Doc No: N3561
Date: 2013-03-15
Authors:
Robert Geva
email: robert.geva@intel.com
Clark Nelson
email: clark.nelson@intel.com
Intel Corp.

Semantics of Vector Loops

Introduction

In the SG1 conference call on Feb 5 2013, I presented the topic of a critical section in a vector loop. The discussion on the narrow topic itself resulted in the consensus that a critical section in a vector loops is undefined behavior. However, the discussion also led to broader interest in the semantics of a vector loop, which I was asked to present. This paper describes the semantics we propose. Note that this paper is meant as a continuation of earlier papers on vector loops, and is probably not self-contained. It does not repeat the syntax and language rules portions of the proposal.

As was stated in earlier meetings, this proposal is an attempt to codify existing practice in vector programming. While in a narrow sense, there are no existing practices of vector programming within standard C++, vector programming is broadly used in ad hoc ways. The expectations of programmers are well understood. The proposal here is not to invent new programming models. Instead, it is an attempt to codify the existing expectations of existing practitioners using C++ methodology.

Syntax disclaimer

As we stated in our previous presentations, our focus at the current stage is on functionality and semantics. We present a specific syntax for clarity, but the syntax is not an inherent part of the core capability and we welcome proposals for improvements.

Vector execution

The iterations of vector loop execute on a single thread, and consecutive iterations are grouped together and execute in chunks. Ideally, the size of the chunk should correspond to as many iterations of a sequential loop as can fit within the vector resources of the target machine. However, the chunk size can also be limited in cases where there are data dependencies across the iterations of the loop. In those cases, the results may differ if the chunk size violate the data dependencies, and therefore the developer needs the ability to limit the size of the chunk.

Vector execution requires reordering expressions from different iterations so that multiple evaluation of the same expression from different iterations can be grouped together. The semantics of vector loops are expressed in terms of allowed reorderings, rather than in terms of vector instructions.

Vector loops are unlike parallel loops. The semantics of parallel loops are that the iterations are unsequenced. The sequencing rules of vector loops provide more strict guarantees. Another interesting distinction is the chunked execution of vector loops. Loop iterations that execute together cannot make progress independent of each other, i.e. a subset of the vector lanes cannot block while others progress. Therefore, using existing constructs for critical sections will not work – they will likely deadlock. As we describe the semantics of vector loops, this interaction will become evident. Seen from the other direction, since parallel loops are expected to have well-defined behavior when critical sections are used within them, the implication is that parallel loops are not appropriate for use as vector loops. Of course, additional programming constructs have well-defined behavior in parallel loops but not in vector loops. Mechanisms such as Cilk™ Plus hyperobjects provide the ability to create linked lists in parallel loops. Vectorizing loops with non vectorizable operations may result in undefined behavior.

Syntax:

The following table lists the set of capabilities we propose for vector loops. The semantics will be described incrementally, corresponding to the list of capabilities.

Capability	Syntax	meaning
Vector loop	`simd_for( ; ; )`	Vector order of evaluation. Parallelism constructs (such as parallel loops and cilk_spawn) shall not appear in the loop body.
Limit the chunk size	`simd_for_chunk(N) ( ; ; )`	Limit the number of iterations that can be grouped together and execute in a chunk
Uniform vs. private variables	Object is declared outside vs. inside the loop	Uniform: a single object for all chunked iterations. Private: each iteration within the chunk has a private instance.
Linear induction variables	`simd_for ( ; ;` *comma separated list of increments*`)`	Each iteration within the chunk has its own value. The values are an arithmetic progression
Reductions	Cilk hyper object	There is a single object for the whole loop. The value produced by an iteration is a function of the value produced by the preceding iteration. Other uses of the value within the loop are undefined.
nosimd blocks	`nosimd { }`	Executions of the block from different iterations of the containing vector loop are not interleaved.
Elemental functions	`__attribute__(ve`	Consecutive operations of the function are chunked

`ctor)`	and execute together, as if they were a body of a vector loop

Definitions:

The *scalar elision* of a `simd_for` loop is the loop obtained by replacing `simd_for` by `for`.
The scalar elision of `simd_for_chunk(`*N*`)` is the loop obtained by replacing `simd_for_chunk` by `for` and erasing the expression (*N*).

A simd_for loop has *logical iterations* numbered 0, 1, … ,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would execute in the scalar elision of the simd_for loop.

The semantics of simd_for loop allow additional orders of evaluation. We will sometimes refer to the additionally allowed orders as “vector orders” and use “scalar order” as a retronym to refer to order of evaluation of a sequential loop as currently specified by the standard.

Notation

Capital letters stand for expressions in the source program.

X_i The evaluation of X in the ith logical iteration of the loop

A vector loop

iteration-statement:

`simd_for (` *for-init-decl* `;` *condition* `;` *expression* `)` *statement*

Consecutive iterations of the loop are grouped and execute in chunks.

Sequencing rules:

0. If X is sequenced before Y in the body of a vector loop, for each iteration i, then X_i is sequenced before Y_i.
1. For every expression X and Y evaluated as part of a vector loop, if X is sequenced before Y and i < j then X_i is sequenced before Y_j

Chunk size expression

iteration-statement:

`simd_for_chunk (` *constant-expression* `) (` *assignment-expression* `;` *condition* `;` *expression* `)` *statement*

Semantics:

The chunk size c is $1 \le c \le$ *constant-expression*.

When the chunk size is specified, the following additional rule applies:

2. For a vector loop with a chunk size of $c \geq 1$, for every expression X in the scalar elision of the vector loop, for every iteration i, X_i is sequenced before X_{i+c}.

Note: if the chunk size is not specified, the implementation chooses a size. When the chunk size is specified, the implementation is restricted to choose a size that is equal or smaller than that size. This proposal does not allow the program an explicit way to depend on the actual size that was chosen. For example, there is no syntax that allows a declaration of array of chunk-size number of elements. If the program behavior changes with different choices of chunk size (other than violating the size specified by the chunk expression) then the behavior is undefined. This applies in particular to the choice of a chunk size of one, which is scalar evaluation.

Discussion point: There may be an interest is providing a way for the program to query the actual chunk size used by the implementation, either at compile time or at run time.

A nosimd statement

statement:
  `nosimd` *statement*

The scalar elision of `nosimd` *statement* is *statement*.

Sequencing rule for the `nosimd` statement:

For every X_i and Y_j evaluated as part of a `nosimd` statement, if $i<j$ then X_i is sequenced before Y_j.

Uniform vs. Private variables

An object declared inside the lexical scope of a vector loop (private to the iteration) shall have separate storage allocated for each iteration of the chunk. Objects can be assigned values within each iteration independently of the operations in other iterations. Each iteration can assign values into its instance of the object independent of other iterations.

Objects that are declared outside the scope of the vector loop (uniform variables) are allocated according to the existing standard. An attempt to assign different values to such an object in different iterations of the same chunk leads to undefined behavior.

Linear induction variables

A linear induction variable shall be declared either as part of the loop control statement or outside of the loop. It shall be incremented as part of the increment clause of the loop. The stride value shall be loop invariant.

The increment expression shall be of one of the following forms:

++ *identifier*
identifier ++
-- *identifier*
identifier --
identifier += *stride-expression*
identifier -= *stride-expression*
identifier = *identifier* + *stride-expression*
identifier = *stride-expression* + *identifier*
identifier = *identifier* – *stride-expression*

The stride-expression may be evaluated only once. A program that depends on the number of time that a stride-expression is evaluated has undefined behavior.

Semantics: the values of the induction variables in each iteration are the same as in the scalar elision of the loop.

Note: Special treatment of induction variables is necessary in order to distinguish them from uniform variables, otherwise their increment would lead to undefined behavior.

Reductions

Like inductions, reductions require special support, in order to distinguish them from uniform object being incremented within the loop, leading to an undefined behavior. The current proposal supports reductions in a library solution, which uses other portions of the proposal, and doesn't require additional language support. For more about reducers and other hyperobjects see
http://dl.acm.org/citation.cfm?id=1584017

Function called from a vector loop

This proposal places no restriction on functions that may be called from a vector loop. We distinguish between calls to elemental and non-elemental functions. The proposal introduces the concept of an "elemental function" which executes as if its body were a part of the body of the loop. Calls to functions that are not elemental functions are evaluated within the loop according to the order of evaluation specified earlier in this paper for expressions in a simd_for loop. While the calls themselves are ordered in vector order, the functions themselves are evaluated (in scalar order) as: if F is a function called in a vector loop and i,j are iterations of the loop, if $i < j$ then F_i is sequenced before F_j.

Note: unlike many of the primitive operations, ordering a chunk of function calls next to each other does not present an opportunity to replace them with a single call, the way c additions can be replaced by a single addition. However, as the semantics are defined in terms of order of evaluation, function calls do not represent a special case. Therefore, the choice of specification for the ordering rules within the functions mostly manifests in practice in the case when the functions are inlined.

Discussion point: It is possible to choose a different semantic rule, which allows for the evaluation of expressions inside a non-elemental function called within a vector loop to be evaluated under the same ordering rules at the expressions that are within the lexical scope of the loop. This alternative would also be sound. Some obvious pros and cons: the advantage of the current proposal is that the author of the non-elemental function may not have designed for it to be called in a vector context. The disadvantage of the current proposal (and an advantage of the alternative), is that it complicates the implementation. If the non-elemental function is inlined, then the compiler will have to treat differently expressions that were originally in the lexical scope of the loop from expressions that were inlined into it from the non-elemental function.

Elemental functions

Elemental functions called from vector loops execute in chunks, corresponding to the chunk of the caller loop. The order of evaluation of expressions within a consecutive chunk of elemental functions is the same as the rules for the vector loop shown above. The number of call operations is undefined.

Note: The implementation is allowed to replace a chunk of calls to an elemental function by a single call, pass chunks of arguments and receive a chunk of return values as part of the vector loop. A call to an elemental function which is not from a vector loop is evaluated according to existing specifications. The implementation can also replace a chunk of calls by fewer calls. For example, if the chunk size of a loop is 8, the implementation is allowed to replace the 8 calls by 2 calls to the elemental function, each call executing in a chunk size of 4.

There is no new syntax associated with a call to elemental functions. The indication that a function is elemental is used when authoring the elemental function, as well as using a consistent prototype in header files. The capabilities required for authoring efficient elemental functions are the ability to qualify that parameters are uniform or linear, and to express the chunk size of the function.

The Intel compiler product supports the capabilities of elemental functions via the `__declspec` syntax for Windows and `__attribute__` syntax for Linux. In both cases, the syntax allows for additional clauses, as well as multiple attributes per function. We take advantage of both. That syntax is not being proposed for the C++ standard.

Note: Like the vector loop, the elemental function allows the specification of a chunk size. However, where in the case of the vector loop the implementation is allowed to choose a chunk size that is smaller than the one specified, in the case of elemental function the size has to be exactly the one specified. The reason is that elemental functions create linkage, and the linkage has to match between the definition and the call site.

Capability	Meaning
Elemental function	Vector order of evaluation across a chunk of consecutive calls to the function is allowed; No parallelism constructs shall appear inside the function.
A uniform parameter	The value of the argument is the same across the consecutive chunk of invocations of the function.
A linear parameter	Values of the argument in consecutive invocations of the function within the chunk differ by the value incr.
Chunk size	The number of consecutive invocations of the function that should be grouped into a single invocation. Editorial: in an implementation, this may create linkage.
Multiple versions	Multiple versions of the function are generated, each corresponding to a different set of clauses.

<u>Discussion point</u>: If the prototype of an elemental function specifies that a certain parameter is uniform, and in a given call site to that function from a vector loop, the matching argument is non-uniform, the current implementation silently generate a chunk of calls to the non-elemental version of the function. The implementation always generates code for the scalar version. A possible alternative is to fail the matching and generate a compile time error diagnostic.

Summary

This paper summarizes the set of capabilities we propose for vector programming as part of parallel programming.