Doc No.:	N4237
Date:	2014-10-10
Reply to:	Robert Geva Clark Nelson

Language Extensions for Vector loop level parallelism

Introduction

This document proposes a language extension for vector level parallel programming (vector programming) as an extension to C++. It is based on both Cilk Plus and OpenMP 4.0, which have almost exactly the same capability in regard to vector parallelism, however with keywords-based syntax instead of pragma-based syntax.

Scope: This proposal is based on an earlier proposal, N3831. However, the elemental functions portion of that proposal encountered questions regarding the integration with the type system, which are still not fully resolved. In order to allow forward progress, this proposal brings forward only the vector loop portion of the previous proposal, while calling out future work to be taken into account.

Language: The proposal does not add new "vector data types" to the language. Existing practices use regular, scalar data types and iterate over collections of them. There is no natural vector size that can be used as the size of the data, in a way that would be analogous to the sizes of scalar data types.

The proposal is based on language and relies on compilers to generate vector code. Unlike in thread parallelism, SIMD parallelism is often provided via an instruction set which is distinct from the scalar instruction set and therefore use of SIMD parallelism requires reliance on code generation.

Rigor vs. teaching: The semantics of vector execution presented here is with an attempt for mathematical rigor. It is meant for the standards committee and for implementers. It is not meant for developers. This duality is similar to that of the memory model. While most programmers would not pass a test on the nuances of the memory model, they are comfortable declaring data objects and manipulating them. The precise semantics is irrelevant for teaching programmers how to use SIMD parallelism.

Countable loops: Both parallel loops and SIMD loops, in OpenMP and in Cilk Plus, require that the loops are "countable loops". While the proposal does not suggest adding countable loops to the language as a distinct feature, the document presents the concept separately, so that it can be reused for both parallel loops and SIMD loops.

Loop grammar modifications

Change the grammar of *iteration-statement* as follows:

iteration-statement:

    `while ( ` *condition* `)` *statement*
    `do` *statement* `while ( ` *expression* `) ;`
    `for` *loop-qualifiers_{opt}* `(` *for-init-statement condition_{opt}* `;` *expression_{opt}* `)` *statement*
    `for` *loop-qualifiers_{opt}* `(` *for-range-declaration* `:` *for-range-initializer* `)` *statement*

Add the following grammar rules:

loop-qualifiers:
    `simd` *safelen-clause$_{opt}$*
safelen-clause:
    `safelen (` *constant-expression* `)`

Countable Loops

A *countable loop* is a `for` statement or range-based `for` statement that is required to satisfy additional constraints. The purpose of these constraints is to ensure that the loop's iteration count can be computed before the loop body is executed.

In a countable range-based `for` loop ([stmt.ranged] 6.5.4), the type of the `__begin` variable, as determined from the *begin-expr*, shall satisfy the requirements for a random access iterator. [*Note:* Intel has implemented most of the features in this proposal, but has not yet implemented support for a vector range-based `for` statement. — *end note*]

All the following constraints apply to a countable `for` loop. When a constraint limits the form of an expression, parentheses are allowed around the expression or any required subexpression.

Constraints on the form of the control clauses

The *condition* shall be an expression. [*Note:* A condition with declaration form is useful in a context where a value carries more information than just whether it is zero or nonzero. This is not believed to be useful in a countable loop. — *end note*] This expression shall be a comparison expression with one of the following forms:

    relational-expression < *shift-expression*
    relational-expression > *shift-expression*
    relational-expression <= *shift-expression*
    relational-expression >= *shift-expression*
    equality-expression `!=` *relational-expression*

Exactly one of the operands of the comparison operator shall be an identifier that designates an induction variable, as described below. This induction variable is known as the *control variable*.

The operand that is not the control variable is called the *limit expression*. Any implicit conversion applied to that operand is not considered part of the limit expression.

The final expression of the control clause of the loop is called the *loop-increment*; it shall be an expression with the following form:

loop-increment:
  single-increment
  loop-increment `,` *single-increment*
single-increment:
  identifier ++
  identifier −−
  ++ *identifier*
  −− *identifier*
  identifier += *initializer-clause*
  identifier −= *initializer-clause*
  identifier = *identifier* + *multiplicative-expression*
  identifier = *identifier* − *multiplicative-expression*
  identifier = *additive-expression* + *identifier*

Each comma in the grammar of *loop-increment* shall represent a use of the built-in comma operator. The *identifier* in each grammatical alternative for *single-increment* is called an *induction variable.* If *identifier* occurs twice in a grammatical alternative for *single-increment*, the same variable shall be named by both occurrences. If a grammatical alternative for *single-increment* contains a subexpression that is not an identifier for the induction variable, that is called the *stride expression* for that induction variable.

An induction variable shall not be designated by more than one *single-increment*.

[*Note:* The control variable is identified by considering the loop's condition and loop-increment together. If exactly one operand of the condition comparison is a variable, it is the control variable, and must be incremented. If both operands of the condition comparison are variables, only one is allowed to be incremented; that one is the control variable. It is an error if neither operand of the condition comparison is a variable. — *end note*]

[*Note:* There is no additional constraint on the form of the initialization clause of a countable loop. — *end note*]

Other statically-checkable constraints

There shall be no `return`, `break`, `goto` or `switch` statement that might transfer control into or out of the loop.

Each induction variable shall have unqualified integral, pointer, or copy-constructible class type, shall have automatic storage duration.

Each stride expression shall have integral or enumeration type.

The *loop count* is computed as follows. In the following table, “*var*” stands for an expression with the type and value of the loop control variable, “*limit*” stands for an expression with the type and value of the limit expression, and “*stride*” stands for an expression with the type and value of the stride expression. The loop count is computed after the loop initialization is performed, and before the control variable is modified by the loop. Operators used within the loop count expression are looked up in namespace scope.

Loop count expression and value

Form of condition	**Form of increment**			
	var*++** **++*var	***var*−−** **−−*var***	***var* += *stride*** ***var* = *var* + *stride*** ***var* = *stride* + *var***	***var* −= *stride*** ***var* = *var* - *stride***
var* < *limit ***limit* > *var***	((*limit*)−(*var*))	n/a	((*limit*)−(*var*)−1)/(*stride*)+1	((*limit*)−(*var*)−1)/−(*stride*)+1
var* > *limit ***limit* < *var***	n/a	((*var*)−(*limit*))	((*var*)−(*limit*)−1)/−(*stride*)+1	((*var*)−(*limit*)−1)/(*stride*)+1
var* <= *limit ***limit* >= *var***	((*limit*)−(*var*))+1	n/a	((*limit*)−(*var*))/(*stride*)+1	((*limit*)−(*var*))/−(*stride*)+1
var* >= *limit ***limit* <= *var***	n/a	((*var*)−(*limit*))+1	((*var*)−(*limit*))/−(*stride*)+1	((*var*)−(*limit*))/(*stride*)+1
var* != *limit ***limit* != *var***	((*limit*)−(*var*))	((*var*)−(*limit*))	((*stride*)<0) ? ((*var*)−(*limit*)−1)/−(*stride*)+1 **:** ((*limit*)−(*var*)−1)/(*stride*)+1	((*stride*)<0) ? ((*limit*)−(*var*)−1)/−(*stride*)+1 **:** ((*var*)−(*limit*)−1)/(*stride*)+1

The loop count expression shall be well-formed. When a stride expression is present, if the divisor of the division is not greater than zero, the behavior is undefined.

The type of the difference between the limit expression and the control variable is the *subtraction type*, which shall be integral. When the condition operation is !=, (*limit*)-(*var*) and (*var*)-(*limit*) shall have the same type. Each stride expression shall be convertible to the subtraction type. The loop odr-uses whatever `operator-` functions are selected to compute these differences.

For each induction variable *V*, one of the expressions from the following table shall be well-formed, depending on the operator used in its single-increment:

Operator	`++ += +`	`-- -= -`
Expression	*V* += *X*	*V* −= *X*

where *X* is some expression with the same type as the subtraction type. The appropriate `operator+=` or `operator-=` function is looked up in namespace scope and the loop odr-uses whatever operator functions are selected by these expressions.

Dynamic constraints

If an induction variable is modified within the loop other than as the side effect of its single-increment operation, the behavior of the program is undefined.

If evaluation of the iteration count, or a call to a required `operator+=` or `operator-=` function, terminates with an exception, the behavior of the program is undefined.

If *X* and *Y* are values of the control variable that occur in consecutive evaluations of the loop condition in the serialization, then the behavior is undefined if ((*limit*) − *X*) − ((*limit*) − *Y*) evaluated in infinite integer precision, does not equal the stride. [*Note:* in other words, the control variable must obey the rules of normal arithmetic. Unsigned wraparound is not allowed. – *end note*] If the condition expression is true on entry to the loop, then the behavior is undefined if the computed loop count is not greater than zero. If the computed loop count is not representable as a value of type `unsigned long long`, the behavior is undefined.

Evaluation relaxations

The stride expressions shall not be evaluated if the loop count is zero; otherwise, it is unspecified how many times the stride and limit expressions are evaluated. If execution of a loop iteration alters the value of the increment or limit expression, the behavior is undefined.

Within each iteration of the loop body, the name of each induction variable refers to a local object, as if the name were declared as an object within the body of the loop with automatic storage duration and with the type of the original object. If the loop body throws an exception that is not caught within the same iteration of the loop, the behavior is undefined, unless otherwise specified.

SIMD loops

Description

This section describes the SIMD loop portion of the vector programming extension. A SIMD loop is syntactically similar to a `for` loop, with the addition of the contextual keyword `simd` after the keyword for. The loop shall be a countable loop, as described above.

The *serialization* of a SIMD loop is the loop obtained by omitting the contextual keyword `simd` or `simd_safelen`.

Semantics

This specification relies on identifying an expression as being the same as it executes in different iterations of the loop. This includes the possibility that an expression executes in some but not all iterations of the vector loop, and the possibility that an expression executes a different number of times in different iterations of the vector loop. Although intuitively easy to grasp, linguistically we must devise a way of numbering expressions (including expressions in multiple iterations of inner loops) such that it is always possible to identify "corresponding" steps from separate iterations of the outer vector loop. For a 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.

A SIMD 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 serialization of the SIMD loop.

The order of evaluation of expression in a vector loops is a modification of the order of evaluation in a scalar and sequential loop. We assume that if an expression X is sequenced before an expression Y in some iteration of the loop, then there is no iteration of the loop in which Y is sequenced before X. With computed control flow, writing such a loop is not impossible. Vectorizing such a loop is most likely to yield results that are different from those of the scalar loop, and we specify that vectorization of loops with that behavior is an undefined behavior.

The order of evaluation of the expressions in a SIMD loop is a partial order and a relaxation of the order specified for sequential loops. Let X_i denote evaluation of an expression X in the i*th* logical iteration of the loop. The partial order is:

0. For all expressions X and Y evaluated as part of a SIMD loop, if X is sequenced before Y in any iteration of the loop and $i \leq j$ then X_i is sequenced before Y_j in the SIMD loop.

Alternatively, the single rule can be describe in two parts, one for the order of evaluation within a single iteration of the vector loop, and a second rule for the order of evaluation of expressions across different iterations:

1. If there is an iteration k of the serialization of the vector loop in which X_k is sequenced before Y_k, for all iterations n, X_n is sequenced before Y_n
2. For all expressions X and Y evaluated as part of a SIMD loop, if X is sequenced before Y in any iteration of the loop and $i < j$ then X_i is sequenced before Y_j in the SIMD loop.

SIMD loop with safelen

The *constant-expression* in simd_safelen(*constant-expression*) shall be an integer constant expression with a value greater than zero.

The value of the argument to the constant-expression is the maximum chunk size, c, for a SIMD loop. The order of evaluation of expressions within a simd loop with `safelen` is the same partial ordering as for a simd loop without `safelen` (above), with the following additional constraint:

3. For a SIMD loop with a maximum chunk size of c, for every pair of expressions X and Y, if there is an iteration k in which X_k is sequenced before Y_K then for each $i<N-c$, Y_i is sequenced before X_{i+c}.

Restrictions on SIMD loops

The behavior is undefined if any of the following language constructs appears within the body of a SIMD loop:

1. a `try` statement
2. a call to `setjmp` or `longjmp`

Note: Because the iterations of a SIMD loop are implicitly allowed to overlap, modifying any non-atomic non-local object carries the potential for unsequenced side effects and value computations (1.9 clause 15), and therefore undefined behavior.

Future Extensions

This proposal is valuable and would form a useful and useable addition to the standard. It is also consistent with some potential further additions which may be proposed later. The proposal allows writing vector loops, as long as either all the code is in the static scope of the loop. It does not support calling vector functions with vector interfaces. Elemental functions are functions with vector variants. They allow modular programming and shipping of functions in libraries, where the vector variants have vector interfaces, so performance is not lost through the interfaces.

The proposal allows writing of vector loops where the code applies the same operations on all “vector lanes” and it allows “forward” dependence pattern where a value computed within a certain iteration depend on a value that was computed in an earlier iteration. Vector execution is capable of more patterns. In fact, most of the interesting growth on the hardware side is in supporting a growing set of dependence pattern. At a minimum, practitioners expect support for reductions and probably partial sums. New developments on the hardware side will necessitate support for vector compression and expansion.

We expect that these patterns will be supported via libraries, similar to the library support for hyper objects and reductions in Cilk Plus. In order for these libraries to have vector semantics, they will be written in elemental functions.