N4238 | An Abstract Model of Vector Parallelism

Pablo Halpern (pablo.g.halpern@intel.com), Intel Corp.

2014-10-13

1 Abstract

At the June 2014 meeting in Rapperswil, SG1 (Parallelism and Concurrency) adopted the minimal recommendation in N4060 to rename the term `vector_execution_policy` to `parallel_vector_execution_policy` in the working draft of the parallelism TS (N4071). The reason for that change was to reserve `vector_execution_policy` for a vector-only policy with characteristics yet to be determined, but which differ from a combined parallel+vector execution policy. This paper is a first step in articulating a vector-only model, not only so that we can define `vector_execution_policy`, but so that vectorization can be integrated into C++ in general.

This paper is not a proposal; it is a white paper with the purpose of creating a conceptual framework and vocabulary for informed discussion of vectorization. Three software models of vector execution are presented, all sharing a common vocabulary and a core set of common concepts. The common concepts are explored first, followed by a discussion of each model in turn. For each model, I explore its distinguishing features and its likely mapping to SIMD hardware. The advantages and disadvantages of each model are summarized at the end.

The models presented here itself are software abstractions. However, the usefulness of a vector model is predicated on having an efficient mapping to SIMD hardware just as the usefulness of our task-parallel model was predicated on having an efficient mapping to multicore hardware. This paper, therefore, dips into implementation concerns, but only as a way to clarify design choices. Hardware concepts would not be part of a formal specification.

2 Motivation

As stated in N4060:

> Although SIMD vector units are common on modern microprocessors such as the x86 (AVX and SSE), ARM (NEON), Power (AltiVec), MIPS (MSA), and several others, vector instructions date back to older "long vector" super computers such as the CDC Star-100, Cray, CM-1, and CM-2. For code that can take advantage of the vector instructions, the performance benefit is often better than that of multicore programming, and is even better when the two types of parallelism are combined. It is therefore crucial that vector programming be given proper attention when considering C++ extensions for parallelism.

The most basic construct for expression vector parallelism is a vector-parallel loop. In Robert Geva and Clark Nelson's proposal to the C++ standards committee, N3831, a vector-parallel loop is distinguished from a task-parallel loop only by the ordering dependencies that are and are not allowed in each. For a vector-parallel loop, the proposal says this:

> For all expressions X and Y evaluated as part of a SIMD loop, if X is sequenced before Y in a single iteration of the serialization of the loop and i <= j, then X_i is sequenced before Y_j in the SIMD loop.

This text makes fine standardese, but it is not a descriptive model. There is not enough information there for us to judge the soundness of the model or compare it to other models. Rather, the dependencies in the excerpt above are a natural consequence of a specific mental model (the wavefront model, as described in this paper) held by the authors of the paper. In the absence of a top-down description of the vector model, these dependencies appear subtle and cryptic to anybody who is not already steeped in the same mental model. As such, this formulation encourages people to gloss over the very real differences between vector parallelism and task parallelism.

For these reasons, I have attempted in this paper to describe more complete models for vector execution. It is my hope that vector loop proposals such as those that succeed N3831 will be able to refer to this paper in their introductions and rationale. Similarly, proposals for reductions and other cross-lane operations can be related to these models.

All of the models presented here are intended to be implementable on modern SIMD hardware without being specific to any one architecture or CPU manufacturer. There is precedent for abstract descriptions of common hardware patterns. The C standard describes a fairly detailed memory and execution model that applies to most, but not all, computation hardware. For example, the memory footprint of a single `struct` is flat – all of the bytes making up the `struct` are consecutive (except for alignment and padding, which are also part of the model). The C model also specifies that integers must be represented in binary, that positive integers must have the same representation as unsigned integers, etc.. This model has withstood the test of time, describing virtually every CPU on the market, regardless of word size, instruction set, or manufacturing technology. Although I don't expect that the models described here can be quite as general, my hope is that they are general enough to describe a wide range of present and future hardware.

3 Overview of the Vector Models

This paper describes three vector models:

1. *Lockstep execution*
2. *Wavefront execution*
3. *Explicit-barrier execution*

The models presented here differ from one another in the kind of interaction that is permitted across lanes and, consequently, the cross-lane dependencies that can be expressed by the programmer. Each model is strictly weaker (makes fewer ordering guarantees) than the one before it, so, for example, an implementation that implements lockstep execution meets all of the semantic constraints of the wavefront and explicit-barrier execution models.

Each model has a different set of pros and cons, as described towards the end of this paper. It is important to note that the models, while similar in some ways, are mutually-exclusive; it is not practical to reason about a software system where multiple models are in use, nor for an implementation to generate code for multiple models (e.g., based on policy objects).

This paper does not propose any syntax. Rather, it is a description of concepts that can be rendered using many different possible language constructs. An important use of these models is as a set of fundamental concepts can be used to judge the appropriateness of future proposals for language and library constructs and to help document those constructs that are adopted.

4 Concepts that are Common to All Vector Models

The concepts of *program steps*, *vector lanes*, *control divergence*, *vector-enabled function*, and *scalar region*, described in this section, are common to all three models. Some of the details vary by model.

4.1 Program Steps

A block of code is divided into one or more *program steps* (statements and expressions), each of which can be decomposed into smaller sub-steps, recursively until we reach a primitive operation. The decomposition of a larger step into smaller steps follows the syntactic and semantic structure of the language such that, e.g., a compound statement is decomposed into a series of (nested) statements and an `if` statement is decomposed into a conditional expression, a statement to be executed in the **true** case, and a statement to be executed in the **false** case. Depending on the model, an actual proposal might need to define the granularity of a primitive step.

4.2 Vector Lanes and Gangs

In serial code, the steps within a block are executed as if one at a time by a single execution unit until control leaves the block.

In vectorized code, a single step can be executed concurrently on data items from consecutive iterations, with each data item occupying a *lane* in the registers of the vector execution unit. (The simultaneous processing of multiple lanes of data creates the illusion that there is an execution unit dedicated to each lane, but that is not always physically the case for SIMD hardware.)

Borrowing from the documentation for the Intel SPMD Program Compiler (ISPC), we will refer to a group of concurrent lanes as a *gang*. The number of lanes in a gang, VL, is an implementation quantity (possibly unspecified in the language) and might be as small as 1 for code that has not been vectorized. A vector loop executes in chunks of VL sequential iterations, with each iteration assigned to a lane. The chunks need not all be the same size (have the same VL); it is common in SIMD implementations for the first and/or last chunk to be smaller than the rest (e.g., for peeling and remainder loops).

The lanes in a gang are logically indexed from 0 to VL-1. (This indexing may or may not correspond directly to components in the hardware.) Lanes with smaller indexes within the same gang are referred to as *earlier than* or *previous to* lanes with larger indexes. Conversely, lanes with larger indexes are *later than* or *subsequent to* those with smaller indexes. In a vectorized loop, iterations are assigned to lanes in increasing order so that earlier iterations in the serial version of the loop map to earlier lanes in each gang.

In all of the models, there is neither a requirement nor an expectation that individual lanes within a gang will make independent progress. If any lane within a gang blocks (e.g., on a mutex), then it is likely (but not required) that all lanes will stop making progress.

4.3 Control-flow divergence and active lanes

Control flow *diverges* between lanes when a *selection-statement* (`if` or `switch`), *iteration-statement* (`while` or `for`), conditional operator (`&&`, `||`, or `?:`), or polymorphic function call (virtual function call or indirection through a function pointer) causes different lanes within a gang to execute different code paths. For each control-flow branch, those lanes that should execute that branch are called *active* lanes. Control flow *converges* when the control paths come back together at the end of the construct construct, e.g., at the end of an `if` statement. Although different models have different ways of dealing with control-flow divergence, it is generally true of all implementations that a large amount of control-flow divergence has a negative impact on performance. If the control flow within a loop has irreducible constructs (e.g., loops that have multiple entry points via gotos), it may be impractical for a SIMD compiler to vectorize those constructs at all.

4.4 Vector-enabled functions

A vector-enabled function (also called a simd-enabled function) is a function that, when called within a vector loop, is compiled as if it were part of the vector loop. Multiple lanes can execute the same vector-enabled function concurrently, and all of the attributes of vector execution apply to the body of the function. In Cilk Plus and OpenMP, a vector-enabled function must be specially declared (using `declspec(vector)` or `#pragma omp declare simd`, respectively) and a scalar version of that function is implicitly generated along with one or more vector versions.

In a SIMD implementation, a vector-enabled function called within a vector loop is passed arguments and returns results via SIMD registers, where each element of each SIMD register corresponds to a lane in the calling context.

4.5 Scalar regions

The language may provide a syntax to declare a segment of code within a vector loop to be a *scalar* region. This region has the following characteristics:

1. Only one lane executes the scalar region at a time.
2. It is considered a single, indivisible step, regardless of how many statements or expressions are within it; no portion of a scalar region on one lane is interleaved with any code on another lane.

In the lockstep model, an implicit scalar region must surround each call to a non-vector-enabled function within a vectorized loop. (The as-if rule applies; if the compiler can prove that a non-vector function can be vectorized without changing the observable behavior, it is free to do so.) In the wavefront and explicit-barrier models, this implicit scalar region is not necessary but might be desirable for making programs easier to reason about.

5 Barriers

5.1 Overview

A *barrier* is a point of synchronization, adding specific happens-before dependencies between steps executed on otherwise-independent lanes in a gang. Two types of barriers are described here: a *lockstep barrier* and a *wavefront barrier*. A barrier need not correspond to a hardware operation – it is an abstraction that allows us to reason about what side effects are known to have occurred, and what side effects are known to have not yet occurred. A compiler is not permitted to reorder instructions across a barrier unless it can prove that such reordering will have no effect on the results of the program.

5.2 Lockstep Barriers

Upon encountering a *lockstep barrier*, execution of an active lane blocks until all other active lanes have reached the same barrier. Put another way, all of the active lanes within a gang synchronize such that they all have the same program counter at the same time. Since the scope of a lockstep barrier is a single gang, it is incumbent on any language specification of lockstep barriers to define the gang width (i.e., VL).

A step following a lockstep barrier may depend on all lanes having completed all steps preceding the barrier. For VL greater than one, iterations of a vector loop can therefore rely on results of computations that would have been computed in "future" iterations if the loop were expressed serially. This feature is occasionally useful, especially for short-circuiting unneeded computations, but it causes the computation to proceed differently (and potentially produce different results) depending on the value of VL, and it makes it difficult to reason about a program by comparing it to a serial equivalent.

It is important to note that, for a gang size of greater than one, the semantics of a lockstep barrier require mandatory vectorization – the compiler must break the computation into gangs even if the computation proceeds on only one lane at a time. If a local variable (including any temporary variable) is needed by any step, then a separate instance of that local variable is created for each lane in the gang. Conversely, if there are no lockstep barriers (either implicit or explicit), the compiler could honor all vector semantics by executing a vector loop serially, making no attempt to distribute data across lanes.

5.3 Wavefront Barriers

Upon encountering a *wavefront barrier*, execution of an active lane blocks until all *previous* active lanes have reached the same barrier. Intuitively, no lane can get ahead of any previous lane. A wavefront barrier is strictly weaker than a lockstep barrier, so a lockstep barrier can always be inserted where a wavefront barrier is needed without breaking the semantics of a program.

A step following a wavefront barrier may depend on all previous lanes having completed all steps preceding the barrier. Unlike the case of a lockstep barrier, a step cannot depend on values that would logically be computed in the future. Thus, a program that uses only wavefront barriers would retain serial semantics and a compiler would be free not to vectorize in cases where it is not deemed beneficial or when vectorization would require heroic effort on the part of the compiler vendor.

6 The Lockstep Execution Model

6.1 Overview

Intuitively, each step in the lockstep execution model occurs simultaneously on all lanes within a gang. More precisely, there is a logical lockstep barrier before each step, with all of the lanes completing one step before any lane starts the next step. Each step is further subdivided into smaller steps, recursively, with a lockstep barrier before each sub-step until we come to a primitive step that cannot be subdivided (where *primitive step* would need to be defined in any specific proposal).

6.2 Mapping to SIMD hardware

The lockstep execution model maps directly to the SIMD instruction set of modern microprocessors. For code that cannot be translated directly into SIMD instructions the compiler must ensure that the program behaves *as if* every step, down to a specified level of granularity, were executed simultaneously on every lane. (In ISPC, for example, the level of granularity is specified as the code that executes between two *sequence points* as defined in the C standard.)
Thus, for three consecutive steps, A, B, C, that cannot be translated into SIMD instructions, the compiler must generate A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3 (or use three loops) in order to get correct execution on four lanes. Of course, this can often be optimized to something simpler, but any re-ordering done by the compiler requires inter-lane data flow analysis to ensure that the inter-lane dependency relationships continue to hold

6.3 Implementation of control-flow divergence

When lockstep execution is mapped to SIMD hardware, a *mask* of lanes is computed for each possible branch of a selection statement, conditional operator, or polymorphic function call. Each mask is conceptually a bit mask with a one bit for each lane that should take a particular branch and a zero bit for each lane that should not take that branch. The code in each branch is then executed with only those lanes having ones in the corresponding mask participating in the execution. Execution proceeds with some lanes “masked off” until control flow converges again at the end of the selection statement or conditional operator. This process is repeated recursively for conditional operations within each branch. Each conditional operation is a step and all branches within a conditional operation comprise a single step.

In theory, the compiler would be free to schedule multiple branches simultaneously. In the following example, the optimizer might condense two branches into a single hardware step by computing a vector of `+1` and `-1` values and performing a single vector addition.

```
for simd (int i = 0; i < N; ++) {
    if (c[i])
        a[i] += 1;
    else
        a[i] -= 1;
}
```

Processing of an *iteration-statement* (`while`, `do` or `for`) is similar to that of a *selection-statement*. In this case, there is single mask containing a one for each lane in which the loop condition is true (i.e., the loop must iterate again). The loop body is executed for each lane in this set, then the mask is recomputed. The loop terminates (and control flow converges) when the set of lanes with a true loop condition is empty (i.e., the mask is all zeros).

7 The Wavefront Execution Model

7.1 Overview

Unlike the lockstep model, the steps in the wavefront execution model can proceed at different rates on different lanes. However, although two lanes may execute the same step *simultaneously*, a step on one lane cannot be executed *before* the same step on an earlier lane. Intuitively, then, execution proceeds from top to bottom and left to right, producing a ragged diagonal “wavefront”. More precisely, there is a logical wavefront barrier before each step. The dependency constraints described in N3831 imply the wavefront execution model.

7.2 Mapping to SIMD hardware

Typically, steps that can be executed using SIMD instructions are executed simultaneously on all lanes. Steps that do not map cleanly to SIMD instructions are executed for one lane at a time. Since there is not a lockstep barrier before every step, the compiler can generate an arbitrarily-long series of non-vector instructions for a single lane before generating the same instructions (or using a loop) for the other lanes. Thus, for three consecutive steps, A, B, C, that cannot be translated into SIMD instructions, the compiler has the option to generate A0, B0, C0, A1, B1, C1, A2, B2, C2, A3, B3, C3, or use a single loop to maintain a wavefront. The lockstep order would also be valid. Note that there is no need to define the minimum granularity for a step, since any sized step may be executed serially without invalidating the required sequencing relationships.

7.3 Implementation of control-flow divergence

For operations that rely on SIMD instructions, control-flow divergence in the wavefront model is handled the same way as in the lockstep model. A segment of code that cannot be translated into SIMD instructions is translated into an arbitrarily-long sequence of scalar instructions which is executed once for each active lane within a branch. By executing the segment for earlier lanes before later lanes, the wavefront dependencies are automatically preserved. Thus, control-flow divergence is easier to handle in the wavefront model than in the lockstep model, though it may be a bit more complicated to describe.

8 The Explicit-Barrier Execution Model

8.1 Overview

This execution model is the most like a task-parallel execution model. All lanes can progress at different rates with no requirement that earlier lanes stay ahead of later lanes. When an inter-lane dependency is needed, an explicit barrier (either lockstep or wavefront) must be issued to cause the lanes to synchronize. Note that if lockstep barriers are omitted, serial semantics are preserved (as in the case of the wavefront model).

8.2 Mapping to SIMD hardware

As in the wavefront model, the compiler uses SIMD instructions where possible with segments of scalar instructions repeated once per lane in-between. There are often no hardware instructions corresponding to a barrier, but a barrier in the code does restrict the order of execution of steps as generated by the compiler. For example, given three consecutive steps, A, B, C, that cannot be translated into SIMD instructions, the compiler can generate A0, B0, C0, A1, B1, C1, A2, B2, C2, A3, B3, C3, or use a single loop. However, if a lockstep barrier is inserted between steps B and C, the compiler would have to generate A0, B0, A1, B1, A2, B2, A3, B3, C0, C1, C2, C3 or use two loops (AB) and (C). The compiler is given total freedom of optimization between barriers so it might generate B0 before A0 if there does not exist a direct dependency between them *in a single iteration*; it does not need to consider dependencies between iterations unless the programmer inserts a barrier.

Because the lanes are not required to stay in sync, this model is the closest to that used by SIMT (Single Instruction Multiple Thread) GPUs, where a group of hardware threads can be used almost like a gang of SIMD lanes. On a SIMT GPU, a lockstep barrier is a fairly cheap operation. A wavefront barrier would probably require issuing a lockstep barrier instruction. On cache-baesd multicore CPUs, a barrier of any sort between cores is very expensive, so this model cannot efficiently substitute for task-based parallelism.

8.3 Implementation of control-flow divergence

On SIMD hardware, for code with no lockstep barriers, control-flow divergence can be handled the same way as for the wavefront model. If a lockstep barrier is inserted, segments of unvectorized code must be broken up at the barrier, as in the case of the lockstep model. In the segments of code between barriers, the lack of implicit inter-lane dependencies also allows the compiler to re-order instructions in a way that is not possible with the other models.

On SIMT GPUs each thread acts independently and follows its own control flow. If barriers or control-flow divergence causes a pipeline stall, other hardware threads tend to smooth over the latency.

9 Varying, linear, and uniform

An expression within a vector context can be described as *varying*, *linear*, or *uniform*, as follows:

- *varying*: The expression can have a different, unrelated, value in each lane.
- *linear*: The value of the expression is linear with respect to the lane index. The difference in value between one lane and the next is the *stride*. A common special case is a *unit stride* – i.e., a stride of one – which is often more efficient for the hardware.

- *uniform*: The value of the expression is the same for all lanes.

A variable declared outside of the vector loop is *uniform* because there is only one copy for the entire loop. Loop induction variables that are incremented by a fixed amount in the loop header are *linear*. Otherwise, an expression is *varying* unless an explicit attribute or other construct indicates otherwise. The distinction between varying and linear is mostly pragmatic, having to do with performance (see below). Using data-flow analysis, the optimizer can sometimes determine that a variable is linear even if it lacks the explicit attribute.

A *linear* expression can be more efficient than a *varying* expression when it is used as an array index (integral values) or an address (pointer values). In most SIMD hardware, instructions that access consecutive memory addresses preform better than scatter/gather instructions on varying addresses.

If an expression is *uniform*, the compiler can use scalar instructions instead of vector instructions. For example, when the expression is an array index or an address, a single memory lookup is needed instead of one per lane. When the uniform expression is a conditional expression, the compiler can avoid generating code to handle control-flow divergence because all lanes will take the same branch.

Writing to a variable with uniform address can cause correctness problems if not all lanes write the same value or if one lane writes a value and another lane reads it without an appropriate barrier in between. For this reason, the rules for writing to a uniform variable need to be carefully specified in any vector language and should ideally be enforced by the compiler.

10 Lane-specific and cross-lane operations

There are situations where the simd model is exposed very directly in the code. A short list of operations that might be useful are:

- Lane index: Get the index for the current lane.
- Number of Lanes: Get the number of lanes in this chunk.
- Number of Active Lanes: Get the number of lanes in this chunk that are active within this branch of code. (Implies a lockstep barrier.)
- Compress: Put the results of a computation from a non-consecutive set of active lanes into a consecutive sequence of memory locations. (Implies a wavefront barrier.)
- Expand: Read values from a consecutive sequence of memory locations for use in a non-consecutive set of active lanes. (Implies a wavefront barrier.)
- Prefix-sum (or other parallel prefix): The value of the expression in each lane is the value of the same expression in the previous active lane summed with a lane-specific value (or combined using some other associative operation). (Implies a wavefront barrier.)
- Chunk reduction: Combine values from multiple lanes into one uniform value. (Implies a lockstep barrier.)
- Shuffle: Re-order elements across lanes. (Implies a lockstep barrier.)

Note that many of these operations, especially those that imply lockstep barriers, could be used in such a way as to break serial equivalence; i.e., cause the result of a computation to differ depending on how many simd lanes exist. Used carefully, however, these operations can produce deterministic results.

11 Comparing the Models

The lockstep model might be the most intuitive for a specific group of programmers, especially those who have been using compiler intrinsics heavily to access the SIMD capabilities of their hardware, but is arguably the least intuitive for those with experience with more general forms of parallelism. It is fairly easy to reason about, but the divergence from serial semantics could cause scalability and maintainability problems. The lockstep model limits optimization options by the compiler. Critically, the lockstep model requires *mandatory* vectorization; even when SIMD instructions cannot be used, the compiler must allocate variables and generate steps that simulate vector execution.

The wavefront model is a bit more abstract than the lockstep model and retains serial semantics. On the other hand, it has more of a feel of auto-vectorization; programmers might have a less intuitive grasp of what the compiler is generating. The wavefront model could potentially be augmented with explicit lockstep barriers (which do violate serial semantics) in order to support certain cross-lane operations and get the best of both worlds, though such a mixed model has not been demonstrated in the wild, to my knowledge. With fewer dependency constraints, the compiler has more opportunities for optimization than it does for the lockstep model but is still limited with respect to serial reordering. It may be difficult to implement the wavefront model in LLVM (see Robison14), other than by moving the vectorization phase into the front-end, impeding communication with the back-end optimizer.

The explicit barrier model is the most flexible model for both the programmer and the compiler, allowing the programmer to choose where dependencies are needed and (if both full and wavefront barriers are provided) what type of dependency is needed. The compiler is free to optimize aggressively, including reordering instructions within code segments between barriers. This model is the least imperative of the models, which may make some programmers wonder if vectorization is actually happening (as in the case of the wavefront model). The explicit barrier model maps well to SIMD CPUs and SIMD GPUs (which require no hardware barrier instructions) and might also be applicable to SIMT GPUs (which would use actual barrier instructions). Serial semantics are retained if only wavefront barriers are provided in the language. Lockstep barriers could be added to the language at the cost of serial semantics, if they are found to be desirable.

12 Next Steps

This paper makes no proposals. However, if C++ is to support vector-parallel programming, we will need to propose language constructs that are consistent with the concepts in this document. I suggest that we consider the following steps in the order specified:

1. Choose a vector execution model. Based on previous SG1 discussions and straw polls, non-mandatory vectorization is strongly desired, so that limits the choices to the wavefront or explicit barrier models **without lockstep barriers**. It bears repeating that trying to support multiple models based on individual programmer preference would be very unwise.
2. Create a syntax and semantics for a vector loop consistent with the chosen model (see N4237).
3. Add reductions to the vector loop. Reductions are the only cross-lane operations that are critical for a minimally-useful vector loop.
4. Publish a TS based on all of the above.
5. Consider adding vector-enabled functions to the TS.
6. Add features to the TS for other cross-lane operations, limiting ourselves to operations that retain serial semantics.
7. Last and least: Consider adding lockstep barriers and cross-lane operations that violate serial semantics and mandate vectorization.

At some point along the way, we will probably want to add a `vector_execution_policy` to the parallelism TS. When that should happen will depend on the staging of the various TSs and standards.

13 Conclusion

The models presented here are intended to directly express the concepts of SIMD execution as implemented by x86, ARM, Power, and other CPUs. They are more descriptive and complete than the model that has been presented to the C and C++ standards committees in the past.

The models are designed to be cohesive, so that programmer and implementation concerns flow naturally from the model descriptions. The dependency constraints described in N3831, for example, follow directly from the wavefront model presented here.

Vector parallelism is different from multicore and SIMT parallelism. It is critical to achieving maximum performance from existing CPUs and deserves our diligent attention.

14 Appendix: Matching Steps and Barriers Across Iterations

The definition of "same barrier" is simple for straight-line code: it can be defined by the program counter or by a location in the source code. In the presence of control-flow divergence, especially loops, the definition becomes more complicated. For example, given a barrier that is executed a different number of times on different lanes based on a conditional expression, how does one match a barrier encountered in one line to the "same barrier" encountered in another lane and what can be said about synchronization of code between occurrences of the barrier? Reasonably intuitive answers exist for these questions, but defining it precisely can be complex. It is interesting to note that CUDA does not allow barriers to be executed in the presence of control divergence; if any lane (thread) executes a barrier, then all of the others must execute that barrier or else deadlock results.

14.1 Numbering steps

The description below was written with the wavefront model in mind. It is incomplete and might be rewritten entirely for a future formal proposal

Although intuitively easy to grasp, linguistically we must devise a way of numbering steps (including steps in multiple iterations of inner loops) such that it is always easy to identify "corresponding" steps from separate iterations of the outer vector loop. For loop without branches, steps are simply numbered consecutively. At the start of an inner loop, we start appending a sub-step number comprising the iteration number (starting from zero) and a step number within the loop (also starting at zero). Two steps in different lanes correspond to each other if they have the same step number. One step precedes another step if its step number is lexicographically smaller. For example, the code below is shown with numbers assigned to each step. (Note that same numbers apply to each lane executing this code.):

```
0           a += b;
1           c *= a;
2.i.0       for (int i = 0;
2.i.1            i < c;
2.i.3            ++i)
2.i.2            x[b] += f(i);
3           y[b] = g(x[b]);
```

With this numbering scheme, step 2.1.1 (first step in the second iteration of the loop) precedes step 2.1.3 (third step in the same iteration) but comes after 2.0.1 (first step in the first iteration). The compiler is permitted to interleave steps from different lanes anyway it wants, provided that the wavefront respects these relationships.

15 Acknowledgments

Thank you to Robert Geva, Olivier Giroux, Arch Robison, Clark Nelson, Milind Girkar, and John Pieper for helping me clarify my thoughts and checking my facts.

16 References

N4060: *Changes to `vector_execution_policy`*, Pablo Halpern, 2014-06-11

N4071 *Working Draft, Technical Specification for C++ Extensions for Parallelism*, Jared Hoberock, editor, 2014-06-19

N3831: *Language Extensions for Vector level parallelism*, Robert Geva and Clark Nelson, 2014-01-14

N4237: *Language Extensions for Vector Loop Level Parallelism*, Robert Geva and Clark Nelson, 2014-10-13

Robison14: *Proposal for "llvm.mem.vectorize.safelen"* (Discussion thread on LLVMdev email list), http://lists.cs.uiuc.edu/pipermail/llvmdev/2014-September/077290.html.

ISPC: *Intel SPMD Program Compiler*

AVX: *Introduction to Intel(R) Advanced Vector Extensions*, https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

NEON: *ARM(R) NEON(tm) SIMD Engine*, http://www.arm.com/products/processors/technologies/neon.php

AltiVec *Altivec(tm) Technologies for Power Architecture*, http://www.freescale.com/webapp/sps/site/overview.jsp?code=DRPP

MSA: *MIPS(R) SIMD Architecture*, http://www.imgtec.com/mips/architectures/simd.asp