Uniformity Analysis in the WebGPU Shading Language

JAMES LEE-JONES, Imperial College London, United Kingdom
JOHN WICKERSON, Imperial College London, United Kingdom
ALASTAIR F. DONALDSON, Imperial College London, United Kingdom

The WebGPU programming model brings general-purpose GPU programming to the web, allowing untrusted JavaScript to issue parallel workloads to client GPUs. To ensure reliability, WebGPU mandates *uniformity analysis*—a static check that rejects programs that could cause *barrier divergence*, a GPU control-flow error that can hang execution and require OS-level recovery. While traditional GPU models treat barrier divergence as undefined behaviour, WebGPU's need for safety and reliability makes this unacceptable. We present the first comprehensive formal and practical study of uniformity analysis in WebGPU, identifying four key issues and making corresponding contributions: (1) *Lack of definition:* The analysis currently defines non-uniform programs only as those it rejects, without an independent notion of barrier divergence. We provide an operational semantics for TinyWGSL, a core calculus of the WebGPU Shading Language (WGSL), developed in consultation with WGSL specification editors and implementers. This semantics rigorously defines barrier divergence in the context of WGSL for the first time. (2) *Complex specification:* The current description of uniformity analysis in WGSL is lengthy and imprecise. We reformulate it via concise formal rules for TinyWGSL and argue their soundness with respect to our semantics. (3) *Lack of soundness and precision:* Using our semantics, we expose soundness and precision flaws in the analysis as presented in the WGSL specification. In response, we have proposed four significant changes to the specification (all accepted). (4) *Testing difficulty:* The complex, semi-formal definition of uniformity in the WGSL specification makes it hard to test implementations. We mechanise uniformity analysis in Alloy and use Alloy's test generation to stress-test the Chromium implementation, revealing specification–implementation discrepancies and a bug in the Chromium implementation. Overall, our work resolves an important GPU programming problem (rigorously defining barrier divergence), brings an interesting new analysis to the attention of the PL community, and demonstrates the impact of applying formal PL techniques to an important industrial language specification.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; • **Theory of computation** → *Operational semantics*; • **Software and its engineering** → *Compilers*.

Additional Key Words and Phrases: GPUs, WebGPU, operational semantics, barrier divergence

ACM Reference Format:
James Lee-Jones, John Wickerson, and Alastair F. Donaldson. 2026. Uniformity Analysis in the WebGPU Shading Language. *Proc. ACM Program. Lang.* 10, PLDI, Article 253 (June 2026), 25 pages. https://doi.org/10.1145/3808331

1 Introduction

WebGPU is a new JavaScript API that brings general-purpose GPU computing to the web browser [58]. Building on the success of its predecessor WebGL [23] (routinely used to render major websites such as Google Maps [35], but limited by the legacy OpenGL API on which it is based), WebGPU offers graphics and compute APIs similar in capability to modern desktop and mobile APIs such as Vulkan, Direct3D and Metal [57]. This unlocks impactful new application areas—e.g. Microsoft

Authors' Contact Information: James Lee-Jones, Department of Computing, Imperial College London, London, United Kingdom, james.lee-jones20@imperial.ac.uk; John Wickerson, Department of Electrical and Electronic Engineering, Imperial College London, London, United Kingdom, john.wickerson@imperial.ac.uk; Alastair F. Donaldson, Department of Computing, Imperial College London, London, United Kingdom, alastair.donaldson@imperial.ac.uk.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART253
https://doi.org/10.1145/3808331

report that WebGPU enabled a 19× speed-up of an encoder in ONNX Runtime Web, a JavaScript library for in-browser deployment of machine learning models [46].

Despite its promise, WebGPU enlarges the attack surface of browsers by allowing untrusted JavaScript code to issue arbitrary computational workloads to client-side GPUs. Unlike traditional web APIs that operate in tightly sandboxed environments, GPU workloads execute in a shared hardware context with limited isolation between applications. A malicious or malformed WebGPU workload has the potential to cause the entire graphics subsystem to hang or crash. Such failures occur at the system level, beyond the browser's process isolation, making it hard to confine their effects to a single tab or browser instance, which can lead to denial-of-service conditions or expose GPU driver vulnerabilities [13]. Notably, a hang in the graphics subsystem of a device can lead to a display freeze until OS-level recovery mechanisms reset the graphics subsystem.

A GPU-specific mechanism that can cause these kinds of hangs is the *execution barrier*. Barriers are the main means of synchronisation between threads in a GPU program, and GPU hardware barriers are only guaranteed to function correctly if executed under *uniform* control flow. Executing a barrier under *divergent* control flow is called *barrier divergence*, and GPU programming models such as Vulkan, Direct3D and Metal dismiss barrier divergence as *undefined behaviour* (see Section 2.3), putting the onus on programmers to ensure their GPU programs are free from barrier divergence.

Simply declaring barrier divergence as undefined behaviour is not an option for WebGPU because programs that exhibit barrier divergence may trigger the kinds of GPU hangs that (as discussed above) threaten to compromise client-side reliability. Instead, WebGPU demands that GPU programs—which are written in the WebGPU Shading Language (WGSL) [15]—must pass a static *uniformity analysis* before being allowed to run on the client-side GPU. The aim of this analysis is to reject all WGSL programs that would lead to barrier divergence if executed.[1]

However, while this aim is laudable, uniformity analysis suffers from several problems: ❶ the analysis is not based on a rigorous definition of barrier divergence, ❷ the analysis is lengthy and described in an imprecise manner, ❸ the analysis suffers from problems related to soundness and precision, and ❹ it is fundamentally challenging to test implementations of the analysis.

We present an in-depth formal and practical study of barrier divergence and uniformity analysis, contributing: operational semantics that formally define barrier divergence for a core WGSL fragment; a formal account of uniformity analysis for this fragment, leading to the fixing of various problems with the WGSL specification; and a lightweight mechanisation of uniformity analysis using the Alloy language and analyzer [20] that facilitates automated testing of implementations.

Problem ❶: Uniformity analysis lacks a rigorous definition of barrier divergence. A conservative static analysis usually targets a *defined* undesirable runtime behaviour to be eliminated: to be *sound*, the analysis must reject all programs that could exhibit the behaviour; to be *useful* it should aim for *precision*, rejecting as few programs that do not exhibit the behaviour as possible. The WGSL uniformity analysis targets barrier divergence, but lacks a rigorous definition of what it means for barrier divergence to occur at runtime. The specification states that "control barriers must only be executed in uniform control flow" [15, §15.6.1], but also that "The [uniformity] analysis is what actually defines [uniform control flow], and when a program is valid or breaks the uniformity rules." [15, §15.2.1]. The role of uniformity analysis is to reject programs that might exhibit barrier divergence, but the programs that "might exhibit barrier divergence" are defined to be exactly those that uniformity analysis rejects, making a claim that the analysis is sound [15, §15.2.2] vacuous.

[1]Uniformity analysis also assesses correct use of other operations such as *derivative*, *texture* and *subgroup* operations, but misuse of these features does not cause GPU hangs. For simplicity we do not discuss them further, but we note that the uniformity analysis improvements arising from our work positively impact treatment of these other features too.

Contribution ① (Section 3): We present operational semantics for TinyWGSL, a fragment of WGSL that captures the language features that pertain to barrier divergence and uniformity analysis. Our semantics gives, for the first time, a rigorous definition of barrier divergence in the context of WGSL. It is informed by a careful study of barrier divergence in GPU programming languages that serve as compilation targets for WGSL and discussions with editors of the WGSL language specification and the engineering team behind the Chromium implementation of WebGPU.

Problem ❷: Uniformity analysis is lengthy and imprecise. Uniformity analysis contains clever ideas that are likely to be of interest to the PL community, but spans 27 pages in a standard print-to-PDF view of the online specification document [15, §15.2]. This makes it hard to understand these ideas precisely or get a bird's eye view of how the analysis works. The analysis is described in terms of a number of sub-analyses, but the manner in which the sub-analyses inter-operate is not always clear from the specification: we often found it necessary to refer to Tint [18], the Chromium implementation of uniformity analysis, to understand the intent of the specification.

Contribution ② (Section 4): We present formal rules that concisely and precisely describe uniformity analysis for TinyWGSL, and argue soundness with respect to our definition of barrier divergence. Our formulation highlights the *essence* of the analysis in a much more streamlined form compared with the full WGSL specification, which is necessarily complicated by many WGSL details that have little-to-no bearing on the core algorithm that drives the analysis.

Problem ❸: Uniformity analysis suffers from soundness and precision issues. Our careful study of the WGSL specification, viewing soundness and precision through the lens of our operational semantics, revealed (a) soundness problems arising due to edge cases missed by the analysis, specifically relating to the 'break-if' construct, and (b) opportunities for refining the analysis in simple ways that make it more precise.

Contribution ③ (Section 5): We summarise four significant changes (as well as various smaller fixes and clarifications) to the WGSL uniformity analysis that we have proposed, and discuss how we have worked with the WGSL specification editors to land them in an upcoming release.

Problem ❹: Implementations of uniformity analysis are hard to test. The WGSL specification uses a mixture of prose, mathematical notation and worked examples to define uniformity analysis, based on which it is hard to test the correctness of an implementation such as Tint.

Contribution ④ (Section 6): We present a lightweight mechanisation of our formulation of uniformity analysis using the Alloy language and analyzer, and explain how we have used Alloy's test case generation facilities to intensively cross-check the behaviour of Tint against our mechanisation. This process led to the finding and fixing of discrepancies between the implementation of the analysis in Tint and the analysis specification, as well as a front-end bug in Tint.

2 Background

2.1 GPU Compute Shaders

Using a GPU for general-purpose computation requires writing a *compute shader* in a *shading language* associated with a GPU programming model, e.g. HLSL [41] for Microsoft's Direct3D [40]; MSL [5] for Apple's Metal [4]; SPIR-V [22] for the Vulkan open standard [21]; and WGSL, the WebGPU shading language studied in this paper. These programming models also support *fragment* and *vertex* shaders for rendering, but we shall only be concerned with compute shaders.

Expressed in single-program multiple-data form, a shader specifies the behaviour of numerous threads[2] organised into one or more equal-sized *workgroups*. A thread has access to a *global ID*, unique to the thread; a *local ID*, unique within its workgroup; and a *workgroup ID*, common to

[2]Threads are called *invocations* in WGSL, but we use *thread* as it is more common in the GPU programming literature.

```
1  @workgroup_size(N)
2  fn main(lid: u32) {
3   for (var d: u32 = N;
4        d > 0;
5        d = d / 2) {
6    workgroupBarrier();
7    if (lid < d) {
8     A[lid] += A[lid + d];
9    }
10  }
11 }
```

(a) Correct use of a barrier to synchronise threads

```
@workgroup_size(N)
fn main(lid: u32) {
 for (var d: u32 = N;
      d > 0;
      d = d / 2) {
  if (lid < d) {
   workgroupBarrier();
   A[lid] += A[lid + d];
  }
 }
}
```

(b) Barrier divergence due to non-uniform conditional

```
@workgroup_size(N)
fn main(lid: u32) {
 for (var d: u32 = N;
      d > 0;
      d = d / 2) {
  workgroupBarrier();
  if (lid >= d) { continue;
      }
  A[lid] += A[lid + d];
 }
}
```

(c) Barrier divergence due to non-uniform continue

Fig. 1. A WGSL shader that uses N threads to perform a tree-style reduction on an array A of size 2N

all threads in a workgroup and unique to the workgroup. These allow distinct threads to take different control-flow paths through a shader and access distinct portions of memory. Without loss of generality we shall only be concerned with the local ID of a thread, which we denote `lid`.

Barriers. Synchronisation between threads in a workgroup is supported via *barriers*. On reaching a barrier, a thread waits until all threads in its workgroup have reached the barrier. Once all threads have arrived, the threads can proceed to execute beyond the barrier with the guarantee that any memory operations issued by threads prior to reaching the barrier have completed. This avoids data races between pre- and post-barrier memory accesses from distinct threads.

Figure 1a shows a WGSL shader for summing the elements of a floating-point array A using a tree-style reduction, where the workgroup size N is a power of two. For brevity, we omit the declaration of A, which should have size 2N. The *enabled* threads are those for which `lid` is less than loop variable d, which decreases exponentially so that each iteration disables half of the enabled threads. On each iteration, an enabled thread accumulates its array element `A[lid]` with another element `A[lid + d]`. The reduction result is ultimately accumulated in `A[0]`. The `workgroupBarrier()` call at line 6 ensures that all threads have finished updating A and have a consistent view of its contents. Without this barrier, it is possible for a thread to continue to the next loop iteration and read from a memory location that is being updated by another thread, resulting in a data-race.

Barrier divergence. Barriers are necessary to avoid data-races, but care must be taken to ensure their correct use. As an example, consider Figure 1b. This shader exhibits *barrier divergence*: the barrier has been moved to line 7 which is inside an `if` statement whose condition evaluates non-uniformly between threads because it depends on `lid`. As a result, some but not all threads in the workgroup will hit the barrier on each loop iteration. Informally, barrier divergence occurs when threads dynamically execute synchronisation barriers under divergent control flow. As discussed further in Section 2.3, traditional shading languages dismiss barrier divergence as a source of *undefined behaviour*, so that there are no guarantees on how a shader that exhibits barrier divergence might behave, with crashes and hangs being acceptable behaviours in practice.

A more subtle and surprising example is shown in Figure 1c. The `if` condition in Figure 1a has been inverted so that disabled threads immediately execute the `continue` at line 7, while enabled threads update A at line 8 after the `if` statement. Although this *looks* like a semantically-equivalent rewrite of Figure 1a, Figure 1c exhibits barrier divergence. When a thread executes a `continue` statement under divergent control flow, all remaining loop iterations are executed under divergent control flow, with threads only reconverging on loop exit. Treating Figure 1c as barrier-divergent may seem overly conservative, but we discuss why it is necessary to do so in Section 2.3.

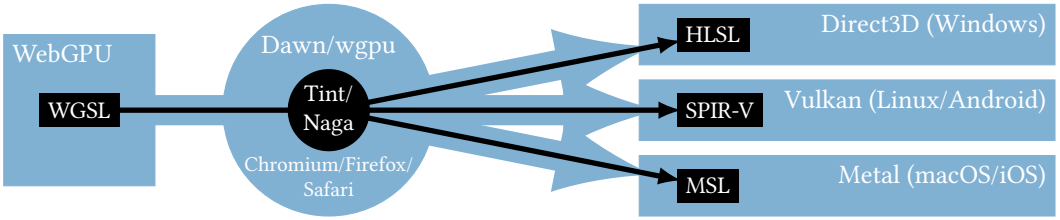


Fig. 2. A block diagram of how the WebGPU API is implemented. Blue relates to APIs and their implementation. Black relates to the languages used within these APIs and their compilation.

2.2 WebGPU, WGSL and Tint

A WebGPU implementation in a browser builds on existing native graphics APIs as shown in Figure 2. On Windows, WebGPU maps to Direct3D; on Android and Linux, to Vulkan; and on Apple platforms, to Metal. This layered design presents a unified API to web developers: the browser translates JavaScript-level WebGPU commands into corresponding downstream API operations. The WebGPU Shading Language (WGSL) allows developers to write shaders in a format that is independent from the downstream API. WGSL shaders are not compiled directly to GPU machine code. Instead, the browser compiles WGSL into the shading language appropriate for the target backend—HLSL (Direct3D), SPIR-V (Vulkan) and MSL (Metal)—before the final GPU-specific compilation step.

In Chromium-based browsers (e.g. Chrome, Edge), WebGPU is implemented as a component called Dawn [17], which has a WGSL compiler, Tint [18]. Firefox implements WebGPU via the wgpu project [49], which has a WGSL compiler, Naga [48], and Safari is based on WebKit [54], which has a WGSL shader compilation module [55]. Uniformity analysis is defined in the WGSL specification [15, §15.2] and implemented by the in-browser shader compiler. Currently only Tint features a mature uniformity analysis implementation, hence we focus our automated testing efforts (Section 6) on Tint, leaving testing of other implementations as future work once they mature.

2.3 Barrier Divergence in Downstream Shading Languages

Each of the downstream shading languages that WGSL targets (Section 2.2) has its own definition of barrier divergence. The WGSL uniformity analysis has been designed to reject shaders that would exhibit barrier divergence in *any* of these languages if lowered straightforwardly. Therefore, a rigorous definition of barrier divergence for a WGSL-based language (Section 3) must be at least as conservative as the most conservative downstream language. Although this requirement reduces precision for backends that could be treated less conservatively, it is a real-world constraint that must be respected. We thus review what these downstream languages say about barrier divergence.

2.3.1 SPIR-V. In SPIR-V a thread is called an *invocation*, and a barrier is offered via the instruction `OpControlBarrier` [22, §3.3.20]. In relation to `OpControlBarrier` the specification says:

> "behavior is undefined unless all invocations [...] execute the same dynamic instance of this instruction." [22, §3.3.20]

The notion of dynamic instances allows for distinguishing between an instruction that (a) has been reached by threads via divergent control flow, or (b) is executed multiple times due to loops or multiple function call sites [22, §2.2.5]. At the entry to a shader, threads begin executing the same dynamic instance of instructions. When threads diverge by taking different branches of a conditional, they execute different dynamic instances of instructions going forward [22, §2.2.5]. Threads will only reconverge and continue executing the same dynamic instance of instructions

after a conditional or loop if they (a) were executing the same dynamic instance of instructions before the conditional or loop and (b) they reach the end of the conditional or loop [22, §2.2.5].

It can now be seen that a natural lowering of Figure 1b to SPIR-V would exhibit barrier divergence: on the first loop iteration all threads would reach the `OpControlBarrier` associated with the barrier at line 7 and on the second iteration only threads with IDs less than `N / 2` would reach this barrier.

Due to the conditions on control flow reconvergence, Figure 1c must also be considered barrier-divergent if it is to be straightforwardly lowered to SPIR-V. Since the `if` statement at line 7 causes control flow to diverge and threads do not reach the end of the conditional, instead iterating the loop via the `continue`, control flow is not guaranteed to reconverge until all threads exit the loop. Therefore, some threads will execute different dynamic instances of the `OpControlBarrier` corresponding to line 6 on subsequent iterations, leading to barrier divergence.

2.3.2 MSL. The Metal Shading Language calls a workgroup a *threadgroup*, and offers a workgroup barrier via the `threadgroup_barrier` command. The language specification states:

> "If `threadgroup_barrier` is inside a conditional statement and if any thread enters the conditional statement and executes the barrier function, then all threads in the threadgroup need to enter the conditional and execute the barrier function. If `threadgroup_barrier` is inside a loop, for each iteration of the loop, all threads in the threadgroup need to execute the barrier function before any threads continue execution beyond the barrier function." [5, §6.9.1]

This makes it clear that a natural lowering of Figure 1b to MSL would lead to barrier divergence, but suggests that a natural lowering of Figure 1c would not, i.e. certain WGSL shaders must be considered barrier divergent to meet the restrictions of SPIR-V (Section 2.3.1) even though they would execute unproblematically if compiled to MSL.

2.3.3 HLSL. HLSL is the least well-specified of the languages to which WGSL is compiled. Although an official specification for HLSL is under development [41], it does not yet cover the use of barriers, which are offered by the `AllMemoryBarrierWithGroupSync` command. In HLSL a workgroup is called a *thread group*. Documentation on `AllMemoryBarrierWithGroupSync` states:

> "The behavior of calls to this function that are within diverging branches of a thread group are undefined." [39]

While it is obvious that "diverging branches of a thread group" must refer to threads in a group having taken different control flow paths according to their thread IDs (so that a natural lowering of Figure 1b to HLSL would be considered barrier-divergent), we are not aware of documentation that makes this terminology precise, and it is thus unclear whether a natural lowering of Figure 1c to HLSL should be considered barrier-divergent.

3 Rigorously Defining Barrier Divergence

As per Problem ❶ in the introduction, the WGSL uniformity analysis aims to reject programs that would result in barrier divergence if executed, yet there is currently no definition of barrier divergence for WGSL. To place the analysis on a formal footing and argue for its soundness, it is necessary to have a definition of barrier divergence against which to compare the analysis.

For this purpose, we define TinyWGSL, a minimal subset of WGSL that contains only those constructs related to the essence of barrier divergence and uniformity analysis—namely the interaction between barriers and control flow. Constructs that are not core to uniformity and can easily be added to TinyWGSL, such as switch statements and short circuiting binary operators, are omitted. Informed by the definitions of barrier divergence in downstream shading languages (Section 2.3), conversations with the editors of the WGSL specification [9] and the developers of Tint (the WGSL

$$
\begin{array}{rcl}
\langle \mathit{Shader} \rangle & ::= & \langle \mathit{Func} \rangle^{*}\ \texttt{@workgroup_size}(\langle \mathit{Int} \rangle)\ \texttt{fn main(lid) \{}\ \langle \mathit{Stmt} \rangle\ \text{List}\ \texttt{\}} \\
\langle \mathit{Func} \rangle & ::= & \texttt{fn}\ \langle \mathit{FnIdent} \rangle(\langle \mathit{LocIdent} \rangle, \ldots, \langle \mathit{LocIdent} \rangle)\ \texttt{\{}\ \langle \mathit{Stmt} \rangle\ \text{List}\ \texttt{\}} \\
\langle \mathit{Stmt} \rangle & ::= & \langle \mathit{LocIdent} \rangle\ \texttt{=}\ \langle \mathit{Expr} \rangle \ \mid\ \langle \mathit{SharedIdent} \rangle\texttt{[}\langle \mathit{Expr} \rangle\texttt{]}\ \texttt{=}\ \langle \mathit{Expr} \rangle \\
& \mid & \texttt{loop}^{\text{ID}}\texttt{\{}\langle \mathit{Stmt} \rangle\ \text{List}\ \texttt{continuing}^{\text{ID}}\texttt{\{}\langle \mathit{Stmt} \rangle\ \text{List}\texttt{\}\}} \\
& \mid & \texttt{continuing}^{\text{ID}}\texttt{\{}\langle \mathit{Stmt} \rangle\ \text{List}\texttt{\}} \\
& \mid & \texttt{if (}\langle \mathit{Expr} \rangle\texttt{) \{}\langle \mathit{Stmt} \rangle\ \text{List}\texttt{\}\ else\ \{}\langle \mathit{Stmt} \rangle\ \text{List}\texttt{\}} \\
& \mid & \langle \mathit{LocIdent} \rangle\ \texttt{=}\ \langle \mathit{FnIdent} \rangle^{\text{ID}}\texttt{(}\langle \mathit{Expr} \rangle, \ldots, \langle \mathit{Expr} \rangle\texttt{)} \\
& \mid & \texttt{barrier}^{\text{ID}} \ \mid\ \texttt{continue}^{\text{ID}} \ \mid\ \texttt{break} \ \mid\ \texttt{return}\ \langle \mathit{Expr} \rangle \\
\langle \mathit{Expr} \rangle & ::= & \langle \mathit{Val} \rangle \ \mid\ \langle \mathit{LocIdent} \rangle \ \mid\ \langle \mathit{SharedIdent} \rangle\texttt{[}\langle \mathit{Expr} \rangle\texttt{]} \ \mid\ \langle \mathit{Expr} \rangle\ \langle \mathit{Op} \rangle\ \langle \mathit{Expr} \rangle
\end{array}
$$

Fig. 4. Syntax for TinyWGSL

compiler in Chromium) [7], and prior work on semantics for GPU languages [14], we present an operational semantics for TinyWGSL that rigorously defines the conditions for barrier divergence.

In line with the full WGSL uniformity analysis, we assume that shaders do not exhibit dynamic errors such as data races, buffer overflows and type-conversion errors on the basis that shaders containing dynamic errors are already erroneous [15, §15.2.2]. WGSL instead defines a set of allowed, safe behaviours for these dynamic errors [36]. In particular, the assumption of data-race freedom allows our operational semantics to assume a sequentially consistent memory model.

3.1 Syntax of TinyWGSL

Figure 4 shows the TinyWGSL syntax. Various statements have associated IDs that are used by the semantics of Section 3.2; we omit them when they are not relevant to our discussion. Finite, disjoint sets *LocIdent*, *SharedIdent* and *FnIdent* provide names for local and shared variables and functions. The sets *Val* and *Op* of data values and operators are left abstract except we require that *Val* contains truth values $\top$ and $\bot$, that $\{0, \ldots, 256\} \subset \mathit{Val}$ to represent thread IDs up to WebGPU's maximum workgroup dimension [58, §3.6.2], and that the operators in *Op* are deterministic.

```
1  @workgroup_size(256)
2  fn main(lid) {
3    d = 256
4    loop[1] {
5      if (d <= 0) {
6        break
7      } else { }
8      barrier[2]
9      if (lid >= d) {
10       continue[3]
11     } else { }
12     A[lid] = A[lid] + A[lid + d]
13     continue[4]
14     continuing[1] {
15       d = d / 2
16     }
17   }
18   barrier[5]
19 }
```

Fig. 3. The TinyWGSL equivalent of Figure 1c for 256 threads

A shader comprises zero or more function declarations, a work-group size specifying the number of threads that will execute the shader, and a `main` function where threads commence execution, with parameter `lid` $\in$ *LocIdent* providing a thread's unique ID.

We use '$\langle \mathit{Stmt} \rangle$ List' to denote a list of statements. Local variables hold values in *Val* and shared variables hold arrays of values in *Val*, hence local variables are updated via direct assignment and shared variables via an indexed assignment. All non-main TinyWGSL functions are assumed to return a value and thus all function calls appear as the right hand side of an assignment. The full list of requirements for a TinyWGSL shader to be well-formed is given in Section 3.2.

Figure 3 shows the TinyWGSL representation of Figure 1c, showcasing a key difference between WGSL and other shading languages: the `loop` statement [15, §9.4.3], of the form `loop{`ss_1 `continuing{`ss_2`}}`. Here ss_1 is the body of the loop. The *continuing statement* `continuing{`ss_2`}` generalises the

update component of a for-loop: a (top-level) `continue` in ss_1 causes control flow to jump to ss_2, which is executed before starting the next loop iteration. Full WGSL includes `for` and `while` loops as syntactic sugar, defining them via a de-sugaring to the more general `loop` form, thus we omit them from TinyWGSL. In Figure 3, the for-loop of Figure 1c has been de-sugared so that initialisation of d comes before the `loop`, the break condition is represented by the `if` statement at the beginning of the loop body, and the update expression makes up the body of the `continuing` statement. The `continuing` statement is a syntactic construct that must not appear directly in TinyWGSL shaders but that is used in the semantics of Section 3.2 during the evaluation of loops.

To define barrier divergence, it must be possible to distinguish different syntactic barriers, continues, and function calls, therefore, each has a unique ID.

3.2 Operational Semantics for TinyWGSL

We now present a non-deterministic, thread-interleaving semantics that rigorously defines barrier divergence for TinyWGSL. We define the states on which the semantics operates, then describe thread-local statement execution rules, and rules for thread interleaving and barrier synchronisation.

States. A *global state* has the form $\langle \sigma, (T_0, \ldots, T_{N-1}) \rangle$. Here $\sigma : (\mathit{SharedIdent} \times \mathit{Val}) \to \mathit{Val}$ is a *shared store* accessible to all threads—a total map from pairs of shared variables and values to values. The T_i are *thread states*. A thread state has the form (τ, κ, ss) where $\tau : \mathit{LocIdent} \to \mathit{Val}$ is a *local store*—a total map from local identifiers to values, κ is a *control stack* and ss a sequence of statements. We use $\mathcal{E}$ to denote a special global state reached only when barrier divergence occurs.

The control stack supports calling and returning from functions, and tracks information about the function calls and loop iterations that a thread has executed in a manner that allows barrier divergence to be detected. It is informed by the definition of barrier divergence in SPIR-V (see Section 2.3.1), where dynamic instances of a statement are distinguished by different function call site chains and the traversal of loop iterations via distinct continue statements. Thus, a control stack may contain two types of elements: (a) a *function call element* (ID, x, ss) comprising the ID of a function call site, the call site variable x into which the returned value should be stored, and the list of statements ss to be executed after the call returns; and (b) a *loop element* (l, cs) comprising a loop ID and a list of the IDs of the $\texttt{continue}^{\text{ID}}$ statements executed so far for which l is the closest enclosing loop. As we shall see, the control stack is key to detecting barrier divergence because differences in control stacks indicate when threads have reached the same barrier via different function call paths or by iterating enclosing loops using different `continue` statements.

Notation. Variable lookup in the shared store is denoted $\sigma(x, i)$ where $x \in \mathit{SharedIdent}$ is an identifier and $i \in \mathit{Val}$ an index. The shared store update $\sigma[(x, i) \mapsto v]$ denotes the store such that $\sigma[(x, i) \mapsto v](x, i) = v$ and $\sigma[(x, i) \mapsto v](y, j) = \sigma(y, j)$ for all $y \neq x$ or $j \neq i$. Local store update is defined similarly. The result of evaluating an expression e in the context of shared and local stores σ and τ is denoted $(\sigma, \tau)(e) \in \mathit{Val}$. Since we leave *Val* and *Op* abstract, we do not concretely define the workings of expression evaluation, which are orthogonal to defining barrier divergence.

We use Π to denote a *function table* that maps each function name to a pair comprising a list of parameter names and list of statements representing the function body.

The empty control stack is represented by [] and the cons notation $e : \kappa$ denotes pushing e to the head of κ. The cons notation is also used when working with lists of statements, as is $+\!\!+$ which denotes list concatenation. The function 'head' is defined to get the head of a list in the usual way.

Definition 3.1 (Initial state). Let P be a TinyWGSL shader where ss is the body of `main` and N is the workgroup size declared in P. An *initial state* for P is a global state $\langle \sigma, (T_0, \ldots, T_{N-1}) \rangle$ where for each $0 \leq i < N$, $T_i = (\tau_i, [], ss)$, where $\tau_i(\texttt{lid}) = i$ and $\tau_i(x) = d \in \mathit{Val}$ for each $x \in (\mathit{LocIdent} \setminus \{\texttt{lid}\})$.

That is, in an initial state the shared store contains arbitrary values, the control stacks are all empty, every thread is poised to execute the body of `main`, and every local variable maps to some fixed default initial value d, except `lid` which maps to the ID of each thread.

$$\frac{v = (\sigma, \tau)(e)}{\langle \sigma, \tau, \kappa, x \ = \ e : ss\rangle \xrightarrow{s} \langle \sigma, \tau[x \mapsto v], \kappa, ss\rangle}\ \textsc{Local-Assign}$$

$$\frac{v_1 = (\sigma, \tau)(e_1) \qquad v_2 = (\sigma, \tau)(e_2)}{\langle \sigma, \tau, \kappa, x[e_1] \ = \ e_2 : ss\rangle \xrightarrow{s} \langle \sigma[(x, v_1) \mapsto v_2], \tau, \kappa, ss\rangle}\ \textsc{Shared-Assign}$$

$$\frac{(\sigma, \tau)(e) = \top}{\langle \sigma, \tau, \kappa, \texttt{if}\ (e)\ \{ss_1\}\ \texttt{else}\ \{ss_2\} : ss_3\rangle \xrightarrow{s} \langle \sigma, \tau, \kappa, ss_1 \mathbin{+\!\!+} ss_3\rangle}\ \textsc{If-True}$$

$$\frac{(\sigma, \tau)(e) = \bot}{\langle \sigma, \tau, \kappa, \texttt{if}\ (e)\ \{ss_1\}\ \texttt{else}\ \{ss_2\} : ss_3\rangle \xrightarrow{s} \langle \sigma, \tau, \kappa, ss_2 \mathbin{+\!\!+} ss_3\rangle}\ \textsc{If-False}$$

$$\frac{\mathrm{head}(\kappa) \neq (l, _)}{\langle \sigma, \tau, \kappa, \texttt{loop}^{l}\ \{\ldots\} : ss\rangle \xrightarrow{s} \langle \sigma, \tau, (l, [\,]) : \kappa, \texttt{loop}^{l}\ \{\ldots\} : ss\rangle}\ \textsc{Loop-Enter}$$

$$\frac{\mathrm{head}(\kappa) = (l, _)}{\langle \sigma, \tau, \kappa, \texttt{loop}^{l}\{ss_1\ \texttt{continuing}^{l}\{ss_2\}\} : ss_3\rangle \xrightarrow{s} \langle \sigma, \tau, \kappa, ss_1 \mathbin{+\!\!+} \underline{\texttt{continuing}^{l}\{ss_2\}} : \texttt{loop}^{l}\{ss_1\ \texttt{continuing}^{l}\{ss_2\}\} : ss_3\rangle}\ \textsc{Loop-Unfold}$$

$$\frac{}{\langle \sigma, \tau, (l, _) : \kappa, \texttt{break} : ss_1 \mathbin{+\!\!+} \texttt{loop}^{l}\ \{\ldots\} : ss_2\rangle \xrightarrow{s} \langle \sigma, \tau, \kappa, ss_2\rangle}\ \textsc{Break}$$

$$\frac{}{\langle \sigma, \tau, (l, cs) : \kappa, \texttt{continue}^{c} : ss_1 \mathbin{+\!\!+} \underline{\texttt{continuing}^{l}\{ss_2\}} \ : ss_3\rangle \xrightarrow{s} \langle \sigma, \tau, (l, c : cs) : \kappa, ss_2 \mathbin{+\!\!+} ss_3\rangle}\ \textsc{Continue}$$

$$\frac{\begin{array}{c}\Pi(f) = \langle (x_1, \ldots, x_n), ss_1\rangle \\ v_1 = (\sigma, \tau)(e_1) \ \ldots \ v_n = (\sigma, \tau)(e_n) \qquad \tau' = \tau[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n] \qquad \kappa' = (id, x, ss_2) : \kappa\end{array}}{\langle \sigma, \tau, \kappa, x \ = \ f^{id}(e_1, \ldots, e_n) : ss_2\rangle \xrightarrow{s} \langle \sigma, \tau', \kappa', ss_1\rangle}\ \textsc{Call}$$

$$\frac{\kappa = \kappa_1 \mathbin{+\!\!+} (id, x, ss_1) : \kappa_2 \qquad \kappa_1 \text{ contains no function elements} \qquad v = (\sigma, \tau)(e)}{\langle \sigma, \tau, \kappa, \texttt{return}\ e : ss_2\rangle \xrightarrow{s} \langle \sigma, \tau[x \mapsto v], \kappa_2, ss_1\rangle}\ \textsc{Return}$$

Fig. 5. Rules defining the $\xrightarrow{s}$ relation for execution of TinyWGSL statements, where Π is the function table

Statements. The rules of Figure 5 define a relation $\xrightarrow{s}$ describing the execution of a non-barrier statement by a thread. The relation describes how a shared store σ and thread state (τ, κ, ss) transitions to an updated shared store σ' and thread state (τ', κ', ss') via a thread executing a single statement. Recall that Π, used in the Call rule, denotes the function table.

As well as the typical programming language well-formedness requirements, the rules of Figure 5 require that a TinyWGSL shader satisfies the following restrictions that are either trivial to enforce or adopted from full WGSL:

(1) The `main` function must not contain any **`return`** statements and must end in a **`barrier`**. (This avoids the need to account for a special case where some threads arrive at a barrier and others terminate execution without executing a final barrier.)

$$\frac{0 \le i < N \qquad T_i = \langle \tau, \kappa, ss \rangle \qquad \langle \sigma, \tau, \kappa, ss \rangle \xrightarrow{s} \langle \sigma', \tau', \kappa', ss' \rangle}{\langle \sigma, (T_0, \ldots, T_{N-1}) \rangle \xrightarrow{t} \langle \sigma', (T_0, \ldots, T_{i-1}, (\tau', \kappa', ss'), T_{i+1}, \ldots, T_{N-1}) \rangle} \textsc{Thread}$$

$$\frac{\forall 0 \le i < N \,.\, T_i = (\tau_i, \kappa, \texttt{barrier}^b : ss)}{\langle \sigma, (T_0, \ldots, T_{N-1}) \rangle \xrightarrow{t} \langle \sigma, ((\tau_0, \kappa, ss), \ldots, (\tau_{N-1}, \kappa, ss)) \rangle} \textsc{Barrier-Sync}$$

$$\frac{0 \le i, j < N \qquad T_i = (\tau_i, \kappa_i, \texttt{barrier}^{b_i} : ss_i) \qquad T_j = (\tau_j, \kappa_j, \texttt{barrier}^{b_j} : ss_j) \qquad b_i \neq b_j \vee \kappa_i \neq \kappa_j}{\langle \sigma, (T_0, \ldots, T_{N-1}) \rangle \xrightarrow{t} \mathcal{E}} \textsc{Barrier-Error}$$

Fig. 6. Rules defining the $\xrightarrow{t}$ relation for interleaving, synchronisation and termination of TinyWGSL threads, and the Barrier-Error rule that detects barrier divergence

(2) The local identifiers used by distinct functions, including parameter names, are disjoint.
(3) For every `loop`$\{ss_1$ `continuing`$\{ss_2\}\}$, ss_1 ends with a **`continue`**.
(4) The IDs of **`barrier`**, **`continue`**, `loop` and function call statements are all distinct.
(5) The ID of each `continuing` statement matches the ID of its associated `loop`.
(6) Every function is declared before all of its callers. (This reflects the fact that WGSL does not allow recursion.)
(7) The final statement in the body of a `continuing` statement may optionally be `if (`e`) {`**`break`**`} else {}`, but all other **`break`** and **`continue`** statements in the `continuing` must appear inside a nested loop. Additionally, the body of a `continuing` statement cannot contain any **`return`** statements. This restriction is inherited from WGSL [15, §9.4].

The Local-Assign, Shared-Assign and If-True/False rules are standard.

Entry to a loop is handled by Loop-Enter: the premise demands that the top of the control stack does *not* represent a loop with ID l (the ID of the loop to be entered). A new loop element is added to the top of the control stack, with an initially empty sequence of associated continue IDs.

The Loop-Unfold rule describes how to execute a loop when the top of the control stack *does* match the loop to be executed—either because the loop was just entered via Loop-Enter, or because a subsequent iteration of a loop for which execution is already underway has been reached. The loop body is unfolded, with the `continuing` component being replaced with the special `continuing` statement used to handle execution of **`continue`** statements.

Because a **`break`** statement must always occur in a loop, when a **`break`** is executed the top of the control stack must be a loop element $(l, _)$ where l is the ID of the innermost loop being executed. The Break rule removes the loop element from the top of the call stack and throws away all statements up to and including the associated loop construct with the ID l.

The Continue rule causes execution to jump to the body of the continuing statement for the current loop, captured via the special `continuing` statement. By construction (thanks to the Loop-Unfold rule), ss_3 is guaranteed to start with a `loop` statement representing further iterations of the loop. The unique ID of the **`continue`** statement is added to the sequence of continue IDs associated with the loop at the top of the control stack.

The Call rule binds a function's parameter names to evaluated argument values in the local store. To allow execution to continue after the call returns, the statements following the call are added to the control stack along with the local variable that will store the returned value, and the unique ID of the call. Execution then continues on the body of the called function.

The Return rule evaluates the result expression e, and based on the function element closest to the head of the control stack, binds the result to variable x and continues executing statements ss_1.

Threads. The rules of Figure 6 define a relation $\xrightarrow{t}$ over global states describing the interleaving, synchronisation and termination of threads, as well as the detection of barrier divergence.

The non-deterministic Thread rule allows an arbitrary thread i to take a step via the $\xrightarrow{s}$ relation, resulting in an update to both the shared store and the local state of thread i.

The Barrier-Sync rule requires that all threads are at the same `barrier` statement with identical control stacks. The identical control stacks mean that all threads have arrived at the barrier under *convergent* control flow, thus all can proceed execution of the statements that follow the barrier.

Finally, the Barrier-Error rule requires that there are (at least) two threads that are at *some* `barrier` statements, and that the threads have different control flow stacks or disagree on the barrier that has been reached. Execution transitions to the special error state $\mathcal{E}$.

We can now formally define what it means for a TinyWGSL shader to exhibit barrier divergence:

Definition 3.2 (Barrier divergence). A TinyWGSL shader P exhibits barrier divergence if there exists a sequence of global states $G_0, G_1, \ldots, G_M$ such that G_0 is an initial state for P (Definition 3.1), $G_i \xrightarrow{t} G_{i+1}$ $(0 \leq i < M)$ and $G_M = \mathcal{E}$.

4 Formalising Uniformity Analysis for TinyWGSL

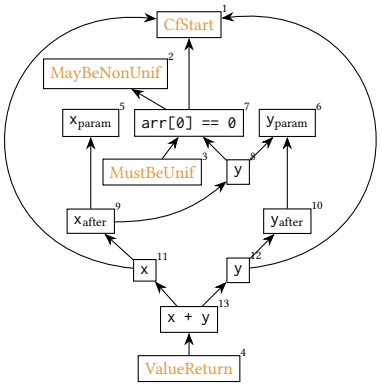
As per Problem ❷ in the introduction, the length and informality of uniformity analysis in the WGSL specification obscures the key ideas behind the analysis. After an overview of how uniformity analysis works (Section 4.1), we present a precise formulation of the analysis for TinyWGSL that is concise compared with the 27-page exposition in the specification, and argue that it is sound with respect to the semantics of Section 3 (Section 4.2). Our formulation is based on meticulous study of the WGSL specification and numerous discussions with the specification editors [9], and has been mechanised using Alloy and rigorously cross-checked against Tint (Section 6). Our presentation incorporates fixes for various deficiencies in the specification that we discuss in Section 5.

4.1 Example-Driven Overview

Uniformity analysis builds a *uniformity graph* for each function in a program: a directed graph whose nodes represent expressions and program locations, and whose edges capture uniformity dependencies: an edge (v_1, v_2) indicates that the uniformity of v_1 depends on the uniformity of v_2; e.g. v_1 could be the 'then' branch of an `if` statement and v_2 the condition of the `if` statement. Special node MayBeNonUnif represents a source of potential non-uniformity, such as a use of `lid` (definitely non-uniform) or a value read from shared memory (conservatively assumed to be potentially non-uniform). Intuitively, any node whose uniformity transitively depends on MayBeNonUnif is regarded as being *tainted* by non-uniformity. Node MustBeUnif represents a uniformity *requirement* and is connected to any node that must be uniform due to the use of a barrier. The aim of the analysis is to construct the uniformity graph of a function such that the set of expressions and program locations tainted as non-uniform soundly over-approximates the set of expressions and program locations that can dynamically be non-uniform. Therefore, if the function could exhibit barrier divergence, the graph will contain a path from MustBeUnif to MayBeNonUnif. Furthermore, the analysis constructs a *function summary* characterising (a) whether the function must only be called under uniform control flow, (b) which function arguments must be passed uniform values, (c) on which function arguments the returned value of the function depends, and (d) whether the returned value is non-uniform.

```
1 fn f(x, y) { // M(x) = 5, M(y) = 6
2   if (arr[0] == 0) {
3     barrier  // M(x) = 5, M(y) = 6
4   } else {
5     x = y    // M(x) = 8, M(y) = 6
6   }          // M(x) = 9, M(y) = 10
7   return x + y
8 }
```

(a) Example TinyWGSL function



(b) Associated uniformity graph

Fig. 7. A TinyWGSL shader and its associated uniformity graph.

Figure 7 shows the uniformity graph for a simple function. Nodes are numbered in order of addition to the graph. The analysis uses a map M to associate each (local variable, program point) pair with a node representing the uniformity of the variable at the program point.

First, nodes CfStart, MayBeNonUnif, MustBeUnif and ValueReturn are added: CfStart represents uniformity of control flow on function entry; MustBeUnif and MayBeNonUnif are as above; ValueReturn represents the uniformity of the value returned from the function. Nodes $\mathsf{x}_{\text{param}}$ and $\mathsf{y}_{\text{param}}$ are then added to represent uniformity of arguments to `f`, and M records that the uniformity of `x` and `y` on function entry is represented by these nodes (Figure 7a line 1).

Analysis of the body of `f` begins at the `if` statement. The analysis produces node 7 to represent control flow uniformity in each branch of the `if`. The uniformity of node 7 depends on both control flow uniformity before the `if` statement (via an edge to node 1), and the uniformity of the branch condition. The edge from `arr[0] == 0` to MayBeNonUnif means that node 7 is conservatively tainted as non-uniform.

Since a `barrier` must be executed in uniform control flow, analysis of the 'then' branch results in an edge from MustBeUnif to node `arr[0] == 0` to indicate that control flow must be uniform.

After assignment `x = y`, the uniformity of `x` and `y` are the same, hence $M(\mathsf{x})$ is updated to node 8 (Figure 7a line 5) produced by the analysis of expression `y`.

After the `if`, the values of `x` and `y` may have come from either of the 'then' or 'else' branches. To *merge* the uniformity of `x` and `y`, nodes $\mathsf{x}_{\text{after}}$ and $\mathsf{y}_{\text{after}}$ are created. In the style of a reaching definitions analysis, the map M is consulted to connect $\mathsf{x}_{\text{after}}$ and $\mathsf{y}_{\text{after}}$ to nodes representing uniformity of `x` and `y` at the end of each branch of the `if` (Figure 7a lines 3 and 5). Merging ignores `if` branches that cannot reach the end of the statement due to breaks, continues or returns. It can be seen that $\mathsf{x}_{\text{after}}$ is tainted as non-uniform due to the path $9 \longrightarrow 8 \longrightarrow 7 \longrightarrow 2$, reflecting the fact that `x` is assigned to under non-uniform control flow.

Analysis of `return x + y` results in nodes 11–13. An edge from ValueReturn to node 13 records that uniformity of the returned value depends on that of `x + y`.

From Figure 7b we can see that `f` fails uniformity analysis because there is a path from MustBeUnif to MayBeNonUnif. The graph also records that: (a) `f` must be called from uniform control, due to the path from MustBeUnif to CfStart; (b) non-uniform values can be passed for `x` and `y` because there is no path from MustBeUnif to $\mathsf{x}_{\text{param}}$ or $\mathsf{y}_{\text{param}}$, and (c) the value returned by `f` may be non-uniform, due to a path from ValueReturn to MayBeNonUnif. For functions that do not fail the analysis, this information forms a function summary used to analyse function calls.

The full WGSL uniformity analysis, and thus our formulation, is deliberately value-insensitive to make shader rejection more predictable; a value-sensitive analysis could be more precise, but its results would depend on the precision of the underlying value analysis. This can lead obviously uniform expressions to be soundly considered possibly non-uniform: `lid == lid` is treated as possibly non-uniform even though it will evaluate to $\top$ in all threads. The analysis is therefore incomplete; the statement `if (lid == lid) {barrier} else {barrier}` does not dynamically exhibit barrier

divergence—all threads take the 'then' branch—but it is soundly rejected by uniformity analysis. For the same reason, the statement `if (false) {if (lid > 0) {barrier} else {}} else {}` will be rejected even though the `barrier` is evidently unreachable.

4.2 Uniformity Analysis

Notation. Let *UnifNode* be a countably infinite set of *uniformity nodes*, containing special elements MayBeNonUnif, MustBeUnif, CfStart and ValueReturn. A *uniformity graph* is a directed graph (V, E), where $V \subset \mathit{UnifNode}$ is a finite set of nodes and $E \subseteq V \times V$ a set of edges. To relate the uniformity of local variables to graph nodes, the analysis uses a *variable uniformity map* M from local identifiers to uniformity nodes. The rules below use 'v fresh' to obtain a fresh node from *UnifNode*. Throughout, N denotes $|\mathit{LocIdent}|$, the number of local variables.

$$
\begin{aligned}
&\mathcal{B}([]) = \{\text{Next}\}\\
&\mathcal{B}(s : ss) = \textbf{if } \text{Next} \in \mathcal{B}(s)\\
&\qquad \textbf{then } (\mathcal{B}(s) \setminus \{\text{Next}\}) \cup \mathcal{B}(ss)\\
&\qquad \textbf{else } \mathcal{B}(s)\\
&\mathcal{B}(x \texttt{ = } e) = \{\text{Next}\}\\
&\mathcal{B}(\texttt{barrier}^b) = \{\text{Next}\}\\
&\mathcal{B}(x \texttt{ = } f^{id}(e_1, \ldots, e_n)) = \{\text{Next}\}\\
&\mathcal{B}(\texttt{break}) = \{\text{Break}\}\\
&\mathcal{B}(\texttt{continue}^l) = \{\text{Continue}\}\\
&\mathcal{B}(\texttt{return } e) = \{\text{Return}\}\\
&\mathcal{B}(\texttt{if } (e) \ \{ss_1\} \texttt{ else } \{ss_2\}) = \mathcal{B}(ss_1) \cup \mathcal{B}(ss_2)\\
&\mathcal{B}(\texttt{loop}^l\{ss_1 \texttt{ continuing}^l\{ss_2\}\}) =\\
&\quad \textbf{let } B_1 = \mathcal{B}(ss_1), B_2 = \mathcal{B}(ss_2) \textbf{ in}\\
&\quad \textbf{if } B_1 = \{\text{Return}\} \textbf{ then } B_1\\
&\quad \textbf{else if } \{\text{Break}\} \notin B_1\\
&\qquad \textbf{then } (B_1 \cup B_2) \setminus \{\text{Continue}, \text{Next}\}\\
&\quad \textbf{else } (B_1 \cup B_2 \cup \{\text{Next}\}) \setminus \{\text{Break}, \text{Continue}\}
\end{aligned}
$$

Fig. 8. Statement behaviour analysis

Statement behaviour analysis. Uniformity analysis relies on *statement behaviour analysis* [15, §9.7] to conservatively determine whether statements are reachable such that unreachable statements are not analysed. Figure 8 gives statement behaviour analysis rules for TinyWGSL via an overloaded function $\mathcal{B}$ that maps a list of statements, or a single statement, to a set of possible *behaviours* drawn from {Return, Break, Continue, Next} representing what the statement(s) *may* do. For example, $\mathcal{B}(\texttt{if } (e_1) \ \{\texttt{return } e_2\} \texttt{ else } \{\}) = \{\text{Return}, \text{Next}\}$.

We focus on the rule for loops (the other rules are straightforward), which involves three cases. If the loop body behaviour is {Return}—the loop unconditionally returns—the loop has behaviour {Return}. Otherwise, conceptually, the behaviour of a loop is the behaviour of its body with Continue removed (a continue cannot escape a loop), and Break replaced with Next (a break causes execution to proceed beyond the loop). The second case deals with a loop that cannot break, so execution never proceeds beyond the loop. The third case handles the replacement of Break with Next.

$$
\begin{aligned}
&\mathcal{UG}_{\text{expr}}(V, E, M, cf, \texttt{lid}) = (V, E, \text{MayBeNonUnif})\\
&\mathcal{UG}_{\text{expr}}(V, E, M, cf, x\texttt{[}e\texttt{]}) = (V, E, \text{MayBeNonUnif})\\
&\mathcal{UG}_{\text{expr}}(V, E, M, cf, l) \text{ where } l \in \mathit{Val} = (V, E, cf)\\
&\mathcal{UG}_{\text{expr}}(V, E, M, cf, x) \text{ where } \texttt{lid} \neq x \in \mathit{LocIdent} =\\
&\quad \textbf{let } v_{\text{res}} \text{ fresh } \textbf{in } (V \cup \{v_{\text{res}}\}, E \cup \{(v_{\text{res}}, cf), (v_{\text{res}}, M(x))\}, v_{\text{res}})\\
&\mathcal{UG}_{\text{expr}}(V, E, M, cf, e_1 \ op \ e_2) =\\
&\quad \textbf{let } (V_1, E_1, v_1) = \mathcal{UG}_{\text{expr}}(V, E, M, cf, e_1) \textbf{ in}\\
&\quad \textbf{let } (V_2, E_2, v_2) = \mathcal{UG}_{\text{expr}}(V_1, E_1, M, cf, e_2) \textbf{ in}\\
&\quad \textbf{let } v_{\text{res}} \text{ fresh } \textbf{in } (V_2 \cup \{v_{\text{res}}\}, E_2 \cup \{(v_{\text{res}}, v_1), (v_{\text{res}}, v_2)\}, v_{\text{res}})
\end{aligned}
$$

Fig. 9. Uniformity analysis of expressions

Analysis of expressions. The $\mathcal{UG}_{\text{expr}}$ function of Figure 9 takes a uniformity graph (V, E), variable uniformity map M, control flow node $cf \in V$, and an expression e to be analysed. The node cf represents control flow uniformity associated with the program point at which the expression occurs. $\mathcal{UG}_{\text{expr}}$ returns an updated graph (V', E') and a node $v \in V'$ representing the uniformity of e.

Because `lid` is non-uniform and values read from the shared store are conservatively assumed to be possibly non-uniform, these result in an unmodified graph and

MayBeNonUnif being returned. A literal expression is only non-uniform if it occurs under non-uniform control flow, hence an unmodified graph and cf are returned. Uniformity of a non-`lid` local variable x depends on the uniformity node currently associated with x, given by $M(x)$, and uniformity of control flow, given by cf. Thus the returned graph features a fresh node representing uniformity of the expression, with edges recording dependencies on $M(x)$ and cf. Dependence on the control flow in which the expression is analysed is necessary to taint any variables updated in non-uniform control flow as non-uniform after merging `if`s and `loop`s (explained below). Analysis of a binary expression involves recursion over sub-expressions in the obvious way.

Analysis of statements. We define statement analysis as a function $\mathcal{UG}_{\text{stmt}}$. Like $\mathcal{UG}_{\text{expr}}$ this function takes a uniformity graph (V, E), variable uniformity map M and control flow node $cf \in V$. It takes two further variable uniformity maps, M_{b} and M_{c}, which are used for handling **`break`** and **`continue`** statements, respectively; we describe their purpose below when we discuss the analysis rules that use them. It also takes a *function summary table* F that maps previously-analysed functions to their summaries, and finally the statement or list of statements to be analysed. The function returns an updated uniformity graph (V', E'), an updated map M' representing the uniformity of variables after the (list of) statement(s), and a control flow node $cf' \in V'$ representing control flow uniformity after the (list of) statement(s).

Analysis of an empty list of statements is trivial. Analysis of a non-empty list involves first analysing the head statement s. The tail is then analysed in the context of the analysis state resulting from analysis of s, but *only* if the tail is reachable, i.e. if Next $\in \mathcal{B}(s)$. If the tail is not reachable the result of analysing the head is returned:

$$\begin{aligned}
&\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, [\,]) = (V, E, M, cf)\\
&\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, s : ss) =\\
&\quad \textbf{let } (V_1, E_1, M_1, cf_1) = \mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, s) \textbf{ in}\\
&\quad \textbf{if } \text{Next} \notin \mathcal{B}(s) \textbf{ then } (V_1, E_1, M_1, cf_1) \textbf{ else } \mathcal{UG}_{\text{stmt}}(V_1, E_1, F, M_1, M_{\text{b}}, M_{\text{c}}, cf_1, ss)
\end{aligned}$$

Analysis of assignments is straightforward—an assignment does not affect control flow, so no graph updates are required. Assigning to a local variable x requires the right-hand-side expression to be analysed, yielding an updated graph and uniformity node v for the expression, and the variable uniformity map to be updated to associate x with v:

$$\begin{aligned}
&\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, x\texttt{[}e_1\texttt{]} = e_2) = (V, E, M, cf)\\
&\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, x = e) = \textbf{let } (V_1, E_1, v) = \mathcal{UG}_{\text{expr}}(V, E, M, cf, e) \textbf{ in } (V_1, E_1, M[x \mapsto v], cf)
\end{aligned}$$

Since a barrier can only be used under uniform control flow, analysing a **`barrier`** statement simply involves adding an edge from MustBeUnif to cf. For a **`return`** statement, the returned expression is analysed to obtain an updated graph and expression uniformity node v, and an edge (ValueReturn, v) is added to the graph:

$$\begin{aligned}
&\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, \texttt{barrier}^b) = (V, E \cup \{(\text{MustBeUnif}, cf)\}, M, cf)\\
&\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, \texttt{return } e) =\\
&\quad \textbf{let } (V_1, E_1, v) = \mathcal{UG}_{\text{expr}}(V, E, M, cf, e) \textbf{ in } (V_1, E_1 \cup \{(\text{ValueReturn}, v)\}, M, cf)
\end{aligned}$$

Analysing an `if` statement involves analysing the condition, 'then' branch and 'else' branch in turn (cf. lines 1–3 below). Analysis of the branches results in two new variable uniformity maps, M_{then} and M_{else}. Based on these, a *merged* map M_{after} is constructed (line 5) to track variable uniformity following the `if` statement. For this purpose, M_{after} maps every local variable x_i to a fresh node v_i, and an edge is added from v_i to $M_{\text{then}}(x_i)$ (resp. $M_{\text{else}}(x_i)$) iff the behaviour of the 'then' (resp. 'else') branch includes Next (lines 6–7). Finally, a node representing control flow

after the `if` must be returned. If the behaviour of the entire statement is exactly {Next}—i.e. the branches of the statement do not break, continue or return, control flow remains unchanged and cf is returned (line 8). Otherwise, a fresh control flow node is returned, and edges are added so that this node depends on the control flow associated with the 'then' and 'else' branches (line 9).

$$\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, \texttt{if (}e\texttt{) \{}ss_1\texttt{\} else \{}ss_2\texttt{\}}) =$$

1 **let** $(V_1, E_1, v) = \mathcal{UG}_{\text{expr}}(V, E, M, cf, e)$ **in**
2 **let** $(V_2, E_2, M_{\text{then}}, cf_{\text{then}}) = \mathcal{UG}_{\text{stmt}}(V_1, E_1, F, M, M_{\text{b}}, M_{\text{c}}, v, ss_1)$ **in**
3 **let** $(V_3, E_3, M_{\text{else}}, cf_{\text{else}}) = \mathcal{UG}_{\text{stmt}}(V_2, E_2, F, M, M_{\text{b}}, M_{\text{c}}, v, ss_2)$ **in**
4 **let** $V_4 = V_3 \cup \{v_1, \ldots, v_N\}$ where $v_1, \ldots, v_N$ fresh **in**
5 **let** $M_{\text{after}} = \{x_i \mapsto v_i \mid x_i \in \mathit{LocIdent}\}$ **in**
6 **let** $E_4 = E_3 \cup \{(M_{\text{after}}(x), M_{\text{then}}(x)) \mid x \in \mathit{LocIdent} \wedge \text{Next} \in \mathcal{B}(ss_1)\}$ **in**
7 **let** $E_5 = E_4 \cup \{(M_{\text{after}}(x), M_{\text{else}}(x)) \mid x \in \mathit{LocIdent} \wedge \text{Next} \in \mathcal{B}(ss_2)\}$ **in**
8 **if** $\mathcal{B}(\texttt{if (}e\texttt{) \{}ss_1\texttt{\} else \{}ss_2\texttt{\}}) = \{\text{Next}\}$ **then** $(V_4, E_5, M_{\text{after}}, cf)$
9 **else let** cf' fresh **in**$(V_4, E_5 \cup \{(cf', cf_{\text{then}}), (cf', cf_{\text{else}})\}, M_{\text{after}}, cf')$

Analysis of a function call first involves accumulating uniformity information for each of its parameters (cf. line 1 below), collecting uniformity information about the function from the function summary table (line 2), and creating a fresh control flow node representing the returned value. To indicate the uniformity requirements on the function call, edges are added from MustBeUnif to each of the parameters that must be uniform, and to the call site if it is required to be uniform (line 4). To model the uniformity of the value returned by the function, edges are added from the result of the call to (a) the uniformity of the call itself, (b) all the parameters that could affect the uniformity of the result, and (c) MayBeNonUnif if the result introduces non-uniformity (lines 5 and 6).

$$\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, x = f^{id}(e_1, \ldots, e_n)) =$$

1 **let** $(V_1, E_1, v_1) = \mathcal{UG}_{\text{expr}}(V, E, M, cf, e_1)$ **in** ... **let** $(V_n, E_n, v_n) = \mathcal{UG}_{\text{expr}}(V_{n-1}, E_{n-1}, M, cf, e_n)$ **in**
2 **let** (callSiteUnif, paramsReqToBeUnif, paramAffectsReturnUnif, returnNonUnif) $= F(f)$ **in**
3 **let** v_{res} fresh **in**
4 **let** $E_{\text{call}} = E_n \cup \{(\text{MustBeUnif}, cf) \mid \text{callSiteUnif}\} \cup \{(\text{MustBeUnif}, v_i) \mid i \in \text{paramsReqToBeUnif}\} \cup$
5 $\{(v_{\text{res}}, v_i) \mid i \in \text{paramAffectsReturnUnif}\} \cup \{(v_{\text{res}}, \text{MayBeNonUnif}) \mid \text{returnNonUnif}\} \cup$
6 $\{(v_{\text{res}}, cf)\}$ **in**
7 $(V_n \cup \{v_{\text{res}}\}, E_{\text{call}}, M[x \mapsto v_{\text{res}}], cf)$

Suppose $M(x) = v$ records the uniformity node associated with local variable x right before a `break` statement with associated loop l. Then the uniformity of x directly after l should depend on v. Similarly, the uniformity associated with a variable before a `continue` should influence the uniformity of said variable at the start of the body of the associated `continuing` statement. This is where the maps M_{b} and M_{c} come in: they map local variables to nodes that represent uniformity upon exiting a loop via a `break` or reaching a `continuing` via a `continue`, respectively:

$$\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, \texttt{break}) = (V,\ E \cup \{(M_{\text{b}}(x), M(x)) \mid x \in \mathit{LocIdent}\},\ M,\ cf)$$
$$\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, \texttt{continue}) = (V,\ E \cup \{(M_{\text{c}}(x), M(x)) \mid x \in \mathit{LocIdent}\},\ M,\ cf)$$

Analysis of a loop requires recording the uniformity of variables at the loop head, on loop exit and at the start of the loop's `continuing` statement (cf. lines 2–4 below) in maps M_{head}, M_{exit} and M_{cont}, each of which uses a fresh node per local variable. Edges are added (line 5) to record that uniformity of a variable x at the loop head depends on the uniformity of x on loop entry. If the loop *cannot* continue (because Continue $\notin \mathcal{B}(ss_1)$), control flow uniformity at the loop head cannot be

affected by the loop body. Otherwise, a new control flow node is required to merge control flow information coming into the loop with control flow information from the body (line 6).

The loop body ss_1 is analysed (line 7) with M_{exit} and M_{cont} passed as the M_{b} and M_{c} arguments to $\mathcal{UG}_{\text{stmt}}$ so that appropriate dependencies are added during recursive analysis of **break** and **continue** statements. The continuing statement body ss_2 is then analysed (line 8), with M_{cont} (reflecting variable uniformity at the start of the continuing statement) passed as the M argument to $\mathcal{UG}_{\text{stmt}}$, and $\emptyset$ passed as the M_{c} argument (since **continue** statements appearing under ss_2 must be in nested loops). Edges reflecting iteration of the loop are captured by E_{iterate} (line 9).

The result returned by $\mathcal{UG}_{\text{stmt}}$ depends on the behaviour of ss_1. If the body *cannot* continue (lines 10–11), only the results of analysing the body are relevant. Otherwise (lines 12–13) the results of analysing the **continuing** statement and edges associated with loop iteration are also required. Control flow after the loop is as it was before the loop if the body cannot return (lines 10 and 13), and is tainted by control flow associated with loop execution otherwise (lines 12 and 13). Note that on line 13, Next $\notin \mathcal{B}(ss_1)$ because ss_1 must end with a **continue** due to restriction (3).

$\mathcal{UG}_{\text{stmt}}(V, E, F, M, M_{\text{b}}, M_{\text{c}}, cf, \textbf{loop}^l\{ss_1\ \textbf{continuing}^l\{ss_2\}\}) =$

1 **let** $V_{\text{entry}} = V \cup \{h_1, \ldots, h_N, v_1, \ldots, v_N, w_1, \ldots, w_N\}$ where h_i, v_i, w_i fresh $(1 \leq i \leq N)$ **in**
2 **let** $M_{\text{head}} = \{x_i \mapsto h_i \mid x_i \in \mathit{LocIdent}\}$ **in**
3 **let** $M_{\text{exit}} = \{x_i \mapsto v_i \mid x_i \in \mathit{LocIdent}\}$ **in**
4 **let** $M_{\text{cont}} = \{x_i \mapsto w_i \mid x_i \in \mathit{LocIdent}\}$ **in**
5 **let** $E_{\text{entry}} = E \cup \{(M_{\text{head}}(x_i), M(x_i)) \mid x_i \in \mathit{LocIdent}\}$ **in**
6 **let** cf' fresh, $cf_{\text{head}} =$ **if** Continue $\in \mathcal{B}(ss_1)$ **then** cf' **else** cf **in**
7 **let** $(V_{\text{afterBody}}, E_{\text{afterBody}}, _, cf_{\text{afterBody}}) = \mathcal{UG}_{\text{stmt}}(V_{\text{entry}}, E_{\text{entry}}, F, M_{\text{head}}, M_{\text{exit}}, M_{\text{cont}}, cf_{\text{head}}, ss_1)$ **in**
8 **let** $(V_{\text{afterCont}}, E_{\text{afterCont}}, M_{\text{afterCont}}, cf_{\text{afterCont}}) =$
$\mathcal{UG}_{\text{stmt}}(V_{\text{afterBody}}, E_{\text{afterBody}}, F, M_{\text{cont}}, M_{\text{exit}}, \emptyset, cf_{\text{afterBody}}, ss_2)$ **in**
9 **let** $E_{\text{iterate}} = E_{\text{afterCont}} \cup \{(M_{\text{head}}(x_i), M_{\text{afterCont}}(x_i)) \mid x_i \in \mathit{LocIdent}\} \cup$
$\{(cf_{\text{head}}, cf), (cf_{\text{head}}, cf_{\text{afterCont}})\}$ **in**
10 **if** $\mathcal{B}(ss_1) = \{\text{Break}\}$ **then** $(V_{\text{afterBody}}, E_{\text{afterBody}}, M_{\text{exit}}, cf)$
11 **else if** $\mathcal{B}(ss_1) \subseteq \{\text{Break}, \text{Return}\}$ **then** $(V_{\text{afterBody}}, E_{\text{afterBody}}, M_{\text{exit}}, cf_{\text{afterBody}})$
12 **else if** $\{\text{Continue}, \text{Return}\} \subseteq \mathcal{B}(ss_1)$ **then** $(V_{\text{afterCont}}, E_{\text{iterate}}, M_{\text{exit}}, cf_{\text{head}})$
13 **else** $(V_{\text{afterCont}}, E_{\text{iterate}}, M_{\text{exit}}, cf)$ // $\mathcal{B}(ss_1) = \{\text{Break}, \text{Continue}\} \vee \mathcal{B}(ss_1) = \{\text{Continue}\}$

Analysis of a shader. Uniformity analysis processes a shader function-by-function, where analysis of each function either yields a uniformity error or a function summary that can be used in the analysis of subsequent functions. Analysis of a function is performed by $\mathcal{UG}_{\text{func}}$ as follows:

$\mathcal{UG}_{\text{func}}(F, \textbf{fn}\ f(x_1, \ldots, x_n)\ \{\ ss\ \}) =$

1 **let** $v_1, \ldots, v_N$ fresh **in**
2 **let** $V_{\text{init}} = \{v_1, \ldots, v_N, \text{CfStart}, \text{MayBeNonUnif}, \text{MustBeUnif}, \text{ValueReturn}\}$ **in**
3 **let** $E_{\text{init}} = \{(v_i, \text{CfStart}) \mid y_i \in \mathit{LocIdent} \setminus \{x_1, \ldots, x_n\}\}$ **in**
4 **let** $M_{\text{init}} = \{y_i \mapsto v_i \mid y_i \in \mathit{LocIdent}\}$ **in**
5 **let** $(V, E, _, _) = \mathcal{UG}_{\text{stmt}}(V_{\text{init}}, E_{\text{init}}, F, M_{\text{init}}, \emptyset, \emptyset, \text{CfStart}, ss)$ **in**
6 **let** callSiteUnif $= (\text{MustBeUnif}, \text{CfStart}) \in E^*$ **in**
7 **let** paramsReqToBeUnif $= \{i \mid (\text{MustBeUnif}, M_{\text{init}}(x_i)) \in E^* \wedge 1 \leq i \leq n\}$ **in**
8 **let** paramAffectsReturnUnif $= \{i \mid (\text{ValueReturn}, M_{\text{init}}(x_i)) \in E^* \wedge 1 \leq i \leq n\}$ **in**
9 **let** returnNonUnif $= (\text{ValueReturn}, \text{MayBeNonUnif}) \in E^*$ **in**
10 **let** $F' = F[f \mapsto (\text{callSiteUnif}, \text{paramsReqToBeUnif}, \text{paramAffectsReturnUnif}, \text{returnNonUnif})]$ **in**
11 **if** $(\text{MustBeUnif}, \text{MayBeNonUnif}) \in E^*$ **then** $\mathcal{E}$ **else** F'

This starts with a uniformity graph comprising a node for every local variable plus each special node (line 2) and an edge from every non-parameter name to CfStart (line 3, where each y_i corresponds to the variable with associated uniformity node v_i). Analysis of the body is performed (line 5), and a corresponding function summary is constructed (lines 6–10). The result of analysing a function is either an updated function summary table, or an error if a uniformity violation is found (line 11).

For a TinyWGSL shader P, $\mathcal{UG}_{\text{shader}}(P) = \text{OK}$ if analysis of the functions of P, in order, is error-free, and $\mathcal{UG}_{\text{shader}}(P) = \mathcal{E}$ if analysis of one of these functions yields an error.

With a rigorous definition of barrier divergence (Definition 3.2) and a precise definition of uniformity analysis we can now state a soundness theorem:

Theorem 4.1 (Soundness of Uniformity Analysis). Let P be a TinyWGSL shader such that $\mathcal{UG}_{\text{shader}}(P) = \text{OK}$. Then P does not exhibit barrier divergence.

Proof sketch. A complete proof of the soundness theorem would require a detailed case analysis considering the interaction between every uniformity analysis rule and each relevant operational semantics rule. The intricate nature of such a proof would make it an ideal application for mechanisation using a proof assistant, which would be an interesting future project. Here we present a high-level sketch of the soundness argument behind Theorem 4.1.

We define some additional notation necessary to be able to access the uniformity node and variable map used to analyse each statement in P. For this purpose, assume that every statement and expression in a TinyWGSL shader has a distinct label $l \in \mathit{Label}$ where Label is a countably infinite set of labels. As described in Section 3.1, **`barrier`**^{b} and **`continue`**^{c} statements are already equipped with the unique IDs b and c respectively. Thus, we adopt this syntax and write a statement s with label l as s^l and similarly, a labelled expression e with label l as e^l. Equality between statements and expressions is defined in terms of their labels.

Take the analysis to return two maps: (1) the map $\mathcal{M}_s : \mathit{Label} \to (\mathit{UnifNode} \times (\mathit{LocIdent} \to \mathit{UnifNode}))$ that maps each statement label l to the pair (cf, M), where cf and M are the uniformity node and variable map taken as input to analyse the statement with the label l; and (2) the map $\mathcal{M}_e : \mathit{Label} \to \mathit{UnifNode}$ that maps the label associated with each expression to the uniformity node returned from the analysis of that expression. It is trivial to augment the definition of TinyWGSL in Section 3 to contain labels and the analysis of Section 4.2 to produce such a map.

Given a global state $G = \langle \sigma, (T_0, \ldots, T_{N-1}) \rangle$, define $G.T_i = T_i = (\tau, \kappa, ss)$ for $0 \le i < N$. Similarly, given the thread local state $T = (\tau, \kappa, ss)$, define $T.ss = ss$.

Consider analysing an arbitrary function f in P. Let (V_f, E_f) be the final graph produced by the analysis of f with $\mathcal{UG}_{\text{func}}$. In this graph, a node cf is considered to be uniform if and only if there is no path from cf to MayBeNonUnif:

Definition 4.2 (IsUniform). $\text{IsUniform}(cf) \iff (cf, \text{MayBeNonUnif}) \notin E_f^*$.

Similarly, a statement is considered uniform if and only if it is analysed with a uniform control flow node:

Definition 4.3 (StmtIsUniform). $\text{StmtIsUniform}(s^l) \iff (\mathcal{M}_s(l) = (cf, M) \implies \text{IsUniform}(cf))$

Being conservative, the analysis is designed to under-approximate the set of variables that are uniform at each program point. That is, if the analysis considers a variable x to be uniform at a statement, then whenever two threads visit the statement with the same control stack, the thread local stores of the two threads agree on the value of x. The predicate $\text{StoresAgreeAt}(l, \tau_i, \tau_j)$ encapsulates the fact that the local stores τ_i and τ_j agree on the value of uniform variables at the statement with label l.

Definition 4.4 (StoresAgreeAt). $\text{StoresAgreeAt}(l, \tau_i, \tau_j) \iff (\mathcal{M}_s(l) = (cf, M) \implies (\forall x \in (\mathit{LocIdent} \setminus \{\texttt{lid}\}).\ (\text{IsUniform}(M(x)) \implies \tau_i(x) = \tau_j(x))))$

Now, to prove the soundness theorem, assume for a contradiction that there exists a shader P such that $\mathcal{UG}_{\text{shader}}(P) = \text{OK}$ but P exhibits barrier divergence. Then from the definition of barrier divergence (Definition 3.2), there are global states $G_0, \ldots, G_M$ such that G_0 is an initial state of P, $G_M = \mathcal{E}$ and $G_0 \xrightarrow{t} \ldots \xrightarrow{t} G_M$. The only rule that can transition to $\mathcal{E}$ is Barrier-Error and thus to satisfy the premise of Barrier-Error, the state before G_M, i.e. G_{M-1}, must have two threads i and j that are at different barriers or the same barrier with different control stacks κ_i and κ_j.

Taking *GlobalState* to be the set of all global states, to find a contradiction we define a relation $R_{i,j} \subseteq \mathit{GlobalState} \times \mathit{GlobalState}$ such that $R_{i,j}(G_m, G_n)$ holds if and only if all of the following hold, where $G_m.T_i = (\tau_i, \kappa_i, s_i^{l_i} : ss_i),\ G_n.T_j = (\tau_j, \kappa_j, s_j^{l_j} : ss_j)$:

- $s_i^{l_i} : ss_i = s_j^{l_j} : ss_j$ (the threads are at the same statement, with the same continuation)
- $\kappa_i = \kappa_j$ (the control stacks agree)
- $\text{StmtIsUniform}(s_i^{l_i})$ (the statement $s_i^{l_i}$ that the threads are at is uniform)
- $\text{StoresAgreeAt}(l_i, \tau_i, \tau_j)$ (threads i and j agree on the values of uniform variables at $s_i^{l_i}$)

Since $R_{i,j}$ requires that i and j are at the same statement with the same control stacks, if we can show $R_{i,j}(G_{M-1}, G_{M-1})$ holds then it is impossible for G_{M-1} to transition to $\mathcal{E}$, contradicting our assumption that barrier divergence occurs.

Intuitively, we will argue that $R_{i,j}$ holds at pairs of global states where i and j are at matching uniform statements, and that from matching uniform statements, i and j transition to the same next uniform statement. It is straightforward to show by contradiction that when the analysis passes for a shader, all of the barriers in that shader are uniform statements; i.e. $\text{StmtIsUniform}(\texttt{barrier}^b)$ holds for every $\texttt{barrier}^b$ in P. Thus, statements deemed non-uniform cannot lead to barrier divergence.

Proving that $R_{i,j}(G_{M-1}, G_{M-1})$ holds proceeds by induction, and involves showing that three conditions all hold:

(1) $R_{i,j}(G_0, G_0)$

(2) (a) $\forall\ G_m, G_n\ .\ R_{i,j}(G_m, G_n)\ \wedge\ G_m \xrightarrow{t} G'_m\ \wedge\ G_m.T_i = G'_m.T_i\ \implies\ R_{i,j}(G'_m, G_n)$

(b) $\forall\ G_m, G_n\ .\ R_{i,j}(G_m, G_n)\ \wedge\ G_n \xrightarrow{t} G'_n\ \wedge\ G_n.T_j = G'_n.T_j\ \implies\ R_{i,j}(G_m, G'_n)$

(3) $\forall\ G_m, G_n\ .$

$$
\begin{aligned}
&R_{i,j}(G_m, G_n) \\
&\wedge\ G_m \xrightarrow{t} \ldots \xrightarrow{t} G_{m+a}\ \wedge \forall m < x < m + a.\ \neg\text{StmtIsUniform}(\text{head}(G_x.T_i.ss)) \\
&\wedge\ G_n \xrightarrow{t} \ldots \xrightarrow{t} G_{n+b}\ \wedge \forall n < x < n + b.\ \neg\text{StmtIsUniform}(\text{head}(G_x.T_j.ss)) \\
&\wedge\ \text{StmtIsUniform}(\text{head}(G_{m+a}.T_i.ss)) \wedge \text{StmtIsUniform}(\text{head}(G_{n+b}.T_j.ss)) \\
\implies\ &R_{i,j}(G_{m+a}, G_{n+b})
\end{aligned}
$$

Condition (1) is the base case and follows directly from the definition of an initial state (Definition 3.1).

Condition (2) says that $R_{i,j}$ is preserved when a thread other than i makes a step from G_m, or a thread other than j makes a step from G_n. This is immediate from the definition of $R_{i,j}$, which only depends on the T_i and T_j components of a given pair of global states.

Condition (3) requires that if threads i and j are at *uniform* statements in $R_{i,j}$-related states G_m and G_n, and each executes a number of exclusively *non-uniform* statements to reach a subsequent *uniform* statement, then the associated global states at the resulting uniform statement must be $R_{i,j}$-related. That is, $R_{i,j}$ is restored whenever the control flow of threads reconverges after a period of (potential) divergence.

It is easy to show by induction on expressions that $\mathcal{UG}_{\text{expr}}$ is sound, i.e. any expression that is deemed uniform and for which StoresAgreeAt holds will be evaluated identically by two threads. The proof of condition (3) then proceeds by structural induction on statements.

If both i and j are at the same **barrier** and they both take a step, the only way they could have done so is via the Barrier-Sync rule, which implies that $G_m = G_n$ and $G_{m+1} = G_{n+1}$. The statement at G_{m+1} will be uniform and $R_{i,j}(G_{m+1}, G_{n+1})$ holds. For assignment, **break**, **continue**, and **return** the proof is simple: both threads reach the next uniform statement in one step. They reach the same next statement with the same control stacks because there is only a single $\xrightarrow{s}$ rule for any statement that is not an **if**. From $\mathcal{UG}_{\text{stmt}}$ rule for local assignment and the fact that $\mathcal{UG}_{\text{expr}}$ is sound, it follows that local assignment preserves local store agreement for uniform variables.

For an **if** statement with a uniform condition, the soundness of $\mathcal{UG}_{\text{expr}}$ implies that threads i and j will step to the same branch, the definition of $\mathcal{UG}_{\text{stmt}}$ guarantees that first statement in the body of the branch will be uniform, and $R_{i,j}$ holds.

For an **if** statement with a non-uniform condition, things are more complicated. The next uniform statement reached depends on the definition of $\mathcal{UG}_{\text{stmt}}$ and, in particular, on the behaviour determined by $\mathcal{B}$ (Figure 8). If the **if** statement has the behaviour {Next}, then the analysis deems the statement ss immediately after the **if** to be uniform. It can be seen that from the definition of $\xrightarrow{s}$, both i and j must reach ss regardless of the branches they took. For an **if** statement with behaviour that is a subset of {Break, Continue}, threads reconverge after the **loop** containing the if statement. If the behaviour of the **if** statements contains Return, then threads reconverge after the call to the enclosing function. In each case, the analysis treats all statements up to the reconvergence point as non-uniform. It can then be shown that both threads do in-fact reach this reconvergence point and that control stack agreement and local store agreement on uniform variables are preserved.

The argument for **loop** statements is similar to that of **if** statements, except that the behaviour of all loops is a subset of {Next, Return} (Figure 8), leading threads to either reconverge after the loop or after the call to the enclosing function.

For the function call case, it is necessary to show that function summaries soundly over-approximate the requirements of a function so that any call to a function satisfying its summary cannot lead to barrier divergence. Because WGSL forbids recursion, this can be done by induction, first proving that the summary of any function with no callees, and thus no function calls, is over-approximate. Then for the inductive step, it can be shown that if all the callees of a given function are over-approximate, then the summary of that function is over-approximate.

Since together (1)-(3) show $R_{i,j}(G_{M-1}, G_{M-1})$ holds, we have found a contradiction.

5 Improving Uniformity Analysis in the WGSL Specification

Our study of WGSL uniformity analysis led us to discover soundness and precision issues, as per Problem ❸ in the introduction. We briefly discuss the main problems we found and the fixes we proposed, all of which have been adopted and will feature in an upcoming specification release.

Unsound handling of 'break-if'. A WGSL **continuing** statement can optionally end with a 'break-if' statement that is semantically equivalent to **if** (*e*) {**break**} **else** {}. Control flow uniformity following a 'break-if' statement depends on the uniformity of the condition; e.g. the shader of Figure 10a exhibits barrier divergence because thread 0 will reach the barrier at line 8, while all other threads continue to the barrier at line 3. We found that WGSL uniformity analysis would *unsoundly* accept shaders with non-uniform 'break-if' expressions, such as Figure 10a, as uniform because the rule for handling 'break-if' statements neglected to add an edge from the node representing control flow uniformity after such a statement to the node representing uniformity of the statement's condition. Our report and suggested fix [32] led to a specification change [43].

```
1 fn main(lid: u32) {
2   loop {
3     workgroupBarrier();
4     continuing {
5       break if (lid == 0);
6     }
7   }
8   workgroupBarrier();
9 }
```

(a) Unsoundly accepted

```
fn main(lid: u32) {
  loop {
    return;
    continuing {
      break if (...);
    }
  }
  if (lid == 0) { workgroupBarrier(); }
}
```

(b) Rejected due to overly-conservative analysis

```
1  fn main(lid: u32) {
2    var u: u32 = 0;
3    loop {
4      workgroupBarrier();
5      loop {
6        if (lid < u) { return; }
7        u++;
8      }
9    }
10 }
```

(c) Rejected due to overly-conservative analysis

```
fn main(lid: u32) {
  loop {
    return;
    continuing {
      if (lid == 0) {
        workgroupBarrier();
      }
    }
  }
}
```

(d) Rejected due to overly-conservative analysis

Fig. 10. WGSL shaders illustrating soundness and precision issues found via our work

Overly conservative statement behaviour analysis. We found a deficiency in statement behaviour analysis (Section 4.2) whereby code following the loop in Figure 10b—clearly unreachable due to the unconditional **`return`** in the loop body—would be analysed, leading to a spurious error due to the seemingly non-uniform (but actually unreachable) barrier at line 8. The problem was that the behaviour of a loop was always taken to be the union of the behaviour of the loop body and the **`continuing`** body, which makes it appear possible to break from the loop. In reporting this [31] we proposed that behaviour of a **`loop`** whose body has behaviour {Return} should also have behaviour {Return} (see the **`loop`** rule of $\mathcal{B}$ on page 13), which led to a specification change [42].

Overly conservative addition of loop back-edges. The shader of Figure 10c cannot exhibit barrier divergence because all threads synchronise once at the barrier on line 4, and then no thread reaches a further barrier: the only way to exit the inner loop is by returning at line 6, so no further iterations of the outer loop are possible. WGSL uniformity analysis rejected this shader by making control flow at the loop head depend unconditionally on control flow at the end of the body. The non-uniform **`return`** on line 6 led the analysis to deem control flow following the inner loop to be non-uniform, thus making control flow at the outer loop head non-uniform. In reporting this [33] we proposed adding a dependency *only* when it is possible for the loop to iterate (see lines 12 and 13 in the loop rule of $\mathcal{UG}_{\text{stmt}}$ on page 16), which led to a specification change [45].

Overly conservative analysis of continuing statements. Figure 10d cannot exhibit barrier divergence because the barrier at line 6 is unreachable. WGSL uniformity analysis needlessly rejected this example by analysing the **`continuing`** statement of a loop, and connecting the results of this analysis to the uniformity graph, even when the continuing construct is unreachable. In reporting this [33] we proposed effectively skipping analysis of unreachable **`continuing`** statements (captured in lines 10 and 11 in the loop rule of $\mathcal{UG}_{\text{stmt}}$ on page 16), which led to a specification change [45].

Additional fixes. Our work also led to us finding and reporting: (1) a lack of clarity surrounding the desugaring assumptions made by uniformity analysis [29], (2) gaps in analysis rules for expressions and statements [27], (3) failure to account for empty statements [26], (4) failure to account for empty

'else' branches in `if` statements [30], and (5) failure to account for parentheses expressions [28]. All of these have been fixed in response to our reports [44].

6 Better Testing of Uniformity Analysis Implementations

To enable automated testing of uniformity analysis implementations against the formal rules of Section 4.2, in response to problem ❹ in the introduction, we have mechanised the analysis using the Alloy modelling language [20], which is ideal for representing graph-structured domains, such as programs and the uniformity graphs induced by uniformity analysis.

We have designed an Alloy model comprising a set of definitions and constraints that together describe a TinyWGSL program and its corresponding uniformity graph. The constraints ensure that the program is well formed, and that the uniformity graph satisfies the analysis rules presented in Section 4.2. The Alloy Analyzer [20] translates these constraints into a Boolean satisfiability problem, solved by an off-the-shelf SAT solver, where a satisfying instance corresponds to a TinyWGSL program and an associated uniformity graph that jointly meet all specified constraints.

The core model comprises 1,115 lines of Alloy code. We provide a collection of predicates that constrain the generation process to produce programs with specific characteristics, for example, programs that are or are not uniform, or that exhibit particular control-flow structures. To test a uniformity analysis implementation, we have developed tooling for converting an Alloy-generated example into WGSL program text and an associated uniformity expectation that serves as a test oracle. For simplicity and to help test-case generation scale, we omit shared variables from our Alloy model. This omission does not exclude interesting behaviours, as our testing is entirely static and uniformity analysis conservatively handles shared variable reads in the same manner as `lid`.

As per Section 2.2, Firefox and Safari do not currently feature mature implementations of uniformity analysis, thus we focus our testing efforts on the Chromium implementation, Tint.

Using our Alloy model to test Tint. We co-developed the uniformity analysis rules of Section 4.2 with our Alloy model, which we cross-checked against Tint. This process revealed a parser bug [25]—subsequently fixed [51]—and discovered both Figures 10a and 10d (Section 5), because Tint diverged from the specification and therefore disagreed with our model. To maximise confidence in our model, we performed a controlled experiment in which we used the Alloy Analyzer to generate ASTs that are constrained to be uniform or non-uniform under the modelled analysis and checked that Tint agreed with this result. Tint has yet to adopt the change resulting from Figure 10d, and so for this experiment we used the model in a (toggleable) more conservative mode to match Tint.

To diversify the generation of examples, we included a set of *universal constraints* that all generated examples must satisfy, and a set of *local constraints* that will be satisfied by some portion of the generated examples. All examples were constrained to: (1) contain some barrier, (2) have no unused local variables, and (3) have no un-called functions. The local constraints are largely arbitrary but provoke the Alloy Analyzer into generating more interesting ASTs. Importantly, one of the local constraints is `true`, which is equivalent to having no local constraint on the AST. The local constraints were distributed amongst threads, repeatedly solved, converted to WGSL shaders, and checked against Tint. If a constraint became unsatisfiable, it was removed.

Over a 24 hour period on a laptop with an i9-185H and 32GB RAM, the Alloy 6.2.0 API with the Sat4J SAT solver generated 7,484 unique shaders (2,383 uniform; 5,101 non-uniform) across 8 threads. Tint commit 10cbb7e was used for comparison and no discrepancies were found.

7 Related Work

GPU program semantics. The most directly related work studies the semantics of GPU programs at the level of control flow graphs corresponding to compiler intermediate representation [14], with

the aim of providing formal underpinnings for the GPUVerify verification tool [10, 11]. A language of unstructured, reducible control flow graphs is presented, with a semantics for detecting data races and barrier divergence in control flow graphs lowered from CUDA and OpenCL (the languages supported by GPUVerify). While this semantics helped inform the semantics of TinyWGSL, there are several key differences: The requirements on barrier divergence in SPIR-V (see Section 2.3.1) place constraints on a WGSL definition of barrier divergence that do not apply to OpenCL and CUDA (an OpenCL or CUDA version of Figure 1c would *not* be considered barrier divergent), necessitating a significant departure in the TinyWGSL semantics; the CFG-based semantics does not support function calls and therefore does not consider the problem of how to detect barrier divergence due to non-uniform execution of function call chains; and our TinyWGSL semantics caters for WGSL-specific features (e.g. the non-standard `loop`…`continuing` construct).

Various works are concerned with the semantics of 'warps' (subgroups of a workgroup) [12, 19], GPU memory consistency [1, 16, 38, 56] and guarantees of independent forward progress between GPU threads [52, 53], but none of these works address the semantics of barrier divergence.

Previous work has formalised static analyses that identify uncoalesced memory accesses [2] and block-size independence [3] in GPU shaders. While both analyses relate to the use of thread IDs, with the latter being a taint analysis, neither deals with barrier divergence nor is graph-based.

Recent work on SPIR-V control flow used Alloy-generated tests to find bugs in SPIR-V tooling [24], akin to our use of Alloy to test Tint. While SPIR-V control flow informs notions of SPIR-V barrier divergence (Section 2.3), the work does not relate directly to barrier divergence semantics.

Uniformity analysis. We believe we are the first to formally study WGSL uniformity analysis, and although in general terms uniformity analysis can be viewed as a form of static taint analysis, we are not aware of any existing analyses with similar aims: while GPUVerify provides support for proving absence of barrier divergence for CUDA and OpenCL, it uses invariant generation and user-supplied annotations to reason about arbitrary use of barriers, whereas uniformity analysis is more akin to a type system, providing one-size-fits-all rules that can be efficiently checked. A different kind of uniformity analysis can be used as part of an optimising compiler targeting GPU or other vector architectures [47, 50], and the LLVM compiler framework provides analyses related to uniformity and convergence [37]. Here, the aim is to restructure an intermediate representation of a program to *maximise* uniform control flow for purposes of execution efficiency, and not to judge whether or not programs should be considered correct with respect to barrier divergence.

8 Conclusions and Future Work

By rigorously defining barrier divergence and uniformity analysis for TinyWGSL in a precise manner, we have laid the foundations for proper reasoning about the soundness of this important analysis. Our careful study of the WGSL specification, interaction with the specification editors and Chromium developers, and mechanisation of our analysis formulation using Alloy has led to the discovery of soundness and precision issues which have been rectified in the official specification.

An ambitious avenue for future work would be to formalise the semantics for one or more of the downstream shading languages to which WGSL is compiled (e.g. SPIR-V, which is defined more rigorously than MSL or HLSL), and propose compilation from WGSL to these languages that are guaranteed to preserve barrier-divergence freedom. Another interesting direction would be to study the *maximal reconvergence* extension in SPIR-V [6, 8], which—if broadly adopted—could avoid the arguably non-intuitive categorisation of shaders such as that of Figure 1c as barrier divergent.

Our testing work so far has focused on the Tint shader compiler in Chromium, as it has a mature implementation of uniformity analysis. It would be valuable to apply our automated testing approach to uniformity analysis implementations in Firefox and Safari once they mature.

Acknowledgements

We are grateful to David Neto, Alan Baker and James Price for their support discussing and resolving the various WGSL issues presented in the paper; to the anonymous PLDI 2026 reviewers on the paper and artifact committees for their valuable feedback; and to Google for support via gift funding.

Data Availability

All tooling and experimental data created/used for this research are openly available for reproduction, reuse, and scrutiny at [34].

References

[1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 577–591. doi:10.1145/2694344.2694391

[2] Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. 2017. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 507–525. doi:10.1007/978-3-319-63387-9_25

[3] Rajeev Alur, Joseph Devietti, and Nimit Singhania. 2018. Block-Size Independence for GPU Programs. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings (Lecture Notes in Computer Science)*, Andreas Podelski (Ed.). Springer, 107–126. doi:10.1007/978-3-319-99725-4_9

[4] Apple. 2025. Metal. https://developer.apple.com/documentation/metal Accessed: 2025-11-07.

[5] Apple. 2025. Metal Shading Langauge Specification. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf Accessed: 2025-11-13.

[6] Alan Baker. 2024. Khronos Releases Maximal Reconvergence and Quad Control Extensions for Vulkan and SPIR-V. https://www.khronos.org/blog/khronos-releases-maximal-reconvergence-and-quad-control-extensions-for-vulkan-and-spir-v Accessed: 2025-11-09.

[7] Alan Baker and David Neto. 2024-2025. Personal communication. These are two of the editors of the WGSL specification. Communication was via a series of teleconferences, and via the WGSL specification issue tracker.

[8] Alan Baker, David Neto, Jeff Bolz, Graeme Leese, Nicolai Hahnle, Ruihao Zhang, and Tobias Hector. 2024. SPV_KHR_maximal_reconvergence. https://github.khronos.org/SPIRV-Registry/extensions/KHR/SPV_KHR_maximal_reconvergence.html Accessed: 2025-11-09.

[9] Alan Baker, David Neto, and James Price. 2024-2025. Personal communication. These are three of the developers of Tint, and the uniformity analysis component of Tint specifically. Communication was via a series of teleconferences, and via the issue tracker of Dawn, the project of which Tint is a component.

[10] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. 2015. The Design and Implementation of a Verification Technique for GPU Kernels. *ACM Trans. Program. Lang. Syst.* 37, 3 (2015), 10:1–10:49. doi:10.1145/2743017

[11] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 113–132. doi:10.1145/2384616.2384625

[12] Zheyuan Chen, Naomi Rehman, Guido Martínez, and Tyler Sorensen. 2026. SIMT-Step Execution: A Flexible Operational Semantics For GPU Subgroup Behaviour. *Proc. ACM Program. Lang.* 10, PLDI (2026). doi:10.1145/3808297

[13] Chromium Docs. 2023. WebGPU Technical Report. https://chromium.googlesource.com/chromium/src/+/main/docs/security/research/graphics/webgpu_technical_report.md Accessed: 2025-10-29.

[14] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 2013. Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 270–289. doi:10.1007/978-3-642-37036-6_16

[15] World Wide Web Consortium. 2025. WebGPU Shading Language. https://www.w3.org/TR/WGSL/ Accessed: 2025-11-05.

[16] Benedict R. Gaster, Derek Hower, and Lee W. Howes. 2015. HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models. *ACM Trans. Archit. Code Optim.* 12, 1 (2015), 7:1–7:26. doi:10.1145/2701618
[17] Google. 2025. Dawn. https://dawn.googlesource.com/dawn Accessed: 2025-11-10.
[18] Google. 2025. Tint. https://dawn.googlesource.com/dawn/+/refs/heads/main/src/tint/ Accessed: 2025-11-07th.
[19] Axel Habermaier and Alexander Knapp. 2012. On the Correctness of the SIMT Execution Model of GPUs. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 316–335. doi:10.1007/978-3-642-28869-2_16
[20] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76. doi:10.1145/3338843
[21] Khronos Group. 2025. Khronos Vulkan Registry. https://registry.khronos.org/vulkan/ Accessed: 2025-11-07.
[22] Khronos Group. 2025. SPIR-V Specification. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html Accessed: 2025-11-07.
[23] Khronos Group. 2025. WebGL Specification. https://registry.khronos.org/webgl/specs/latest/1.0/ Accessed: 2025-10-29.
[24] 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 (2023), 1740–1769. doi:10.1145/3571253
[25] James Lee-Jones. 2025. Chromium Issue Tracker issue: failure to compile a continuing containing an empty statement before a break if. https://issues.chromium.org/issues/453961764 Accessed: 2025-11-13.
[26] James Lee-Jones. 2025. GitHub issue: Add a rule for empty statements to uniformity rules for statements. https://github.com/gpuweb/gpuweb/issues/5123 Accessed: 2025-11-13.
[27] James Lee-Jones. 2025. GitHub issue: Add function call statement and expression rules to uniformity analysis. https://github.com/gpuweb/gpuweb/issues/5128 Accessed: 2025-11-13.
[28] James Lee-Jones. 2025. GitHub issue: Add parenthesis rule to uniofrmity rules for expressions. https://github.com/gpuweb/gpuweb/issues/5121 Accessed: 2025-11-13.
[29] James Lee-Jones. 2025. GitHub issue: Desugaring assumptions in uniformity analysis. https://github.com/gpuweb/gpuweb/issues/5129 Accessed: 2025-11-13.
[30] James Lee-Jones. 2025. GitHub issue: Explicitly state how to handle an if statement with an empty else in uniformity analysis. https://github.com/gpuweb/gpuweb/issues/5122 Accessed: 2025-11-13.
[31] James Lee-Jones. 2025. GitHub issue: Making uniformity analysis less conservative via statement behaviour analysis. https://github.com/gpuweb/gpuweb/issues/5100 Accessed: 2025-11-13.
[32] James Lee-Jones. 2025. GitHub issue: Uniformity Analysis - Account for break if expression uniformity. https://github.com/gpuweb/gpuweb/issues/5277 Accessed: 2025-11-13.
[33] James Lee-Jones. 2025. GitHub issue: Uniformity Analysis - Conservative Handling of Loop Return and a Possible Solution. https://github.com/gpuweb/gpuweb/issues/5364 Accessed: 2025-11-13.
[34] James Lee-Jones, John Wickerson, and Alastair Donaldson. 2026. *Artifact for "Uniformity Analysis in the WebGPU Shading Language", PLDI 2026.* doi:10.5281/zenodo.19556935
[35] Mira Leung. 2022. WebGL-powered maps features now generally available. https://mapsplatform.google.com/resources/blog/webgl-powered-maps-features-now-generally-available/ Accessed: 2025-10-29.
[36] Reese Levine, Ashley Lee, Neha Abbas, Kyle Little, and Tyler Sorensen. 2025. SafeRace: Assessing and Addressing WebGPU Memory Safety in the Presence of Data Races. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 697–725. doi:10.1145/3763075
[37] LLVM Compiler Infrastructure. 2025. Convergence And Uniformity. https://llvm.org/docs/ConvergenceAndUniformity.html Accessed: 2025-11-12.
[38] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. (2019), 257–270. doi:10.1145/3297858.3304043
[39] Microsoft. 2025. Reference for HLSL - AllMemoryBarrierWithGroupSync function. https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/allmemorybarrierwithgroupsync Accessed: 2025-11-13.
[40] Microsoft Corporation. 2021. Direct3D 12 programming guide. https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide Accessed: 2025-11-07.
[41] Microsoft Corporation. 2025. HLSL Specifications. https://github.com/microsoft/hlsl-specs Accessed: 2025-11-09.
[42] David Neto. 2025. GitHub pull request: uniformity: fix analysis for loop bodies that only return. https://github.com/gpuweb/gpuweb/pull/5392 Accessed: 2025-11-13.
[43] David Neto. 2025. GitHub pull request: uniformity: fix analysis of break-if condition. https://github.com/gpuweb/gpuweb/pull/5369 Accessed: 2025-11-13.
[44] David Neto. 2025. GitHub pull request: 'uniformity: fix editorial issues'. https://github.com/gpuweb/gpuweb/pull/5394 Accessed: 2025-11-13.

[45] David Neto. 2025. GitHub pull request: uniformity: relax rules when loop bodies always break or return. https://github.com/gpuweb/gpuweb/pull/5419 Accessed: 2025-11-13.
[46] Emma Ning. 2024. ONNX Runtime Web unleashes generative AI in the browser using WebGPU. https://opensource.microsoft.com/blog/2024/02/29/onnx-runtime-web-unleashes-generative-ai-in-the-browser-using-webgpu Accessed: 2025-10-29.
[47] Julian Rosemann, Simon Moll, and Sebastian Hack. 2021. An abstract interpretation for SPMD divergence on reducible control flow graphs. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–31. doi:10.1145/3434312
[48] Rust Graphics Mages. 2025. Naga. https://github.com/gfx-rs/wgpu/tree/trunk/naga Accessed: 2025-11-07.
[49] Rust Graphics Mages. 2025. wgpu. https://github.com/gfx-rs/wgpu Accessed: 2025-11-10.
[50] Sameer Sahasrabuddhe and Nicolai Hähnle. 2022. Uniformity Analysis for Irreducible CFGs. In *LLVM Developers' Meeting*. https://llvm.org/docs/ConvergenceAndUniformity.html Accessed: 2025-11-12.
[51] Dan Sinclair. 2025. Google Git commit: Fix parsing of a semicolon before a break-if. https://dawn.googlesource.com/dawn/+/10cbb7e35aaefc47143c7104043545d5bd0fb9e3 Accessed: 2025-11-13.
[52] Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson. 2018. GPU Schedulers: How Fair Is Fair Enough?. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China (LIPIcs, Vol. 118)*, Sven Schewe and Lijun Zhang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:17. doi:10.4230/LIPICS.CONCUR.2018.23
[53] 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 (2021), 1–30. doi:10.1145/3485508
[54] The WebKit Open Source Project. 2025. WebKit. https://github.com/WebKit Accessed: 2025-11-10.
[55] The WebKit Open Source Project. 2025. WGSL shader translation. https://github.com/WebKit/WebKit/tree/main/Source/WebGPU/WGSL Accessed: 2025-11-07.
[56] Haining Tong, Natalia Gavrilenko, Hernán Ponce de León, and Keijo Heljanko. 2024. 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 2024, Hilton La Jolla Torrey Pines, La Jolla, CA, USA, 27 April 2024 - 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 329–344. doi:10.1145/3622781.3674174
[57] Corentin Wallez, Brandon Jones, and François Beaufort. 2023. WebGPU: Unlocking modern GPU access in the browser. https://developer.chrome.com/blog/webgpu-io2023/ Accessed: 2025-10-29.
[58] World Wide Web Consortium. 2025. WebGPU. https://www.w3.org/TR/webgpu/ Accessed: 2025-10-29.

Received 2025-11-13; accepted 2026-04-03