instance i_t and returns the first instruction i'_t that appears after i_t in E_t, such that $i_t.db \neq i'_t.db$. In other words, $next\_db(i_t)$ identifies the transition point from i_t to the next dynamic block along thread t's execution.

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 in the grid.
- **Final**: We assume a final dynamic block db_f, where all threads branch when they finish shader execution. We do not consider infinitely running programs in this work.
- **No spurious divergence**: Given a thread t, for each of its basic block instruction sequences $bbis \in BBIS_t$, the dynamic block associated with each $i \in bbis$ must be the same, i.e., $i, i' \in bbis \implies i.db = i'.db$. Intuitively, this means that once a thread starts executing a dynamic block, it will continue executing the same dynamic block until completion. This allows us to associate a $bbis$ with a single dynamic block instance, noted as $bbis.db$.
- **Converged 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. That is, given two threads t, t', for any two instructions i_t and $i_{t'}$ such that $i_t.db = i_{t'}.db$ and $next\_db(i_t).b = next\_db(i_{t'}).b$, it must hold that $next\_db(i_t) = next\_db(i_{t'})$
- **Reconvergence**: For a given thread t, for all basic block instruction sequences $bbis_t$, if $bbis_t.db$ contains a merge block, then there must exist $bbis'_t$ later in the execution sequence that is associated with the merge block of $bbis_t$, i.e., $bbis_t.db.merge = bbis'_t.db$. Furthermore, there must not exist another $bbis''_t$ such that $bbis''_t$ lies in the execution sequence between $bbis_t$ and $bbis'_t$, and its associated basic block is the merge target of $bbis_t.db$, i.e., it cannot be the case that $bbis_t.db.merge.b = bbis''_t.db.b$. Intuitively, this states that a merge target dynamic block must be the next dynamic block in D_t associated with the merge target (static) basic block.
- **Non repeating**: A thread t cannot execute a dynamic basic block twice. That is, given any $i_{t0}, i_{t1}, i_{t2} \in E_t$ such that they appear in sequence, then $i_{t0}.db \neq i_{t1}.db \implies i_{t0}.db \neq i_{t2}.db$.
- **Acyclic**: The dynamic execution graph must be acyclic.

Combined, these rules allow the Thread ($\mathcal{T}$) set per dynamic block to specify which threads have converged execution at that point of the execution. This also formalizes the notion of maximal reconvergence (which prior to this work, has only been specified in prose). That is, for any two instruction instances, i, i', if $i.db = i'.db$, those two threads are executing in a converged manner; and the reconvergece property above specifies that threads must reconverge at the specified merge block as early as possible. However, the real power of dynamic blocks is that we can now formally specify which threads are guaranteed to participate in a subgroup operation, whether it is executed uniformly or not:

Property 1 (Subgroup Participation). *The execution of a subgroup operation by thread t gives rise to a dynamic instruction instance i_t and it belongs to a dynamic block db. The threads that must participate in this collective are defined $\mathcal{T}' \in db.\mathcal{T}$ such that all threads in $\mathcal{T}'$ belong to the same subgroup and there does not exist any other $t' \in db.\mathcal{T}$ such that t' is in the same subgroup as $\mathcal{T}'$. Intuitively, all threads that belong to the same subgroup that execute db will participate in any subgroup operation in db.*

Dynamic Block Example. We illustrate dynamic blocks with a simple example, focusing on how they determine thread participation in subgroup operations under divergent control flow. Fig. 4 shows a GLSL loop containing a conditional with a subgroup operation inside. The conditional is non-uniform and may split the subgroup. The dynamic execution graph below approximates basic blocks from the high-level code (a SPIR-V version would be more precise but more complex). We assume the program is executed with two threads, with `tid` values of 0 and 1, and we assume they are both in the same subgroup.

The execution begins with both threads in the initial block (db_0), as noted by the $\mathcal{T}$ set in the dynamic block. Both threads then enter the first loop iteration block (db_1) where `i = 0`. This dynamic block diverges depending on the conditional, but the threads must reconverge at the next loop iteration (db_3). The conditional splits the subgroup, with thread 0 executing the conditional block (db_2). This block contains a subgroup operation. According to Prop. 1, the participating threads in this subgroup operation is only thread 0, as it is the sole thread in $\mathcal{T}$ for (db_2). Similarly, in the next loop iteration (`i = 1`), the next conditional block db_4 is executed only by thread 1, and its constituent subgroup operation is only executed by thread 1. Because the dynamic execution graph must be acyclic, db_2 and db_4 must be disjoint dynamic blocks, and thus it is guaranteed that the first two iterations of the loop must execute two distinct instances of the subgroup operation with thread 0 and thread 1 as the participating threads, respectively. It is not allowed to merge them into a single subgroup operation with both threads participating.

In the third and final loop iteration, `i = 2`, the conditional is true for both threads, so both threads execute db_6, and thus both threads participate in the subgroup operation execution, before proceeding to the final block db_7. Because no thread executed the fall through from db_5 to db_7, they are technically not required to be connected.

Limitations: Escaping Merge Targets and Functions. In this work, we assume that if a thread t executes a dynamic block db, with a dynamic merge block $db_m = db.merge$, then t must also execute $db.merge$. That is $db.\mathcal{T} = db.merge.\mathcal{T}$. However, SPIR-V provides an exception to this condition. It may be possible for a thread to execute a terminating instruction before executing the merge target. We have explored candidate semantics under these conditions in our TLA+ implementation, but more work is needed to understand the impact of such behaviors.

Additionally, it is not clear if a function call should create a new dynamic block in a language like SPIR-V, as function calls are supported directly and not implemented with a branch. Thus, there may be subtle interactions with our current formalization, and we leave further analysis to future work.

tid	$\in \{0, \ldots, s-1\}$	Thread ID (up to subgroup size)
ℓ	$\in$ *Label*	Basic block labels
a	$\in$ *Addr*	Memory addresses
r	$\in$ *Reg*	Thread-local registers
v	$\in$ *Val*	Program values (e.g., integers)
S	::= shader f $\{B^+\}$	A shader function f
B	::= *Label* ; N^* ; T	A basic block
Label	::= label(ℓ)	A label instruction
N	::=	Normal instructions
	Load $r \leftarrow a$	Load from memory
	| Store $a \leftarrow$ *Unit*	Store to memory
	| $r \leftarrow$ SG_op(*op*, *Unit*)	Subgroup operation
	| Local(*Unit**)	An abstract local instruction
Term	::=	Terminating instruction
	branch(ℓ)	Unconditional branch
	| branchCond(*Unit*, ℓ_1, ℓ_2)	Conditional branch
	| return	Exit shader
Merge	::=	Merge instruction
	| selectionMerge(ℓ)	Merge for conditionals
Unit	::= r | v	A value or register

Fig. 5. The syntax for a simple GPU shader language (similar to SPIR-V) that we define an operational semantics for

4 SIMT-Step Operational Semantics for Subgroup behavior

With the dynamic block semantics in place, we can now design a flexible set of operational semantics for subgroup execution. We explore the semantic design space by varying the instructions that are executed either collectively, i.e., all at once across participating threads, or synchronously,

$M\colon\ a \longrightarrow v$	*Global memory: maps addresses to values*
$T:\ tid \longrightarrow \langle pc, db, R\rangle$	*Thread states: a program counter (pc), currently executing dynamic block (db), and set of registers (R)*
$D:\ \{db_0, \ldots, db_n\}$	*Dynamic blocks forming the dynamic execution graph*
$State ::= \langle M, T, D\rangle$	*Full system state*

Fig. 6. A summary of the system state

which implicitly adds a synchronization barrier across participating threads before execution. We reference the models summarized in Tab. 1 to discuss how they are instantiated.

For simplicity, we restrict our semantics to a single subgroup unless explicitly stated otherwise. Extending to arbitrary grid configurations—such as multiple subgroups and workgroups—is straightforward and involves partitioning $\mathcal{T}$ for each dynamic block. However, this generalization adds complexity without illuminating our core contribution: the semantics of subgroup execution. Full support for such configurations is included in our TLA+ implementation.

4.1 Syntax and State

The language, shown in Fig. 5, defines a single shader function f composed of at least one basic block B. Each basic block begins with a label instruction, followed by one or more normal (N) instructions, and ends with a block terminator $Term$. The standard instructions we model include memory loads and stores (assumed to be sequentially consistent), subgroup operations, and an abstract `Local` instruction that modifies only local state, such as registers. The subgroup operation `SG_op` takes an additional op argument that specifies its operation, e.g., `any` or `all`.

The system state is shown in Fig. 6. It consists of memory, thread-local state and a (partial) dynamic execution graph. The thread-local state includes a program counter (pc), which behaves as expected: given a shader program represented as a list of instructions I, the expression $I_{[pc]}$ refers to the instruction at position pc. When executing a basic block, the pc can be incremented to move to the next instruction. Note that each thread has its own independent pc, allowing flexibility in relaxing the constraints of subgroup lockstep execution. A thread also contains a dynamic block db, which it is currently executing.

The instructions are (statically) partitioned into the following sets: labels (*Label*), block-terminating instructions (*Term*), merge instructions (*Merge*), and normal instructions (*N*), as defined in Fig. 5. The normal instructions are further divided into collective operations (*Collective*), synchronous operations (*Synchronous*), and independent operations (*Ind*). This partitioning can be instantiated in several different ways, giving SIMT-Step the flexibility to capture a variety of possible semantics.

The dynamic block state is defined in Sec. 3; however, we add three additional components to a dynamic block d to help build and maintain the dynamic execution graph during execution:

- Unknown threads ($db.U$): Threads for which it is currently unknown whether they will execute db.
- Merge stack ($db.ms$): A stack of dynamic blocks that specify merge targets. This constrains which dynamic block a thread can branch to, in accordance with the reconvergence property of dynamic blocks (see Sec. 3).
- Synchronous Instruction Status ($db.sis$): A dictionary mapping each static synchronous instruction in $db.b$ to a Boolean value indicating whether the subgroup is currently synchronized at that instruction. Initially, all entries are set to `False`.

$$\frac{T(t) = \langle pc, db, R\rangle \quad I_{[pc]} \in \mathit{Collective} \quad \mathit{Aligned}(pc, db)}{\langle M, T, D\rangle \xrightarrow{\mathcal{T}} \langle M', T[\forall t' \in db.\mathcal{T} : t' := \langle pc{+}1, db, R'\rangle], D\rangle} \quad (\textsc{Step-Collective})$$

$$\frac{T(t) = \langle pc, db, R\rangle \quad I_{[pc]} \in \mathit{Synchronous} \quad \mathit{Aligned}(pc, db)}{\langle M, T, D\rangle \xrightarrow{t} \langle M, T, D[db.sis[I_{[pc]}] \leftarrow \mathit{True}\rangle} \quad (\textsc{Step-Synchronous Arrive})$$

$$\frac{T(t) = \langle pc, db, R\rangle \quad I_{[pc]} \in \mathit{Synchronous} \quad db.sis[I_{[pc]}]}{\langle M, T, D\rangle \xrightarrow{t} \langle M', T[t' := \langle pc{+}1, db, R'\rangle, D\rangle} \quad (\textsc{Step-Synchronous Execute})$$

$$\frac{T(t) = \langle pc, db, R\rangle \quad I_{[pc]} \in \mathit{Independent}}{\langle M, T, D\rangle \xrightarrow{t} \langle M', T[t' := \langle pc{+}1, db, R'\rangle, D\rangle} \quad (\textsc{Step-Independent})$$

Fig. 7. Rules for executing collective, synchronous, and independent instructions within a dynamic block.

4.2 Executing a Dynamic Block

We now specify how a subgroup executes within the same dynamic block db, i.e., excluding label and *Term* instructions. We define the following property, which we assume at this point:

Definition 1 (Converged Basic Block Execution). *Converged Basic Block Execution provides two properties: (1) dynamic blocks always have a complete view of their threads, that is, $db.\mathcal{T}$ is never updated after creation; and (2) threads execute dynamic blocks together. That is, if there exists a thread t executing dynamic block db (i.e., $t.db = db$) then all of db's threads are executing db, that is: $\forall t' \in db.\mathcal{T} : t.db = db$.*

Instructions can be executed in one of three ways: *collective*, *synchronous*, or *independent*, as specified in Fig. 7. The predicate $\mathit{Aligned}(pc, db)$ intuitively checks whether all threads in the dynamic block are at the same program counter; i.e., $\forall t' \in db.\mathcal{T} : t'.pc = pc$.

Collective. We begin with the most constrained form of execution. Collective execution occurs when all threads in the same subgroup are aligned at the same instruction within a dynamic block. In this case, the instruction is executed atomically by all threads, and its semantics ($\xrightarrow{\mathcal{T}}$) are defined over the entire set of threads, rather than a single thread. The operation may update memory (M) and thread-local registers (R), and the program counter is incremented for all threads in the dynamic block. The dynamic block itself is not updated. Executing all instructions collectively was the default approach in prior subgroup execution models (e.g., as implemented in GPUVerify). In contrast, SIMT-Step introduces flexibility by allowing different instruction sets to be executed collectively, depending on the desired semantics.

In our proposed SIMT-Step models, all of the direct models (`CM`, `SM`, `SM`, and `SSO`) execute subgroup operations collectively, as semantics get very complex when this isn't the case (as seen in Sec. 4.5). Furthermore, the strongest model (`CM`) also executes loads collectively, meaning that tranhreads in the same group are guaranteed to see the same value if they load from the same location. The semantics of collectively executing stores are less clear, but could be specified. For instance, one might guarantee that the value written is that of the thread with the highest (or lowest) ID. However, this behavior does not appear to be supported in current GPU programming models (e.g., CUDA), as discussed in the CUDA Programming Guide (see § 7.1 [44]), where it states that "which thread performs the final write is undefined".

Synchronous. The next type of normal instruction execution is synchronous: in this mode, threads must align on an instruction before execution, but they do not need to execute it collectively; for

$$\frac{T(t) = \langle pc, db, R\rangle \quad I_{[pc]} = merge(\ell)}{\langle M, T, D\rangle \xrightarrow{t} \langle M,\ T[t' {:=} \langle pc{+}1, db, R\rangle,\ create\_merge\_db(db, \ell, D)\rangle} \quad \text{(Step–Merge)}$$

$$\frac{T(t) = \langle pc, db, R\rangle \quad I_{[pc]} = Branch(\ell) \quad Aligned(pc, db)}{\langle M, T, D\rangle \xrightarrow{\mathcal{T}} \langle M,\ T[\forall t' \in db.\mathcal{T} : t' {:=} \langle pc', db', R\rangle],\ get\_or\_create\_child(db, \ell, db.\mathcal{T}, D)\rangle} \quad \text{(Step–UBranch)}$$

$$\frac{T(t) = \langle pc, db, R\rangle \quad I_{[pc]} = Label(\ell) \quad Aligned(pc, db)}{\langle M, T, D\rangle \xrightarrow{\mathcal{T}} \langle M,\ T[\forall t' \in db.\mathcal{T} : t' {:=} \langle pc+1, db, R\rangle]\rangle, D\rangle} \quad \text{(Step–Label)}$$

Fig. 8. Semantics for merge, unconditional branch, and label instructions under collective control flow. Conditional branch instructions are described in the text

example, their executions may interleave, especially with threads from other subgroups. These semantics are implemented with two steps: the first, Synchronous Arrive, simply occurs once all threads become aligned on the instruction: this triggers an update to the synchronous instruction status in the dynamic block (*db.sis*), which updates the map for the aligned instruction to be *True*. Once this status is updated, threads are free to continue with the Synchronous Execute step, where each thread may take a step individually according to the thread-local semantics ($\xrightarrow{t}$).

As with collective operations, SIMT-Step allows flexibility in whether instructions are treated as synchronous. In practice, the most consequential design space is around how memory operations are handled. That is, enforcing collective execution for memory operations may be too strong (and appears to lack empirical support; as shown in Fig. 13). Allowing memory operations to be synchronous instead of collective could be a reasonable relaxation. In fact, most GPUs appear to support this behavior empirically (again, see Fig. 13). In our proposed SIMT-Step models, `SM` provides synchronous memory operations, `CM` provides synchronous stores.

Independent. The most relaxed execution step is Independent; it mirrors how independent execution is typically modeled in concurrent programming. Each thread is free to execute any independent instruction at any point, without constraints on other threads in the dynamic block. These instructions update only thread local state and potentially memory. This step introduces the greatest flexibility: if memory operations are allowed to execute independently, interleaved behaviors, such as the one shown in Fig. 1, become possible. In the SIMT-Step models, `SCF` permits independent execution around memory operations within a basic block.

4.3 Collective Control Flow

The previous section assumed converged basic block execution (Def. 1) within a dynamic block, which simplifies the semantics. However, enforcing this property across control flow can be challenging, particularly when synchronous execution is relaxed. To address this, we now describe how control flow can be implemented to ensure converged basic block execution. This requires certain control flow instructions, specifically, branches and (perhaps surprisingly) labels, to execute collectively. A subset of their semantics is summarized in Fig. 8.

Reconvergence at merge blocks is managed using a *merge stack*, which is a stack of dynamic blocks. Each dynamic block maintains its own merge stack (*db.ms*), where the top element can be accessed via the *top* field and *pop*() function is used to remove the top element and return the updated merge stack. The stack records dynamic blocks that were designated as merge targets by previously executed control flow. In structured control flow, each newly executed merge instruction specifies the next merge block, resulting in a last-in-first-out pattern that makes a stack well-suited to implement reconvergence.

We first define a few helper functions that create and update dynamic blocks in the dynamic execution graph.

- $\mathit{create\_merge\_db}(db, \ell, D)$: This function creates a new dynamic block db', which will serve as the merge block of db. If it is called multiple times (e.g., by different threads), the block is created only once. It performs the following updates to D:
  - Create db'
  - $db.merge := db'$ (set the merge block of db)
  - $db'.\mathcal{T} := db.\mathcal{T}$ (all threads that execute db will also execute its merge block)
  - $db'.b := I_{[\ell]}$ (the basic block for db' corresponds to label ℓ in the program)
  - $db'.ms = db.ms.pop()$ (remove top element, assign the updated $db.ms$ to $db'.ms$)

- $\mathit{get\_or\_create\_child}(db, \ell, \mathcal{T}, D)$: This function returns a dynamic block db' that corresponds to $I_{[\ell]}$, is executed by threads $\mathcal{T}$, and is added as a child of db (i.e., $db.C$). In some cases, this block must be the dynamic merge block previously created by a call to *create_merge_db*.
  - If the top of the merge stack corresponds to $I_{[\ell]}$, i.e., $db.ms.top().b = I_{[\ell]}$, then the required db' already exists and can be returned.
  - Otherwise, a new db' is created with the following:
    * $db'.ms := db.ms + db.merge$ (inherit and update the parent's merge stack)
    * $db'.\mathcal{T} := \mathcal{T}$ (set the executing threads, which may differ from $db.\mathcal{T}$)
    * $db'.b := I_{[\ell]}$ (set the basic block based on the label)
    * $db'.merge :=$ none (no merge block assoicated with db')

Merge. The first operation we discuss is Merge, which takes an instruction with a label ℓ identifying the basic block where any potentially divergent control flow is guaranteed to reconverge. A thread executes this instruction by updating its program counter and the dynamic execution graph using the $\mathit{create\_merge\_db}(db, \ell, D)$ function. This execution does not need to be synchronous or collective, as long as the new block is created only on the first call for a given dynamic block.

Unconditional Branch. The next operation to consider is an unconditional branch (UBranch). To preserve converged execution, which requires that all threads in $db.\mathcal{T}$ execute within the same dynamic block db, branches must be executed collectively. This ensures that all threads leave the current dynamic block simultaneously. When the group of threads collectively executes this instruction, it updates their dynamic blocks to the next dynamic block db' and sets the program counter to the first instruction of db', denoted as pc'. It may also update or add new dynamic blocks to D, which is handled by the $\mathit{get\_or\_create\_child}(db, \ell, \mathcal{T}, D)$ function described earlier. Since this is an unconditional branch, the child block is executed by all threads in $db.\mathcal{T}$.

Conditional Branch. A conditional branch follows similar logic to the unconditional branch and is therefore executed collectively. The difference is that it involves a condition c and two branch targets, ℓ and ℓ'. Since our semantics do not include symbolic values, the condition c must be resolved at execution time across all threads. Due to its complexity, we specify this operation in text rather than in the operational notation.

First, we partition the threads: $db.\mathcal{T}_c$ and $db.\mathcal{T}_{\neg c}$ are the sets of threads for which c evaluates to true and false, respectively. $db.\mathcal{T}_c$ targets ℓ, while $db.\mathcal{T}_{\neg c}$ targets ℓ'. We define two (potentially new) dynamic blocks as targets for the branches, as follows:

- $db_c = \mathit{get\_or\_create\_child}(db, \ell, db.\mathcal{T}_c, D)$
- $db_{\neg c} = \mathit{get\_or\_create\_child}(db, \ell', db.\mathcal{T}_{\neg c}, D)$

The thread states are then updated as follows. Assume that the program counter (pc) for the first instruction of a dynamic block can be obtained through $db.pc$.

- $\forall t \in db.\mathcal{T}_c : t = \langle db_c.pc, db_c, R\rangle$
- $\forall t \in db.\mathcal{T}_{\neg c} : t = \langle db_{\neg c}.pc, db_{\neg c}, R\rangle$

Collective Labels. The collective branch operations ensure that threads *leave* dynamic blocks together. However, to preserve converged basic block execution, threads must also *enter* a dynamic block together. That is, consider a dynamic block db with two parents, p and p'. One parent, p, may collectively execute (and thus synchronize) on its branch and continue into db, but the threads from p' are not guaranteed to be executing in db.

To ensure that threads enter a dynamic block together, the label instruction can be set as a collective operation. While this is slightly unconventional, since labels are typically not considered executable, having them synchronize threads as a collective step provides the final mechanism needed to specify control flow semantics that preserve converged basic block execution execution. This behavior is specified in Label.

Divergent Execution. The semantics above allow divergent behavior within a subgroup, specifically through the conditional branch operation, where threads may take different paths. Currently, SIMT-Step conservatively permits the most permissive behavior across divergent thread groups: arbitrary interleavings. However, a specification may choose to prioritize one divergent path over another, with both 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 (as described in CUDA Programming Guide in Section §4.1[41]).

Considerations and Example. Combined, these rules ensure converged basic block execution (Def. 1): each dynamic block knows exactly which threads will execute it at the time of its creation, and those threads proceed through the block together. This property greatly simplifies the semantics. Allowing relaxations within a basic block, while still enforcing the intuitive notion of converged execution, may represent a sweet spot in semantic design. Our SIMT-Step model SCF embodies this: the control flow operations are collective (as described above), but there is no collective or synchronous instructions for memory operations within a basic block. However, some programming languages may choose not to provide any form of synchronization related to control flow (or, more generally, dependencies). For example, in the C++ memory model, the standards committee explicitly chose not to model such dependencies in order to allow greater flexibility for optimizations.

We note that enforcing synchronization at control flow instructions does not completely prohibit optimizing away control dependencies; they would simply need to be replaced with subgroup synchronization instructions, which are expected to be lightweight on current hardware, as threads are likely executed synchronously anyways.

Initial: `int *x = 0, *y = 0;`

```
int *addr1, *addr2;
if (tid == 0) addr1 = x; addr2 = y;
if (tid == 1) addr1 = y; addr2 = x;

atomicStore(addr1, 1);
if (tid == 0) {
  // (opt.) int a=subgroup_all(0);
  atomicStore(addr2, 2);
} else {
  // (opt.) int a=subgroup_all(0);
  atomicStore(addr2, 2);
}
```

Allowed behaviors

Converged:	`*x == 2 && *y == 2 ;`
Relaxed:	`*x == 2 && *y == 2 ||`
	`*x == 1 && *y == 2 ||`
	`*x == 2 && *y == 1 ;`

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

We illustrate the synchronous control flow behavior with an example in Fig. 9, which is similar to the example in Fig. 1. It is executed by two threads in the same subgroup. Each thread initializes its own address and writes to opposite locations in sequence. Under converged basic block execution, both threads must execute the store on line 5 and synchronize before taking the branch. After the branch, thread 0 and thread 1 write the value 2 to y and x, respectively. Thus, if branches provide synchronization, then only the converged (basic block) behavior is allowed. However, if threads are allowed to execute independently across branches, relaxed outcomes, corresponding to unconstrained interleavings of thread execution, may be possible. We will consider the case with the subgroup operation in the next section.

4.4 Independent Branches

In this section, we explore how to further relax synchronous behavior by removing the requirement that branches and labels be executed collectively. Our investigation focuses on the semantic design space under the following constraints: (1) we utilize an operational semantics; (2) we avoid introducing complex mechanisms such as speculative execution or symbolic values; and (3) subgroup operations remain collective, since they must combine values across all threads in a dynamic block. Under those constraints, we propose the weakest semantics we are able to derive. Given its complexity, we outline the formalism here and refer the reader to our TLA+ implementation for further details.

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

There are two stages involved when a thread t branches out of a dynamic block db. The first is dynamic block creation. The first thread to branch to a target label ℓ creates the new block. It does so similarly to $get\_or\_create\_child(db, \ell, \mathcal{T}, D)$. However, instead of initializing the block with a set of threads, it simply sets $db'.\mathcal{T}$ to contain only itself.

To initialize the unknown thread set of a newly created dynamic block db, take the thread set of the currently executing block, $db.\mathcal{T}$, and subtract the thread sets of all existing child blocks, i.e., compute $db.\mathcal{T} \setminus (\bigcup_{c \in db.C} c.\mathcal{T})$. This ensures that only threads which have not branched to any child are considered unknown. If thread t is not the first to branch to ℓ from db, then no new dynamic block is created, as the target block already exists.

The second stage of a branch is propagation. After branching, the target of thread t, denoted db', is known. At this point, t must be removed from the unknown thread set and added to $db'.\mathcal{T}$ and any other dynamic block that db' dominates. For any other child db_c of db, thread t must also be removed from the unknown thread sets along every path starting from db_c until it reaches a merge block db_m where $t \in db_m.\mathcal{T}$.

There are no special rules for label instructions. They can simply be executed non-collectively.

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

Implicit Synchronization. We view the above rules as weakening synchronization as much as possible without relying on speculation or symbolic values. However, some implicit synchronization remains. We illustrate this with the example in Fig. 9. Under the relaxations proposed in this section, if subgroup collective operations are excluded, then relaxed behaviors are allowed: control flow no longer synchronizes threads, and without collective operations, threads execute independently.

However, if subgroup operations are included, they change the behavior of the program. This is noteworthy because each subgroup operation collectively involves only a single thread: the operation on line 7 is executed only by thread 0, and the operation on line 10 only by thread 1. Due to *Align*, threads must wait to execute a collective operation until any *potential* participating thread has branched to a path where it is guaranteed not to execute the collective. Thus, while the collective operations do not individually synchronize threads 0 and 1, they are indirectly synchronized through the branch on line 6. As a result, only the converged behavior is allowed.

As noted earlier, if converged basic block execution is abandoned, the semantics become significantly more complex. We have found it difficult to describe the resulting implicit synchronization behavior in a way that would be useful to developers or implementers (or reasonable to include in a programming specification). One possible attempt might be:

> Before executing a collective subgroup operation c, any potentially divergent control dependency leading to c must be resolved for all threads in the subgroup.

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.

4.5 Further Relaxations

Lastly, we explore additional relaxations that could be possible through the use of speculation or 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 then discussing high-level tests. However, we note that they had serious consideration in the specification group discussions, as they allow incredible flexibility in implementations.

Initial: `int *x = 0;`

```
int cond = atomicLoad(x);
if (x == 0) {
  subgroup_all(0);
  atomicStore(x, 1);
}
```

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.

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

We illustrate the counterintuitive behavior enabled by speculative collectives in Fig. 10. This test is executed by two threads in the same subgroup with IDs 0 and 1. Rather than checking a final memory value, 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.

In fact, similar executions can be constructed for arbitrary subsets of threads. This is counter-intuitive because the speculation effectively justifies itself. In a non-speculative model (e.g., as described in the previous sections), the collective would not execute until the branching behavior of all threads is known. In that case, both threads would read 0 from x, take the branch, and participate in the collective operation. While we did not implement it due to the difficulties around speculative execution, this specification describes the SC SIMT-Step execution model.

This exact test spurred a lengthy discussion 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 and counter intuitive, but the implementation flexibility was desirable. The hope is to design a semantics to allow speculative collective execution without allowing counter intuitive behavior (and without specifying dependencies explicitly), but we are unable to see that solution at this time.

Symbolic Collectives. As discussed in Sec. 1, there has been ongoing debate about the extent to which compilers should be allowed to optimize subgroup operations. While there are clear cases where such operations could be optimized (e.g., when the argument is a constant), this is not always so obvious. When reasoning about allowed compiler optimizations, 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 such as super-optimizers [46]. This challenge is one reason the C++ memory model avoids formalizing the notion of dependencies: doing so could block future optimizations. However, this decision also leads to undesirable thin-air behaviors [16].

Initial: `int *x = 0;`

```
int *addr1, *addr2;
if (tid == 0) addr1 = x; addr2 = y;
if (tid == 1) addr1 = y; addr2 = x;
int b = atomicLoad(addr1);
int ta = subgroup_broadcast(b, 0);
atomicStore(addr2,ta);
```

Allowed behaviors

Symbolic (thin-air): `*x==`Any

Fig. 11. A test, executed with two threads in the same subgroup, showing the implications of allowing symbolic execution of collectives. If the subgroup operation is allowed to symbolically broadcast a value from thread 0 to thread 1, then thread 1 can store that symbolic value to a memory location, which can be read by thread 0 and used as the broadcast value. This creates a thin-air situation, where the final value of *x can be any possible value.

Taken to its logical conclusion, if the compiler is allowed to remove *any* subgroup operation, then the specification should be permissive enough to allow it to remove *all* subgroup operations. Semantically, this could be modeled by relaxing the requirement that collective operations execute collectively. Instead, each collective operation could execute independently and return a symbolic value. This symbolic value would be constrained and concretized after all relevant threads have completed execution.

However, allowing symbolic execution in synchronization semantics reintroduces thin-air behaviors, just as it does in relaxed memory models. We illustrate this with the example in Fig. 11. The test involves two threads in the same subgroup. Thread 1 first loads an arbitrary value from y. It then executes a broadcast operation, which attempts to read a value b from thread 0. Because thread

0 has not yet executed and initialized this value, the subgroup operation returns a symbolic value to thread 1, allowing it to proceed without synchronization. Thread 1 then stores this symbolic value to x.

When thread 0 eventually executes, it loads from x, receiving the symbolic value that originated from thread 1. It then broadcasts this symbolic value, which is also stored at x. At this point, the symbolic value is completely unconstrained and may resolve to any value. This constitutes a thin-air execution, despite all memory operations being sequentially consistent. Such behaviors are generally viewed as problematic and difficult to reason about [16]. Although we don't implement the semantics, and have concerns about its implications, the SIMT-Step model SymC could theoretically execute subgroup collective operations symbolically, following the above description. At this point, there are *no more* possible relaxations. There are no operations that are executing collectively or synchronously. Thus SymC represents the extreme end of independent subgroup execution, and contains concerning semantic consequences.

To avoid these thin-air issues, one approach is to allow a compiler to optimize a subgroup collective operation by replacing it with a subgroup synchronization barrier, omitting the actual collective operation. This may be slightly more efficient while still preserving the required synchronization.

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 four of the models described in §1.2; (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 proposed semantics.

Executable Semantics. We implement the semantics from §4 in TLA+, closely mirroring the specified states and transitions. Each GPU thread is modeled as an independent TLA+ process that executes SPIR-V-like instructions. The system can be configure the executing threads into arbitrary grid dimensions, i.e., subgroup and workgroup sizes. Assertions can be evaluated either during execution or post-mortem, once all threads have terminated.

Because we assume sequential consistency, memory is modeled as a simple mapping from integer addresses to integer values. Loads return the current value at an address, and stores update the map accordingly. While the current model supports only scalar memory accesses (not arrays) and only integral types (integers, unsigned integers, and booleans).

Our TLA+ semantics implement over 100 SPIR-V instructions, which are used to support a GLSL front-end (described next). The majority are local instructions that operate solely on thread-local state. Relevant to SIMT-Step semantics, we implement the merge instruction and both types of branches. We also implement six subgroup collective operations, including a synchronization barrier, which is treated as a collective operation without a computation.

To support the range of SIMT-Step models described in Section 1.2, our implementation provides several configurable options. Specifically, memory loads, branches, and labels can each be treated as either independent or collective operations. Additionally, any instruction can be marked for synchronous execution, which we use to implement the SM (Synchronous Memory) model, i.e., by marking memory operations as synchronous. To provide modularity, the operational semantics are written in a TLA+ module; a program can then be expressed as a list of instructions as TLA+ structures. The two can be combined and then checked in TLC, the model checker for TLA+. As a heuristic to help tame the exploding state space, we constrain execution so that interleavings are only considered at steps that can have impact outside of the thread local state, e.g., memory operations or control flow (because it updates the dynamic execution graph).

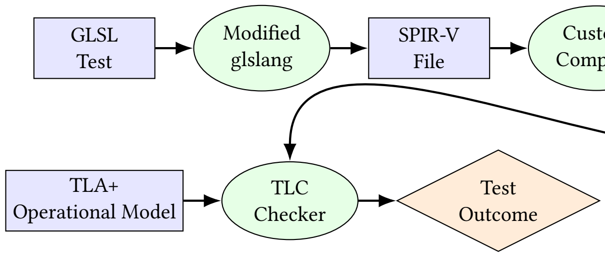


Fig. 12. End-to-end toolchain for executing high-level test cases (in GLSL) in the TLA+ model (that executes SPIR-V instructions)

An Accessible Front End. To make test development easier, we provide two translations layers that enable writing test shaders in GLSL. Our execution pipeline is shown in Fig. 12. First, although our executable model defines semantics for SPIR-V, the TLA+ model requires input in its own language. We developed a custom compiler that translates SPIR-V into TLA+, including full lexing and parsing. This approach allowed us to extend SPIR-V with additional features, such as grid configuration directives (e.g., subgroup size and thread count), synchronization directives (e.g., which instructions to execute collectively) and intra-shader assertions, which are not natively supported.

Because GLSL is a much richer language, we do not implement a compiler from scratch. Instead, we modify the existing `glslang` compiler, which translates GLSL to SPIR-V. Our extensions add support for custom `#pragma` directives to specify grid configuration and synchronization options, as well as intra-shader assertions. These features are compiled directly into our extended SPIR-V.

Discussion. Although this paper focuses on subgroup semantics, our model also includes candidate extensions to support workgroup-level synchronization, e.g., workgroup barriers. This generalization fits naturally into our framework, specifically, the `Align` function (Sec. 4.2) can be extended to collectively execute across converged workgroups. One reason we chose TLA+ is its support for temporal logic, which has been used to model other GPU semantic properties, e.g., inter-workgroup forward progress[48]. Extending our framework to model such properties is feasible, though nontrivial. Incorporating relaxed memory semantics, however, presents a deeper challenge, as even defining operational models for mainstream memory models (such as C++) remains a difficult and ongoing area of research [2, 18].

5.1 Generalizing Tests

Throughout, we have shown example tests, i.e., see the tests column in Tab. 1). We now discuss how test cases can be generalized along two dimensions: 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.

- **Collective Mem**: This test corresponds to Fig. 2 and tests whether memory operations are executed collectively.
- **Mem LockStep** : this test corresponds to Fig. 1 without the subgroup operation and tests whether memory operations execute synchronously.
- **Branch Sync**: This test corresponds to Fig. 9 without the subgroup operations and tests whether threads have synchronous control flow

- **Subgroup Sync**: This test corresponds to Fig. 9 with the subgroup operations included. It tests if subgroup operations synchronize with associated control flow.

With the exception of Collective Mem, the other tests follow a similar pattern: they create a racey (i.e., non-deterministic depending on the scheduler) program that, under fully independent execution, would allow the full set of interleaved results. The test then asks which subset of these behaviors are allowed, and infers any implicit subgroup synchronization properties from the results.

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 (ww), read-write (rw) and write-read (wr) pairs. This gives us 10 possible base tests. While SIMT-Step generally doesn't distinguish between the memory access type (loads and stores), these different test instantiates 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, as long as it is greater than 1). Instead of using fixed addresses (e.g., x and y), the tests now use a buffer. Each thread accesses a dedicated slot in a buffer indexed by its thread id for the first access; and for the second access, threads target a subgroup neighbor, typically the next thread in the subgroup, wrapping around at the end. This spreads race conditions across the entire subgroup, and instantiates the test across potentially many subgroups that are executing the test. The Collective Mem. test is an exception, as it checks intra-subgroup atomicity and permits all threads to access the same location; thus it requires no changes to scale.

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 [49], which applies semantics-preserving transformations to the input code. That is, given a test such as the one in Fig. 2, 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. To use GraphicsFuzz, we had to add support for atomic operations, which were previously missing. This strategy increases coverage across compiler behaviors that may interfere with otherwise strong subgroup synchronization at the hardware level.

6 Evaluation and Results

6.1 Executable Semantics

To avoid the state space explosion in model checking concurrent systems, 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. 5.1, which we write in GLSL and pass 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-cores, 20-threads, 5.40 GHz i9-13900H and execute TLC in parallel using all 20 threads. Our results are shown in Tab. 3. Each test behaves as expected under each implemented model (i.e., the direct models from Sec. 1.2). Note that the smallest configuration to test the Collective Mem tests (2,2) as opposed to (1,2). This is larger because it requires two subgroups to potentially disrupt atomicity violations in collective memory operations.

Perhaps surprisingly, 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 see that there are many more SPIR-V lines of code than GLSL (around 3×), as there are boilerplate SPIR-V instructions around the higher-level GLSL constructs. We start to see the state

Table 3. 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.

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

space explosion in the Collective Mem test (with over 10× more states), as it requires four total threads instead of just two. Despite this, our tests still execute in under one second.

6.2 Empirical Investigation

We evaluate subgroup behavior on a diverse set of real-world GPUs by running our fuzzed test suite across nine devices from seven vendors, covering mobile, integrated, and discrete hardware (see Tab. 4). This allows us to assess how well current devices align with potential semantics, as specifications are more likely to adopt behaviors that are already broadly supported.

Tests are written in GLSL (via GraphicsFuzz) and executed using the Amber framework [13], which compiles them to SPIR-V and runs the resulting code. Amber is widely supported and provides a mature subgroup API, in contrast to portable platforms like WebGPU, where subgroup support is still limited and evolving.

Our tests require the `VK_KHR_shader_subgroup` and `VK_KHR_shader_maximal_reconvergence` extensions, and SPIR-V version 1.3 or newer. Two notable GPU vendors were not included: Apple because they don't support Vulkan, and Imagination, as their devices report a subgroup size of one, i.e., which offers no meaningful intra-subgroup behavior. Each test is run with 65K workgroups, each with 128 threads (though subgroup sizes vary across devices). These parameters are the largest that are Vulkan guarantees to be portably supported [25]. Our testing runtime totals over 700 hourse, with mobile GPUs generally taking longer than larger, discrete GPUs.

GLSL (and SPIR-V) do not support sequentially consistent atomics, so our tests use the strongest available operations: release for stores and acquire for loads. Although this falls short of full sequential consistency, we found no practical alternative for testing a broad range of GPUs. All memory operations use subgroup scope, except in the collective memory test, which requires device scope because all threads access the same location. Despite this limitation, the tests remain well-defined and race-free. We consider this acceptable for several reasons: (1) weak memory behaviors on GPUs are rare and typically require highly specialized setups; and (2) our aim is exploratory, not to provide definitive proof of subgroup behavior.

As with any large empirical study, occasional failures occurred. In some cases, GraphicsFuzz generated code with compiler errors; this was largely due to our updates which added atomic operations, in which there was an occasional mismatch between supported memory regions. Rather than fix the issue (which ended up being quite complex), we simply ignored such tests. In very rare cases, we observed driver, compiler, and runtime crashes. These issues affected fewer than 15% of all tests run. Only tests that executed successfully are included in our reported results.

Table 4. 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, which is necessary to get the correct subgroup size [28].

Device	GPU	Vendor	Type	OS	Driver	SG Size
Desktop	Radeon RX 7900	AMD	Discrete	Windows 11	23.10.2	64
Desktop	GeForce RTX 4070 Super	NVIDIA	Discrete	Ubuntu 22.04	537.13	32
Jetson Nano	Tegra Orin	NVIDIA	Integrated	Ubuntu 22.04	2264989696	32
Minisforum MS-01	Intel Iris Xe Graphics	Intel	Integrated	Arch Linux 6.15.2	Mesa 25.1.4	16
Pixel 7a	Mali-G710 MP7	ARM	Mobile	Android 13	213909504.0	16
Galaxy S24	Xclipse 940	Samsung	Mobile	Android 14	100675593.0	32
One+ 11	Adreno 740	Qualcomm	Mobile	Android 13	2150252609.0	64
Galaxy S22	Adreno 730	Qualcomm	Mobile	Android 14	2150002688.0	64
Surface Pro	Adreno X1-85	Qualcomm	Integrated	Windows 11	0.814.0	64

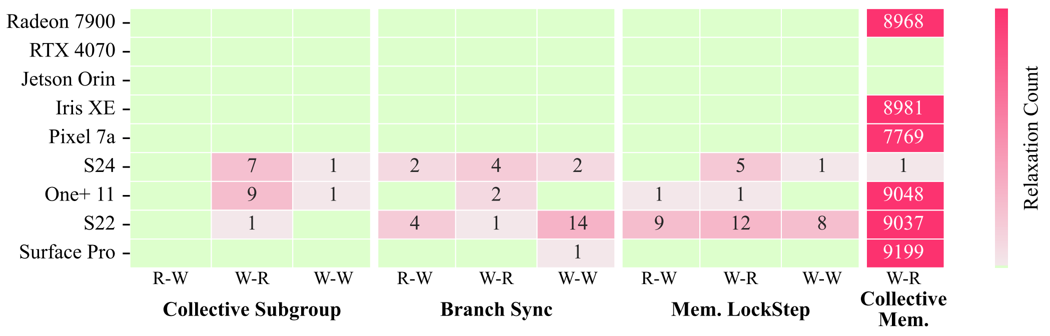
Results. Our results are presented in Fig. 13, with rows representing GPUs and columns representing test patterns. Tests are grouped first by pattern (as described in Sec. 5.1), and then by memory access sequence, where **W** indicates a write and **R** a read. Each cell shows how many times relaxed behavior was observed. Green cells indicate no relaxed behavior, while red cells indicate one or more occurrences (the darker the red, the more frequently relaxed behavior appeared).

Our first observation is that most devices exhibit non-collective memory behavior, with frequent relaxed outcomes. The only exceptions were NVIDIA, which showed no relaxed behaviors, and a Samsung device, which exhibited just one. In cases where violations were rare, we suspect potential compiler bugs, as our tests rely on atomic operations and inter-subgroup behaviors that have not been widely fuzzed. These results suggest that the strongest SIMT-Step model, Collective Memory (`CM`), is too strict for current systems and therefore likely unsuitable for adoption in official specifications. Despite follow-up testing and discussions with implementers, we were unable to determine whether the observed relaxations stem from hardware or compiler behavior. Notably, collective memory behavior was assumed in several classic GPU optimization techniques (e.g., leader election [34]), which our results show may be unsound on non-NVIDIA devices. Fortunately, modern subgroup APIs offer safer and more portable alternatives.

Our next observation is that additional relaxations appeared only on Qualcomm and Samsung GPUs, and even then they were extremely rare. We attempted to apply a test case reducer to isolate the causes, but it failed to significantly reduce the tests. Given the rarity of these behaviors, we hypothesize that they may result from compiler bugs rather than deliberate relaxation of subgroup synchronization. We are currently in contact with vendor representatives to discuss these findings.

Finally, we observe that many GPUs, including NVIDIA devices, exhibit strong subgroup-synchronous behavior. This holds even in the presence of features such as NVIDIA's independent thread scheduler, which has been described as weakening subgroup synchronization. Notably, NVIDIA also appears to support collective memory behavior. These findings suggest that, empirically, most GPUs exhibit strongly synchronous execution. We believe that this supports our endorsement of a synchronous control flow (`SCF`) model, as current GPUs seem to support these semantics, and furthermore, they do not even seem to be taking advantage of simpler relaxations allowed within basic blocks (as would be detected by Mem. LockStep tests). Thus, it is difficult to see the need for more complicated specifications.

We hypothesize that prior reports of subgroup relaxations (e.g., from [47]) may not appear in our results because our tests use atomic operations, which may inhibit compiler reorderings. This highlights an important considerations: relaxed behavior may arise either from relaxations either



	Collective Subgroup R-W	Collective Subgroup W-R	Collective Subgroup W-W	Branch Sync R-W	Branch Sync W-R	Branch Sync W-W	Mem. LockStep R-W	Mem. LockStep W-R	Mem. LockStep W-W	Collective Mem. W-R
Radeon 7900										8968
RTX 4070										
Jetson Orin										
Iris XE										8981
Pixel 7a										7769
S24		7	1	2	4	2		5	1	1
One+ 11		9	1		2		1	1		9048
S22		1		4	1	14	9	12	8	9037
Surface Pro						1				9199

Fig. 13. Relaxation Heatmap. Cells where a device showed no relaxed behaviors are colored green. Red cells are annotated with the number of relaxations observed.

in subgroup execution or the memory model. As described earlier, we do not believe our tests show relaxed behavior, but future work, e.g., on program analysis tools, will need to be able to incorporate both subgroup and memory model semantics to precisely model systems.

7 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. We provided many details about the history and prior work in this area in Sections 1 and 2. Here we note a few explorations into updating hardware SIMT execution.

Hardware Subgroup Execution. Subgroups generally correspond to sets of GPU threads executing in SIMT style, 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 underlying SIMT implementation directly constrains feasible programming models. Several works have proposed new hardware SIMT execution. For example, [11] introduced dynamic warp formation, regrouping threads based on control flow to improve utilization under divergence. Another work, [9] tackled synchronization bottlenecks by proposing BOWS, a scheduler that deprioritizes spinning warps, and DDOS, a hardware mechanism for detecting spin loops. 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.

8 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 explicitly tracks convergence and divergence across threads. We show that relaxed models require complex semantics and allow counterintuitive behaviors. We instantiated several SIMT-Step models and implemented them in TLA+, validating their behavior with a suite of idiomatic tests. To explore the subgroup behavior current GPUs implement, we run thousands of fuzzed GLSL tests across a wide range of GPUs, finding that most exhibit strong subgroup synchrony. These contributions provide a foundation for formalizing subgroup semantics in specifications and implementations.

References

[1] Jade Alglave and Luc Maranget. 2015. Towards a formalisation of the HSA memory model in the cat language.

[2] anonymous. [n. d.]. Recurrence Sets for Proving Fair Non-termination under Axiomatic Memory Consistency Models. UNDER SUBMISSION.

[3] Apple Inc. 2024. Metal Developer Resources. https://developer.apple.com/metal/resources/ Accessed: 2025-07-06.

[4] Apple Inc. 2025. *Metal Shading Language Specification*. Apple Inc. Version 4.0.

[5] 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

[6] 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.

[7] 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

[8] 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

[9] 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

[10] 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.

[11] 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

[12] Matt Godbolt. 2025. Compiler Explorer. https://godbolt.org. Accessed: 2025-07-08.

[13] Google. 2025. Amber. https://github.com/google/amber. Accessed: 2025-04-24.

[14] Mark Harris. 2007. Optimizing Parallel Reduction in CUDA. *NVIDIA Developer Technology* 2 (2007), 70. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf Slide deck, accessed 2025-07-07.

[15] 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

[16] 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

[17] Amir Ashraf Kamil and Katherine A. Yelick. 2006. *Concurrency Analysis for Parallel Programs with Textually Aligned Barriers*. Technical Report UCB/EECS-2006-41. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-41.html

[18] 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

[19] Khronos Group. 2015. *OpenCL Overview*. Khronos Group. https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf Accessed: 2025-07-07.

[20] 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.

[21] Khronos Group. 2021. SPIR-V Specification. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html#Tangle. Accessed: 2025-05-12.

[22] Khronos Group. 2024. SPV_KHR_maximal_reconvergence Extension Specification. https://github.khronos.org/SPIRV-Registry/extensions/KHR/SPV_KHR_maximal_reconvergence.html Revision 2, Last Modified: 2024-04-18.

[23] Khronos Group. 2025. OpenCL™ SPIR-V Environment Specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_Env.html Version 3.0.18, April 3, 2025.

[24] Khronos Group. 2025. Vulkan API Specification. https://docs.vulkan.org/spec/latest/index.html Version 1.3, Last accessed: May 12, 2025.

[25] Khronos Group. 2025. Vulkan® 1.3.283 - Limits. https://docs.vulkan.org/spec/latest/chapters/limits.html. Accessed: 2025-07-10.

[26] 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
[27] 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
[28] Raph Levien. 2020. Prefix sum on Vulkan. https://raphlinus.github.io/gpu/2020/04/30/prefix-sum.html. Blog post.
[29] 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
[30] Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2024. Sound and Partially-Complete Static Analysis of Data-Races in GPU Programs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 357 (Oct. 2024), 28 pages. doi:10.1145/3689797
[31] Justin Luitjens. 2014. Faster Parallel Reductions on Kepler. NVIDIA Technical Blog. https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/ Accessed: 2025-07-03.
[32] Daniel Lustig, Sameer Sahasrabuddhe, 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
[33] 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
[34] 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
[35] Microsoft. 2017. HLSL Shader Model 6.0 features for Direct3D 12: Terminology. https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/hlsl-shader-model-6-0-features-for-direct3d-12#terminology Accessed: 2025-07-07.
[36] Microsoft. 2025. DirectX-Specs. https://microsoft.github.io/DirectX-Specs/. Accessed: 2025-07-06.
[37] Microsoft Corporation. 2019. Reference for HLSL. https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference Accessed: 2025-05-12.
[38] NVIDIA Corporation. 2008. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0 Beta2.* NVIDIA Corporation, Santa Clara, CA. June 7, 2008.
[39] NVIDIA Corporation. 2012. CUDA C Programming Guide. Version 4.2, released 2012-04-16.
[40] NVIDIA Corporation. 2012. CUDA C Programming Guide. Version 5.0, released 2012-10-16.
[41] NVIDIA Corporation. 2016. *CUDA C Programming Guide, Version 8.0.* NVIDIA Corporation. https://docs.nvidia.com/cuda/archive/8.0/pdf/CUDA_C_Programming_Guide.pdf Section §4.1, p. 69.
[42] NVIDIA Corporation. 2017. *CUDA C Programming Guide, Release 9.0* (9.0 ed.). NVIDIA Corporation, Santa Clara, CA. https://docs.nvidia.com/cuda/archive/9.0/pdf/CUDA_C_Programming_Guide.pdf PG-02829-001_v9.0.
[43] NVIDIA Corporation. 2024. *CUDA C Programming Guide: SIMT Architecture.* NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture Accessed: 2025-05-12.
[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.
[45] NVIDIA Corporation. 2025. *CUDA Volta Tuning Guide* (v12.9 ed.). NVIDIA, Santa Clara, CA. https://docs.nvidia.com/cuda/volta-tuning-guide/index.html Last updated May 31, 2025.
[46] 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
[47] Tyler Sorensen, Ganesh Gopalakrishnan, and Vinod Grover. 2013. Towards shared memory consistency models for GPUs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. Association for Computing Machinery, New York, NY, USA, 489–490. doi:10.1145/2464996.2467280
[48] 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, Article 131 (Oct. 2021), 30 pages. doi:10.1145/3485508
[49] The GraphicsFuzz Authors. 2025. GraphicsFuzz testing framework. https://github.com/google/graphicsfuzz. Accessed June 30, 2025.
[50] The Khronos Group. 2018. *Vulkan has just become the world's first graphics API with a formal memory model. So what is a memory model and why should I care?* The Khronos Group. 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.
[51] Haining Tong, Natalia Gavrilenko, Hernan Ponce de Leon, and Keijo Heljanko. 2025. Towards Unified Analysis of GPU Consistency. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA,

329–344. doi:10.1145/3622781.3674174

[52] W3C. 2025. WebGPU Shading Language (WGSL) specification: Subgroup operations (15.6.3). https://www.w3.org/TR/2025/CRD-WGSL-20250116/#subgroup-ops. Accessed: 2025-07-07.

[53] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 86–97. doi:10.1145/1806596.1806606