Doc No.:	N3831
Date:	2014-01-14
Reply to:	Robert Geva Clark Nelson

Language Extensions for Vector 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 mutual capability in regard to vector parallelism, however with keywords-based syntax instead of pragma-based syntax.

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. The constructs actually being proposed are:

1. Array notations (in part II)
2. SIMD loops
3. SIMD enabled functions.

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 not yet implemented support for a parallel 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.

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 loop odr-uses whatever `operator+=` and `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

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 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:

> 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 $i \leq j$, then X_i is sequenced before Y_j in the SIMD loop.

[*Note:* In each iteration of a SIMD loop, the “sequenced before” relationships are exactly as in the corresponding serial loop. — *end note*]

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:

> For a SIMD loop with a chunk size of c, for every expression X in a single iteration of the SIMD loop, for every iteration i, X_i is sequenced before X_{i+c}.

SIMD-enabled functions

Description

A SIMD-enabled function is a function declared with one or more SIMD specifiers. The SIMD specifiers direct the compiler to generate multiple variants of the function for use with data parallel context such as array sections or within a SIMD loop. Although a call to a SIMD-enabled function from a vectorizable context may be handled specially, calls from "scalar" (non-vector) contexts are not affected by the SIMD specifier.

Conceptually, an invocation of a SIMD-enabled function from a loop or array-section context is matched against the declared variants. If a variant is found with an appropriate vector length and with matching uniform and linear arguments (see below), then that variant is called. Otherwise, the scalar variant is used. Since the scalar variant of the SIMD-enabled function can always be invoked, there are no error conditions associated with matching a variant of the SIMD-enabled function to the call site.

Syntax

[*Note:* `simd` is added as a new context-dependent keyword, like the *virt-specifiers* `override` and `final`, and it is recognized/allowed in basically the same syntactic contexts.

However,the *virt-specifiers* are semantically applicable only to virtual functions, and according to 9.2p8, shall not appear in any other kind of declaration.

In addition to that restriction, the use of *virt-specifier-seq* in the grammar is limited to *member-declarator* and *function-definition*. Given the semantic restriction, the grammatical limitations appear to be superfluous.

Because a SIMD specifier is applicable to any sort of function, including a lambda, and to enable the more general use of context-dependent keywords in declarations, a new *cdk-specifier* non-terminal is added to the top-level grammars for declarators. The existing *virt-specifiers*, and the new *simd-specifier*, are moved under this category. — *end note*]

Change existing grammar rules as follows:

lambda-declarator:

    `(` *parameter-declaration-clause* `)` `mutable`$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$ *cdk-specifier-seq*$_{opt}$

declarator:

    ptr-declarator *cdk-specifier-seq*$_{opt}$

    noptr-declarator parameters-and-qualifiers trailing-return-type *cdk-specifier-seq*$_{opt}$

abstract-declarator:

    ptr-abstract-declarator *cdk-specifier-seq*$_{opt}$

    noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers trailing-return-type* *cdk-specifier-seq*$_{opt}$

    abstract-pack-declarator *cdk-specifier-seq*$_{opt}$

function-definition:

    attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator* ~~*virt-specifier-seq*$_{opt}$~~ *function-body*

member-declarator:

    declarator ~~*virt-specifier-seq*$_{opt}$~~ *pure-specifier*$_{opt}$

    declarator brace-or-equal-initializer$_{opt}$

    identifier$_{opt}$ *attribute-specifier-seq*$_{opt}$ `:` *constant-expression*

~~*virt-specifier-seq*:~~

    ~~*virt-specifier*~~

    ~~*virt-specifier-seq virt-specifier*~~

Add the following new grammar rules:

cdk-specifier-seq:

    cdk-specifier

    cdk-specifier-seq cdk-specifier

cdk-specifier:

    virt-specifier

    simd-specifier

simd-specifier:

    `simd`

`simd (` *simd-function-clauses*$_{\text{opt}}$ `)`

simd-function-clauses:

simd-function-clause

simd-function-clauses `,` *simd-function-clause*

simd-function-clauses simd-function-clause

simd-function-clause:

simdlen-clause

uniform-clause

linear-clause

inbranch-clause

The SIMD specifier consists of the contextual keyword `simd` and an optional list of clauses:

Example:

```
void vec_add (float *r, float *op1, float *op2, int k)
simd(uniform(r,op1,op2) linear(k:1)) simd
{ r[k] = op1[k] + op2[k]; }
```

This function can be called in three different ways:

A scalar context, to add two values and place the result in a scalar variables:

```
vec_add(*x, *a, *b, 0);
```

In this case, the scalar variant of vec_add is matched to the caller.

A vector context where the actual pointer arguments to `r`, `op1` and `op2` are fixed across calls to the function and the actual values matched into the argument `k` are an arithmetic sequence with a step of 1

```
for (int n = 0; n < N; ++n) vec_add(res, op1, op2, n);
```

In this case, the vector variant corresponding to the specifier `simd(uniform(r,op1,op2)` is matched to the caller.

A vector context with no known relationship between the values within the vectors of arguments

```
for (int n = 0; n < N; ++n)
vec_add(res[idout[n]], op1[id_in1[n]], op2[id_op2[n]], n);
```

In this case, the vector variant corresponding to the specifier `simd` is matched to the caller.

Semantics

The semantics of a SIMD-enabled function differ from a normal function in sequencing and in constraints on the program, as follows:

Invocations of the SIMD-enabled function in consecutive iterations of a SIMD loop or for consecutive elements of an array section are unsequenced with respect to one another.

The body of a SIMD-enabled function is required to conform to the same constraints as the body of a SIMD loop.

Implementation note: The compiler will typically generate a scalar variant of the function, and two vector variants per SIMD specifier. The first vector variant processes multiple array elements at a time through the use of vector registers and SIMD instructions. The second vector variant does the same, but additionally takes an implicit mask argument, which disables processing of some of the vector lanes. The compiler calls the second variant and supplies the implicit mask when the function is called from a conditional statement for which the condition might differ between iterations. The programmer can use an `inbranch` clause to suppress one of the two vector variants if they are not needed. The compiler may suppress generation of any variant (including the scalar variant) if it knows that the variant will not be called. In a separate-compilation environment, such knowledge might require access to all compilation units that may use the SIMD-enabled function.

For each `simd` specifier, one vector variant of the vectorized function is created. The `simd` specifier and its associated clauses are considered part of the function interface.

A SIMD-enabled function shall not have a dynamic exception specification.

In a SIMD specifier, no parameter shall be the subject of more than one clause of kind `uniform` or `linear`.

The `simdlen` clause

simdlen-clause:

`simdlen (` *constant-expression* `)`

A SIMD specifier shall not contain more than one `simdlen` clause.

Every vector variant of a SIMD-enabled function has a vector length (VL). If the `simdlen` clause is used, the VL is the value of the argument of that clause. Otherwise the VL is defined by the implementation's ABI.

The `uniform` clause

uniform-clause:

`uniform (` *param-list* `)`

param-list:

parameter-name

param-list `,` *parameter-name*

parameter-name:

identifier

`this`

The `uniform` clause declares one or more parameters to have an invariant value for all invocations of the vector variant in the execution of a single SIMD loop or array-section expression.

The identifier in a parameter-name shall match the name in a parameter-declaration of the function to which the containing clause applies; the parameter shall have arithmetic or pointer type. The keyword `this` shall not be used as a parameter-name except with a non-static member function.

The `linear` clause

linear-clause:

    `linear (` *param-list* `)`

    `linear (` *param-list* `:` *linear-step* `)`

linear-step:

    parameter-name

    constant-expression

The `linear` clause declares one or more parameters to have values that increase or decrease linearly in consecutive invocations of the function in the execution of a single SIMD loop or array-section expression.

If a *linear-step* consists of a single identifier matching the name of a parameter, or in the declaration of a non-static member function is the keyword `this`, the *linear-step* is interpreted as referring to the parameter; otherwise, the *linear-step* shall satisfy the requirements of a *constant-expression*. A *constant-expression* in a *linear-step* shall be an integer constant expression. A parameter referenced as a *linear-step* shall be the subject of a `uniform` clause. An omitted *linear-step* is equivalent to a *linear-step* of 1.

The `inbranch` clauses

inbranch-clause:

    `inbranch`

    `notinbranch`

Hint to suppress generation of the other variant.

Implementation note: In typical implementations, both variants are generated and the clause serves to suppress generation of an unnecessary variant. By default, for every SIMD specifier, two variants are generated: one especially suitable for conditional invocation (i.e. masked), and another suitable only for unconditional invocation (i.e. unmasked). If all invocations are conditional, generation of the unmasked variant can be suppressed using the `inbranch` clause. Similarly, if all invocations are unconditional, generation of the masked variant can be suppressed using the `notinbranch` clause. A SIMD specifier shall not have both an `inbranch` clause and a `notinbranch` clause.

Restrictions on SIMD-enabled functions and SIMD loops

The behavior is undefined if any of the following language constructs appears within the body of a SIMD-enabled function, or in a SIMD loop:

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

If execution of a function called from a SIMD-enabled function or SIMD loop terminates with an exception or a call to `longjmp`, the behavior is undefined.

Note: Because invocations of a SIMD-enabled function, and iterations of a SIMD loop, are implicitly allowed to be unsequenced, 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.