Changes to vector_execution_policy | N4060

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

2014-06-11

1 Abstract

The working paper for the Parallelism TS, N3989, defines the `vector_execution_policy` as a variant of the `parallel_execution_policy` with additional constraints on exceptions and synchronization. This definition assumes a stronger similarity between parallel execution and vector execution than exists in most real CPUs on the market. Because of the lack of ordering constraints, certain operations that are possible using SIMD hardware cannot be exploited using this model.

The inability to depend on certain orderings when using `parallel_execution_policy` has been known within SG1 for quite some time, but the specific ordering guarantees seemed subtle and the use cases for such guarantees seemed sparse. However, recent studies of customer code at Intel have uncovered a number of important use cases that have not previously been shared with the standards committee. It is my belief that these new use cases make a compelling case that a vector execution policy should not be treated as a refinement of the parallel execution policy.

The Parallelism TS appears to be close to moving to PDTS. Because of this schedule, it is probably too late to add a new, vector-only, policy to the WP and describe its semantics, especially as there was no paper to do so in the pre-meeting mailing for Rapperswil. Therefore, to avoid upsetting the release of the first version of TS, this paper recommends only a few tweaks to the WP. Chief among these tweaks is renaming `vector_execution_policy` to something else so that we may hold `vector_execution_policy` in reserve for a future TS.

2 The problem

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 WP for the parallelism TS, however, attempts to minimize the differences between vector and multicore (task) parallelism. This attempt satisfies a certain aesthetic and allows the model to extend to GPU parallelism, but it short-changes vector parallelism so that some important use cases cannot be expressed. The purpose of parallelism is to exploit the hardware efficiently. Given the prevalence of vector hardware, we risk falling short of our mission.

3 A brief recap of the difference between parallel and vector execution

This topic has been discussed a lot, so I will endeavor to keep this section brief. I will confine this discussion to loop-style parallelism, where the parallel execution agents are running essentially the same code. Vector parallelism is rarely effective for non-loop code.

Both vector- and task-parallelism relax the ordering guarantees of sequential code and both require a certain amount of independence between the iterations of the loop. However, where task parallelism is implemented with thread-like execution agents that, once assigned to an iteration, make independent forward progress, vector parallelism is implemented on a single thread, where forward progress is more regulated. This means that vector parallelism imposes more constraints on the programmer, but can also make additional guarantees. The current specification in N3989 adds those constraints (no exceptions, no synchronization constructs) but does not add the guarantees.

The main guarantee provided by vector parallelism is that given two iterations within a loop, execution of the logically later iteration can never “get ahead” of the earlier iteration, i.e., it will not execute a later expression than has been executed in an earlier iteration. This quality has been formalized in other proposals, and the formalism could possibly be improved. However, the purpose of this paper is not to define a vector model in detail, but to leave room for adding a complete vector model in the future.

Up until now, we have not done a good job at motivating the need for this ordering guarantee. A recent study of customer code, however, illuminated a set of operations that were vital for taking advantage of vectorization. These operations rely on the ordering guarantee as well as other aspects of vector hardware and reinforce the semantics that have been presented and largely agreed-to in the vector loop proposal, N3831.

Without going into excessive technical detail, here is a short list of some of these operations:

- Compress: Put the results of a computation from a non-contiguous set of iterations into a contiguous sequence of memory locations.
- Expand: Read values from a contiguous sequence of memory locations for use by a non-contiguous set of iterations.
- Uniform control flow: Evaluate a conditional as the logical AND or OR of values from all “lanes” at once, thus avoiding extra work if any lane detects that we are done.

All of these operations would require significant synchronization overhead in a multithreaded implementation but are extremely fast in a vector implementation. Compress and expand are beginning to show up as primitive operations in hardware whereas they would actually require blocking in a parallel implementation in order to get the elements in the correct order.

4 Proposal summary

4.1 The status quo

Some time ago, it was suggested that execution policies should be described by the following taxonomy:

- **Sequential**: No parallelism – Fewest constraints, maximum ordering guarantees.
- **Parallel**: Task Parallelism – Maximum dependency constraints, no ordering guarantees. Synchronization and exceptions permitted.
- **Vector**: Vector Parallelism – Some dependency constraints, some ordering guarantees. Synchronization and exceptions not permitted.
- **Parallel + Vector**: Task and Vector Parallelism – Union of constraints, intersection of guarantees.

The WP leaves out "Vector" and renames "Parallel + Vector" to "Vector". It was argued that one could get the effect of vector-only execution by specifying parallel+vector but restricting it to a single worker thread. This approach does not work for several reasons:

1. It impedes straight-forward analysis and is asymmetrical. We could just as easily define "Parallel" as "Vector+Parallel with only one lane" or "Sequential" as "Parallel on only one worker." It doesn't make sense to make vectorization a second-class player in the optimization game.
2. Parallelization imposes certain static constraints and provides certain static guarantees. Where else in the standard do we change static constraints based upon the *value* of a function argument?
3. It is not clear, especially in the current WP, how a compiler is expected to determine that no task-parallel constraints apply. Such knowledge is critical to vectorizing the code correctly and producing correct diagnostics (as well as avoiding incorrect diagnostics).

The only logical course is to re-introduce a vector-without-parallel execution policy that is different than vector-and-parallel. Failing that, we should rename the vector-and-parallel policy so that "parallel" is part of its name, thus leaving "vector" for future definition.

4.2 Minimal proposal

The following change would have a minimal impact on the WP and should not delay moving to PDTS.

- Rename `vector_execution_policy` to `parallel_vector_execution_policy`.
- Rename `vec` to `parvec`.

4.3 Ideal proposal

Without creating a complete vector model for now (but aiming for such a model in a future revision of the TS), it should be possible to add the following to the minimal proposal:

- Define `vector_execution_policy` with a minimal set of constraints and guarantees that distinguish it from `parallel_execution_policy`
- Define `vec` as an object of `vector_execution_policy` type.

5 Formal Wording

TBD

6 References

N3989: *Working Draft, Technical Specification for C++ Extensions for Parallelism*, Jared Hoberock, editor, 2014-05-23

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

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