Explicit Vector Programming with OpenMP 4.0 SIMD Extensions

Xinmin Tian†, Bronis R. de Supinski‡
†Intel Corporation, Santa Clara, California USA
‡Lawrence Livermore National Laboratory, Livermore, California, USA
OpenMP Architecture Review Board (ARB)
Email: xinmin.tian@intel.com, bronis@llnl.gov

Abstract — Modern CPU and GPU processors with on-die integration of SIMD execution units for achieving higher performance and power efficiency have posed challenges to use the underlying SIMD hardware (or VPU, Vector Processing Unit) effectively. Wide vector registers and SIMD instructions — single Instructions operating on Multiple Data elements packed in wide registers such as AltiVec [2], SSE, AVX[10] and MIC [9] — pose a compilation challenge that is greatly eased through programmer hints. While many applications implemented using OpenMP [13, 17], a widely accepted industry standard for exploiting thread-level parallelism, to leverage the full potential of today’s multi-core architectures, no industry standard has offered any means to express SIMD parallelism. Instead, each compiler vendor has provided its own vendor-specific hints for exploiting vector parallelism, or programmers relied on the compiler’s automatic vectorization capability, which is known to be limited due to many compile-time unknown program factors.

To alleviate the situation for programmers, the OpenMP language committee added SIMD constructs to OpenMP to support vector-level parallelism. These new constructs provide a standardized set of SIMD constructs for programmers who no longer need to use non-portable, vendor-specific vectorization intrinsics or directives. In addition, these SIMD constructs provide additional knowledge about the code structure to the compiler and allow for a better vectorization that blends well with parallelization. To the best of our knowledge, the OpenMP 4.0 specification is the first industry standard that includes explicit vector programming constructs for programmers.

This paper describes the C/C++ and Fortran SIMD extensions for explicit vector programming available in the OpenMP 4.0 specification. We explain the semantics of SIMD constructs and clauses with simple examples. In addition, a set of explicit vector programming guidelines and programming examples are provided in Section 3 and 4 to help programmers to write efficient SIMD programs for achieving a higher performance. Section 5 presents a case study of achieving a ~200x performance speedup using OpenMP 4.0 PARALLEL and SIMD constructs on Intel® Xeon Phi™ coprocessors. Section 6 summarizes this paper.

Keywords: OpenMP, Explicit Vectorization, SIMD programming model, Multicore

1. Explicit Vector Programming Rationale

While the OpenMP loop construct defines loop-level parallelism on for (C/C++) or do loops (Fortran), re-using these existing constructs to perform explicit vector programming for vectorization would not work. The example in Figure 1.1 shows a code fragment that can (partially) be executed in parallel, but cannot be vectorized by the compiler without additional knowledge about vectorization and safety information for SIMD vector length selection.

```
#define N 1000000
float x[N][N], y[N][N];
#pragma omp parallel
{
#pragma omp for
for (int i=0; i<N; i++) {
#pragma omp simd safelen(18)
    for (int j=18; j<N-18; j++) {
      x[i][j] = x[i][j-18] + sinf(y[i][j]);
      y[i][j] = y[i][j+18] + cosf(x[i][j]);
    }
}
```

**Figure 1.1. An example of using worksharing omp for and omp simd constructs**

In Figure 1.1, there are two issues that inhibit vectorization when only the loop (for) construct is used: i) the loop construct cannot easily be extended to the j-loop, since it would violate the construct’s semantics; ii) the j-loop contains a lexically backward loop-carried dependency that prohibits vectorization. However, the code can be vectorized for any given vector length for array y and for vectors shorter than 18 elements for array x. The solution to address these issues is to extend OpenMP with a set of constructs and clauses that enable programmers to identify SIMD loops explicitly in addition to parallel loops.

Primeur Magazine 2014-11-10
http://primeurmagazine.com/weekly/AE-PR-12-14-32.html
Vectorizing loops is one of the most significant and common ways of exploiting vector-level parallelism among optimizing compilers. However, one major obstacle for vectorizing loops is how to handle function calls without fully inlining these functions into loops that should be vectorized for vector execution on SIMD hardware. The key issue arises when the loop contains a call to a user-defined function `sfoo()` as shown in Figure 1.2. As the compiler does not know what `sfoo()` does, unless the function is inlined at the call site, the compiler will fail to auto-vectorize the caller loop as shown in Figure 2.

To overcome this issue, OpenMP allows programmers to annotate user-defined functions that can be vectorized with the `declare simd` directive. The compiler parses and accepts these simple annotations of the function. Thus, programmers can mark both loops and functions that are expected to be vectorized as shown in the Figure 1.3.

```
#pragma omp declare simd
extern float sfoo(float);

void invokesfoo(float *restrict a, float *restrict x, int n)
{  
  for (int i=0; i<n; i++)  a[i] = sfoo(x[i]);
}
```

Figure 1.2. An example of loop and user-defined function not vectorized.

The functions `invokesfoo()` and `sfoo()` do not have to reside in the same compilation unit (or in the same file). However, if the function `invokesfoo()` is compiled with a compiler that supports SIMD annotations, the scalar function `sfoo()` needs to be compiled with the compiler that supports SIMD annotation as well. The vectorization of `invokesfoo()` creates a call to `vfoo()` (i.e., `simdized sfoo()`), and the compilation of annotated `sfoo()` needs to provide a vector variant `vfoo()`, in addition to the original scalar `sfoo()`.

With the SIMD vector extensions, the function can be vectorized beyond the conventional wisdom of loop-nest (or loop) only vectorization,

```
#pragma omp declare simd
float sfoo(float x) {
  ...  } // function body
```

Figure 1.3. An example of loop and user-defined function vectorized

The functions `invokesfoo()` and `sfoo()` do not have to reside in the same compilation unit (or in the same file). However, if the function `invokesfoo()` is compiled with a compiler that supports SIMD annotations, the scalar function `sfoo()` needs to be compiled with the compiler that supports SIMD annotation as well. The vectorization of `invokesfoo()` creates a call to `vfoo()` (i.e., `simdized sfoo()`), and the compilation of annotated `sfoo()` needs to provide a vector variant `vfoo()`, in addition to the original scalar `sfoo()`.

With the SIMD vector extensions, the function can be vectorized beyond the conventional wisdom of loop-nest (or loop) only vectorization,

```
#pragma omp declare simd
__m128 vfoo(__m128 vx)
{  
  ...  } // vector body
```

Figure 1.4. An example of converting a scalar function to a vector function and executing on SIMD hardware
With the compiler code-conversion of a scalar function to its vector function as shown in Figure 3, its SIMD vector execution is illustrated in Figure 4. Essentially, the compiler generates a vector function based on the original scalar function and its vector annotations. At the caller site, after vectorization, every four scalar function invocations ($sfoo(x0)$, $sfoo(x1)$, $sfoo(x2)$, $sfoo(x3)$) are replaced by one vector function invocation $vfoo(<x0...x3>)$ with vector input argument $<x0...x3>$ and vector return value $<r0...r3>$ through vector registers. The input argument $x0$, $x1$, $x2$ and $x3$ can be packed into one 128-bit vector register for argument passing, and the result $r0$, $r1$, $r2$, and $r3$ can be packed into one 128-bit vector register for returning the value. The execution of $vfoo()$ exploits SIMD parallelism.

2. SIMD VECTOR EXTENSIONS TO OPENMP

To facilitate explicit vector programming of OpenMP programs, a new set of pragmas (or directives) were added in the OpenMP 4.0 specification. Ordinary, but annotated, C/C++ and Fortran complex loops and functions enable SIMD execution on modern microprocessors, so programmers do not have to rely on compiler data dependency analysis and vendor-specific compiler hints (e.g., #pragma ivdep supported by IBM*, Cray*, Intel® compilers). This section describes the syntax and semantics of these extensions.

The OpenMP SIMD extensions have restrictions. For example, C++ exception handling code, any call to throw exceptions, and longjmp and setjmp function calls are not allowed in the lexical or dynamic scope of SIMD functions and loops. Other restrictions include:

- The function or subroutine body must be a structured block.
- The execution of the function or subroutine, when called from a SIMD loop, cannot result in the execution of an OpenMP construct.
- The execution of the function or subroutine cannot have any side effects that would alter its execution for concurrent iterations of a SIMD chunk.
- A program that branches into or out of the function is non-conforming.

Detailed syntax restrictions and language rules of OpenMP SIMD extensions can be found in the OpenMP 4.0 specification [13].

2.1 SIMD Constructs for Loops

The basis of OpenMP 4.0 SIMD extensions is the `simd` construct for `for` (C/C++) and `do` loops (Fortran), as shown in Figure 1. This new construct instructs the compiler to vectorize the loop. The `SIMD` construct can be applied to a loop to indicate that the iterations of the loop can be divided into contiguous chunks of a specific length and, for each chunk, multiple iterations can be executed with multiple SIMD lanes concurrently within the chunk while preserving all data dependencies of the original serial program and its execution. The syntax of the `simd` construct is as follows:

```
C/C++:
#pragma omp simd [clause[[], clause] ...] new-line
for-loops

Fortran:
!$omp simd [clause[[], clause] ...] new-line
do-loops
(!$omp end simd )
```

The `simd` construct closely follows the idea and syntax of the existing loop construct. It supports several clauses that we cover in Section 2.4. The loop header of the associated `for` or `do` loop must obey the same restrictions as for the loop construct. These restrictions enable the OpenMP compiler to determine the iteration space of the loop upfront and to distribute it accordingly to fit the vectorization.

The `simd` construct can be also applied to the existing work-sharing loop construct (and parallel loops) to form loop SIMD construct (and combined parallel loop SIMD construct), which specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel by threads in the team.

The loop (`for` or `do`) SIMD construct will first distribute the iterations of the associated loop(s) across the implicit tasks of the parallel region in a manner consistent with any clauses that apply to the loop construct. The resulting chunks of iterations will then be converted to a SIMD loop in a manner consistent with any clauses that apply to the `simd` construct. The effect of any clause that applies to both constructs is as if it were applied, more details see Section 2.8.3 and Section 2.10 in the OpenMP 4.0 specification [13].

2.2 SIMD Constructs for Functions

Beyond the `simd` construct for loops, OpenMP 4.0 introduces the `declare simd` construct, which can be applied to a function (C, C++ and Fortran) or a subroutine (Fortran) to enable the creation of one or more versions that can process multiple instances of each argument using SIMD instructions from a single invocation from a SIMD loop. There may be multiple `declare simd` directives for a function (C, C++, Fortran) or subroutine (Fortran). The syntax of the `declare simd` construct is as follows:

```
C/C++:
```
#pragma omp declare simd [clause[{}, clause] ...] new-line
[  #pragma omp declare simd [clause[{}, clause] ...] new-line]

function definition or declaration

Fortran:

!$omp declare simd (proc-name) [clause[{}, clause] ...] new-line

The declare simd construct instructs the compiler to create SIMD versions of the associated function. The expressions appearing in the clauses of this directive are evaluated in the scope of the arguments of the function declaration or definition.

2.3 SIMD Execution Model

In the SIMD execution model, a SIMD loop conceptually has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations. The logical numbering denotes the sequence in which the iterations would be executed if the associated loop(s) were executed with no SIMD instructions. A SIMD function has a logical number of invocations numbered 0,1,...,VL-1, where VL is the number of SIMD lanes. The logical numbering denotes the sequence in which the invocations would be executed if the associated function was executed with no SIMD instructions. In other words, a legal SIMD program and its execution should obey all original data dependencies among iterations (or invocations) and dependencies within an iteration of the serial program and its execution.

Given a SIMD hardware unit with 8 lanes, i.e., 8 elements of float type data can be packed into a single SIMD register for 8-way SIMD execution, a chunk of iterations is mapped onto SIMD lanes and starts running concurrently on those SIMD lanes. The group of running SIMD lanes is called a SIMD chunk. The program counter is a single program counter shared by the SIMD lanes; it points to the single instruction to be executed next. To control execution within a SIMD chunk, the execution predicate is a per-SIMD-lane boolean value that indicates whether or not side effects from the current instruction should be observed. For example, if a statement were to be executed with an "all false" predicate, it should have no observable side-effects.

Upon entering a SIMD context (i.e., a SIMD loop or a SIMD function) in an application, the execution predicate is "all true" and the program counter points to the first statement in the loop or function. The following two statements describe the required behavior of the program counter and the execution predicate over the course of execution of a SIMD context:

- The program counter will have a sequence of values that correspond to a conservative execution path through statements of the SIMD context, wherein if any SIMD lane executes a statement, the program counter will pass through that statement;
- At each statement through which the program counter passes, the execution predicate will be set such that its value for a particular SIMD lane is "true" if and only if the SIMD lane is to be enabled to execute that statement.

The above SIMD execution behavior provides the compiler some latitude. For example, the program counter is allowed to skip a series of statements for which the execution predicate is "all false" since the statements have no observable side-effects. In reality, the control flow in the program can be diverging, which leads to reductions in SIMD efficiency (and thus performance) as different SIMD lanes must perform different computations. The SIMD execution model provides an important guarantee about the behavior of the program counter and execution predicate: the execution of SIMD lanes is maximally converged. Maximal convergence means that if two SIMD lanes follow the same control path, they are guaranteed to execute each program statement concurrently while preserving all original data dependencies. If two SIMD lanes follow diverging control paths, they are guaranteed to re-converge as soon as possible in the SIMD execution.

2.4 Clauses for SIMD constructs

To refine the execution behavior of the SIMD construct further, OpenMP provides clauses for programmers to write optimal vector programs that explicitly specify the data sharing, data movement, visibility, SIMD length, linearity, uniformity and memory alignment.

2.4.1 Data Sharing Clauses

The private, lastprivate and reduction clauses control data privatization and sharing of variables for a SIMD execution context. The private clause creates an uninitialized vector for the given variables. For SIMD function arguments, by default, a parameter has a distinct storage location or value for each its instance among hardware SIMD lanes, i.e., it is private. The lastprivate clause provides the same semantics but also copies out the values produced from the last iteration to outside the loop. The reduction clause creates a vector copy of the variable and horizontally aggregates partial values of that vector into the original a scalar variable.

2.4.1.1 The uniform Clause

A parameter that is specified with the uniform clause represents an invariant value for a chunk of concurrent invocations of the function in the execution of a single SIMD loop. It effectively is shared across SIMD lanes of vector execution. Specifying function parameters that are shared across the SIMD lanes as uniform allows the vectorizer to generate optimized code for scalar (or unit-stride) memory loads (or stores), and optimal control flow. For instance, when a base address is uniform and the offset is a linear unit stride, the compiler can generate faster unit-stride vector memory load/store instructions (e.g., movaps or movups supported on Intel® SIMD hardware) instead of generating gather/scatter instructions. Also when a test condition for a control flow decision is based on a uniform compiler, the compiler can exploit that all running code instances will follow the same path at that point to save the overhead of masking checks and control flow divergence.

Primeur Magazine 2014-11-10
http://primeurmagazine.com/weekly/AE-PR-12-14-32.html
2.4.1.2 The linear Clause

A variable (or a parameter) specified in a linear clause is made private to each iteration (or each SIMD lane) and has a linear relationship with respect to the iteration space of the SIMD execution context. A variable cannot appear in more than one linear clause, or in a linear clause and also in another OpenMP data clause. A linear-step can be specified for a variable in a linear clause. If specified, the linear-step expression must be invariant during the execution of the region associated with the construct. Otherwise, the execution results in unspecified behavior. If linear-step is not specified, it is assumed to be 1.

Under a SIMD loop context, the value of the linearized variable on each iteration of the associated loop(s) corresponds to the value of the original variable before entering the construct plus the logical number of the iteration times linear-step. The value corresponding to the sequentially last iteration of the associated loops is assigned to the original variable. If the associated code does not increase the variable by linear-step in each iteration of the loop then the behavior is undefined.

Under a SIMD function context, a parameter referenced in a linear-step must be the subject of a uniform clause. No parameter of a vector function can be the subject of more than one uniform or linear clause. For a linear parameter, if the corresponding argument values in consecutive iterations (in the serial version of the program) do not differ by linear-step, the behavior is undefined.

2.4.2 The aligned Clause

Memory access alignment is important since most platforms can load (or store) aligned data much faster than unaligned data accesses, especially SIMD (vector type) data. However, compilers often cannot detect alignment properties of data across all modules of a program, or dynamically allocated memory (or objects), so they must conservatively generate code that uses only unaligned loads/stores. Hence, the aligned(variable[:alignment] [,variable[:alignment]]) clause allows programmers to express alignment information (i.e., number of bytes that must be a constant positive integer value) to the compiler. For each variable in the list of the aligned clause, the programmer can specify an alignment value, if no optional alignment value is specified, an implementation defined default alignment for SIMD instructions on the target platforms is assumed.

2.4.3 The safelen Clause

If a safelen clause is specified, then no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than its value. The parameter of the safelen clause must be a constant positive integer expression. The number of iterations that are executed concurrently at any given time is implementation defined but guaranteed not to exceed the value specified in the safelen clause. A different SIMD lane will execute each concurrent iteration. Each set of concurrent iterations is a SIMD chunk.

2.4.4 The simdlen Clause

For a function annotated with declare simd, when a SIMD version is created, the number of concurrent elements packed for each argument of the function is determined by the vector length specified in the simdlen clause, or, by default, is selected by the compiler for a given SIMD hardware. When the specified vector length is a multiple of the hardware SIMD length, the compiler may apply double-pumping, triple-pumping, or quad-pumping that emulates longer vectors by fusing multiple vector registers into a larger logical vector register. The parameter of the simdlen clause must be a constant positive integer expression. In practice, it should be a multiple of the hardware SIMD length. Otherwise, the number of elements packed for each argument of the function is implementation defined.

2.4.5 Inbranch and Notinbrnach Clauses

The inbranch clause indicates that a function will always be called under conditions in the SIMD loop / function. The notinbranch clause indicates that a function will never be called under conditions of a SIMD loop / function. If neither clause is specified, then the function may or may not be called from inside a conditional statement of a SIMD loop / function. By default, for every SIMD variant function declared, two implementations are provided: one especially suitable for conditional invocation (i.e., inbranch version) with a predicate, and another especially suitable for unconditional invocation (i.e., notinbranch version).

If all invocations are conditional, generation of the notinbranch version can be suppressed using the inbranch clause. Similarly, if all invocations are unconditional, generation of the inbranch version can be suppressed using the notinbranch clause. Suppressing either inbranch or notinbranch version of a SIMD function helps to reduce code size and compilation time. By default, both inbranch and notinbranch versions of vector variants have to be provided, since the compiler cannot determine that the original scalar function is always called under condition (or not).

3. SIMD Programming Guideline

This section provides several programming guidelines for OpenMP programmers to develop correct and high performance SIMD programs using SIMD extensions in the OpenMP 4.0 specification.

3.1 Ensure SIMD Execution Legality

If programmers apply the SIMD construct to loops so that they are transformed into SIMD loops, they guarantee that the loops can be partitioned into chunks such that iterations within each chunk can correctly execute concurrently using SIMD instructions. To provide
this guarantee, the programmers must use the `safelen` clause to preserve all original data dependencies or remove data dependencies that prevent SIMD execution by specifying data sharing clauses such as `private`, `lastprivate` or `reduction`. Recall that:

- A loop annotated with a SIMD pragma/directive has logical iterations numbered 0, 1, ..., `N-1` where `N` is the number of loop iterations;
- The logical numbering denotes the sequence in which the iterations would execute if the associated loop(s) executed with no SIMD instructions;
- If the `safelen(L)` clause is specified, then no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than `L`.

Programmers can use two mechanisms to ensure that an OpenMP canonical loop can legally be transformed for SIMD execution.

**Mechanism 1**: Given a logical loop iteration, use the `safelen` clause to prohibit loop-carried lexically backward dependencies between any two iterations in the chunk. For example, the chunk is `[k, k+1, k+2, k+3]`, then the results produced by the iteration `k`, must not be consumed by the iteration `k+1`, `k+2` and `k+3`.

```c
#pragma omp simd safelen(4)
for (k=5; k<N; k++) {
    a[k] = a[k-4] + b[k];
}
```

**Figure 3.1.1. A code example of SIMD loop with `safelen` clause**

The `safelen` in Figure 3.1.1 asserts the loop may be safely SIMDized with any vector length (`VL`) less than 5. Assuming `VL = 4` is selected by the compiler, its SIMD execution is shown below. The `iteration-n` denotes the serial execution logical iteration number, `vector iteration-n` denotes the SIMD execution logical vector iteration number, `r-n` denotes general scalar register, `vr-n` vector register.

<table>
<thead>
<tr>
<th>Serial execution</th>
<th>SIMD execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteration-0</td>
<td>vector iteration-0</td>
</tr>
<tr>
<td>load r0, a[0]</td>
<td>simdload vr0, a[0..3]</td>
</tr>
<tr>
<td>load r1, b[4]</td>
<td>simdload vr1, b[4..7]</td>
</tr>
<tr>
<td>add r0, r1</td>
<td>simdadd vr0, vr1</td>
</tr>
<tr>
<td>store a[4], r0</td>
<td>simdstore a[4..7], vr0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Serial execution</th>
<th>SIMD execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteration 4</td>
<td>vector iteration-1</td>
</tr>
<tr>
<td>load r0, a[4]</td>
<td>simdload vr0, a[4..7]</td>
</tr>
<tr>
<td>load r1, b[8]</td>
<td>simdload vr1, b[8..11]</td>
</tr>
<tr>
<td>add r0, r1</td>
<td>simdadd vr0, vr1</td>
</tr>
<tr>
<td>store a[8], r0</td>
<td>simdstore a[8..11], vr0</td>
</tr>
</tbody>
</table>

**Figure 3.1.2. Execution samples of SIMD loop with `safelen` clause**

In the Figure 3.1.2, from the serial execution of logical iterations of the loop, the result produced by `iteration-0 store a[4]` is carried over to `iteration-4 load a[4]`; the result produced by `iteration-1 store a[5]` is carried over to `iteration-5 load a[5]`, and so on. In addition, the lexical order of `store a[k]` to load `a[k-4]` is backward, thus, this loop has a loop-carried lexically backward dependency between iteration `iteration-(k)` and `iteration-(k+4)` where `k=0, 1, 2, ..., N-4`. Thus, this loop cannot be vectorized with `VL>4`. Since the programmer specified `safelen(4)`, the compiler can correctly vectorize the loop, as the above table shows for the SIMD execution of vector `iteration-0` and vector `iteration-1`.

**Mechanism 2**: Given a logical loop iteration space, use data sharing clauses to ensure that no two iterations in any chunk have `write-write` or `write-read` conflicts. The example below demonstrates how the `private` clause can be used to achieve this effect.

```c
{ float x;
  #pragma omp simd private(x)
  for (k = 0; k<N; k++) {
    x = a[k];
    b[k] = foo(x+a[k+1]);
  }
}
```

**Figure 3.1.3. A code example of SIMD loop with `private` clause**

In the Figure 3.1.3, the op may be safely vectorized with any `VL < N` by privatizing the variable `x`. However, if `x` is not privatized for each SIMD lane, there will be a `write-write` and `write-read` conflict involving `x`.

Primeur Magazine 2014-11-10
http://primeurmagazine.com/weekly/AE-PR-12-14-32.html
3.2 Effective Use of the uniform and linear Clauses

The uniform clause directs the compiler to generate code that passes the parameter’s value (or address if it is a pointer) via a scalar register instead of a vector register. The linear clause for a scalar parameter (or variable) directs the compiler to generate code that loads/stores its value using a scalar register instead of a vector register. For a pointer, the clause directs the compiler to generate a unit-stride (or the explicitly indicated stride in linear-step) load/store vector instruction (e.g., movups) to load/store data to a vector register.

A typical case is that the base address of a memory access is uniform, and the index (or offset) has the linear property. Putting them together, the compiler generates linear unit-stride memory load/store instructions to obtain performance gains. Given the example below, the function SqrtMul is marked with omp declare simd annotation,

```c
#pragma omp declare simd uniform(op1) linear(k) notinbranch
double SqrtMul(double *op1, double op2, int k)
{
    return (sqrt(op1[k]) * sqrt(op2));
}
```

**Figure 3.2.1. A CODE EXAMPLE OF SIMD FUNCTION WITH UNIFORM AND LINEAR CLAUSES**

Given the example in Figure 3.2.1, the compiler generates the following vector SqrtMul function on the Core_i7 processor, for the faster SIMD vector function. The uniform(op1) and linear(k:1) attributes allow the compiler to pass the base address in the eax register and the initial offset value in the ecx register for the 32-bit memory address computation, and then, use one movups instruction to load two 64-bit floating-point data (a[k] and a[k+1]) to a 128-bit XMM register, as shown in Figure 3.2.2.

```c
;; Generated vector function for
;; #pragma omp declare simd uniform(op1) linear(k:1) notinbranch
PUBLIC _ZGVxN4uvl_SqrtMul.P
; parameter 1: eax ; uniform op1
; parameter 2: xmm0 ; xmm0 holds op2_1 and op2_2
; parameter 3: ecx ; linear k with unit stride
movups xmm1, XMMWORD PTR [eax+ecx*8] ; xmm1 holds op1[k] and op1[k+1]
sqrtpd xmm0, xmm0 ; vector_sqrt(xmm0)
sqrtpd xmm2, xmm1 ; vector_sqrt(xmm1)
mulpd xmm0, xmm2 ; vector_multiply
ret
```

**Figure 3.2.2. SSE4.2 CODE GENERATED FOR THE SIMD FUNCTION WITH UNIFORM AND LINEAR CLAUSES**

If the programmer omits the uniform and linear clauses, the compiler cannot determine that the memory loads/stores of all SIMD lanes have the same base address and that their offset is a linear unit-stride. Thus, the compiler must use XMM registers for passing op1, op2, and k for the scalar function SqrtMul under a SIMD execution context, as the example shows in Figure 3.2.3. For the memory address computation, the compiler generates memory load instruction movq for loading op1_1[x] to xmm3 register low quadword and movhpd for loading op1_2[k2] to xmm3 register high quadword in order to perform vector execution of vector_sqrt(xmm3) using the vector instruction sqrtpd. Calling this version provides much lower performance, even if the function invocation at call site passes in a uniform memory address op1 and a linear unit-stride value k.

```c
;; Generated vector function for #pragma omp declare simd notinbranch
PUBLIC _ZGVxN4uvv_SqrtMul.P
; parameter 1: xmm0 ; vector_op1 holds op1_1 and op1_2
; parameter 2: xmm1 ; vector_op2 holds op2_1 and op2_2
; parameter 3: xmm2 ; vector_k holds k1 and k2
pslld xmm2, 3 ; vector_k*8 is index value
paddd xmm0, xmm2 ; vector_op1 + vector_k*8
movd eax, xmm0 ; load op1_1 + k1*8 to EAX
psblw xmm2, xmm0, 238 ; shift to get opt1_2 + k2*8
movd edx, xmm2 ; load op1_2 + k2*8 to EDX
sqrtpd xmm0, xmm1 ; vector_sqrt(xmm1)
movq xmm3, DWORD PTR [eax] ; load op1_1[k1] to xmm3 low quadword
movhpd xmm3, DWORD PTR [edx] ; load op1_2[k2] to xmm3 high quadword
sqrtpd xmm3, xmm3 ; vector_sqrt(xmm3)
mulpd xmm0, xmm3 ; vector_multiply
ret
```

**Figure 3.2.3. SSE4.2 CODE GENERATED FOR THE SIMD FUNCTION WITHOUT UNIFORM AND LINEAR CLAUSES**

The vector variant function generated from the small kernel program in Figure 3.2.4 that uses the uniform and linear clauses produces a 1.62x speedup over one that omits the clauses, when it is compiled with –xSSE4.2 option and –DSIMDGOPT using the Intel® C++ compiler. The generated SIMD vector executable runs on an Intel® Core™ i7 processor with a 64-bit Linux OS installed.

Primeur Magazine 2014-11-10
http://primeurmagazine.com/weekly/AE-PR-12-14-32.html
```c
#include <stdio.h>
#include <stdlib.h>
#define M 1000000
#define N 1024

void init(float a[])
{ int i;
  for (i = 0; i < N; i++) a[i] = (float)i*1.5;
}

float checksum(float b[])
{ int i;
  float res = 0.0;
  for (i = 0; i < N; i++) res += b[i]; return res;
}

#pragma omp declare simd simdlen(8)
#ifdef SIMDOP
#pragma omp declare simd linear(op1) uniform(op2) simdlen(8)
#endif
float fSqrtMul(float *op1, float op2) {
  return sqrt(*op1)*sqrt(op2);
}

int main(int argc, char *argv[])
{ int i, k; float a[N], b[N];
  float res = 0.0f; init(a);
  for (i = 0; i < M; i++) {
    float op2 = 64.0f + i*1.0f;
    #pragma omp simd
    for (k=0; k<N; k++) {
      b[k] = fSqrtMul(&a[k], op2);
    }
    res = checksum(b); printf("res = %.2f\n", res);
  }
}
```

**Figure 3.2.4. An Example of Using uniform and linear Clauses**

### 3.3 Understanding Vector Length Selection

If the vector length (VL) is not directly specified by programmers using the `simdlen(VL)` clause with a `declare simd` directive, the compiler determines VL based on the physical vector width of the target instruction set and the characteristic type. This choice generates the most efficient SIMD vector code. For example, on Intel® architectures, VL selection uses the following rules:

- If the target processor has an XMM vector architecture (i.e., no YMM vector support) and the characteristic type of the function is `int`, VL is 4.
- If the target processor has Intel® AVX (Advanced Vector Extensions) YMM support
  - VL is 4 if the characteristic type of the function is `int` (Integer vector operations in Intel® AVX are performed on XMM).
  - VL is 8 if the characteristic type of the function is `float`.
  - VL is 4 if the characteristic type of the function is `double`.

For applications that do not require many vector registers, higher performance may be seen if the program is compiled to a doubled vector width — 8-wide for SSE using two XMM registers and 16-wide for AVX using two YMM registers. This method can lead to significantly more efficient execution due to greater instruction level parallelism and amortization of various overhead over more program instances. For other workloads, it may lead to a slowdown due to higher register pressure. Trying both approaches using the `simdlen` clause for key kernels may be worthwhile.

### 3.4 Memory Access Alignment

Alignment optimization is important for current SIMD hardware. Given architecture trends of increasing vector register widths, its importance will grow. OpenMP 4.0 provides the `aligned(n)` clause so programmers can express alignment information. For compilers to generate optimal SIMD code on current Intel® systems, n may be 8, 16, 32, or 64 to specify 8B, 16B, 32B, or 64B alignment.

For instance, a good array element access alignment is 16-byte alignment for Intel® Pentium™ 4 to Core™ i7 processors, 32-byte alignment for Intel® AVX processors, and 64-byte alignment for Intel® Xeon Phi™ coprocessors. Given an example below:
void arrayref(float * restrict x, float *y, int n, int n1) {
    __assume(n1%8=0);
    #pragma omp simd aligned(y:32)
    for (int k=0; k<n; k++) {
        x[k] = x[k] + y[k] + y[k+n1] + y[k-n1];
    }
}

FIGURE 3.4.1. A SIMD LOOP EXAMPLE OF USING ALIGNED CLAUSE

In the example shown in Figure 3.4.1, the array x is marked as 32-bytes align and the pointer y is marked as 32-bytes align. The memory reference offset n1 is asserted with “mod 8 = 0”. These annotations tell the compiler that all vector memory references are 32B aligned.

3.5 Struct and Multi-dimensionial Array

The vectorization for scalar and unit-stride memory accesses has been effectively supported by SIMD architectures and modern compilers [1, 2, 5]. However, vectorizing for structure and multi-dimension-al array accesses normally results in uses of non-unit strided-load / store and gather / scatter instruction supported in the SIMD hardware to handle non-unit stride and irregular memory accesses.

A practical way for programmers to achieve effective vector-parallelism through explicit vector programming is to convert Array of Structures (AOS) to Struct of Arrays (SOA), change the access order of array dimensions, and then apply SIMD vectorization. For C/C++ array accesses, SIMD should apply to the inner-most dimension; for Fortran array accesses, SIMD should apply to the outer-most dimension. The guideline is to re-shape arrays or to shuffle array access in order to achieve unit-stride memory accesses.

4. Explicit Vector Programming Examples

This section provides several explicit vector programming examples in C/C++ and Fortran. In practice, compilers may not vectorize loops when they are complex or have potential dependencies, even though the programmer is certain the loop will execute correctly as a vectorized loop. Sophisticated compilers can do runtime tests and multi-versioning for simple cases, but the code size and the compile time increase. The SIMD construct can be used to address these issues. Application programmers can use add SIMD construct to assure the compiler that the loop can be vectorized.

The second example (written in both OpenMP C++ and Fortran) in Figure 4.2 shows that *ioff is unknown at compile time, so the compiler has to assume *ioff could be a either negative or positive integer value for each invocation of function star. Also the compiler does not know if a, b and c are aliased. For example, if a and c are aliased and *ioff = -2, then this loop has a loop-carried lexically backward dependency, so it is not vectorizable. However, if a programmer can guarantee *ioff is a positive integer, the loop can be vectorized even if a and c are aliased. The programmer can use a SIMD construct to guarantee this property.

FIGURE 4.1. A SIMD LOOP EXAMPLE WITTEN IN C++ AND FORTRAN

The example in Figure 4.1 (written in both OpenMP C++ and Fortran) shows that *ioff is unknown at compile time, so the compiler has to assume *ioff could be a either negative or positive integer value for each invocation of function star. Also the compiler does not know if a, b and c are aliased. For example, if a and c are aliased and *ioff = -2, then this loop has a loop-carried lexically backward dependency, so it is not vectorizable. However, if a programmer can guarantee *ioff is a positive integer, the loop can be vectorized even if a and c are aliased. The programmer can use a SIMD construct to guarantee this property.

References:
[1, 2, 5]
```c
int fib( int n )
{
    if (n <= 2)
        return n;
    else {
        return fib(n-1) + fib(n-2);
    }
}

int main(void)
{
    for (int i=0; i < N; i++) b[i] = i;
    #pragma omp simd
    for (int i=0; i < N; i++) {
        a[i] = fib(b[i]);
    }
    printf("Done a[%d] = %d
", N-1, a[N-1]);
    return 0;
}
```

**Figure 4.2. A SIMD PROGRAM OF USING DECLARE SIMD AND SIMD CONSTRUCT**

Although the call to fib in main is not under the condition, the programmer can manually inline the top level call in the loop to fib, which would allow the use of the inbranch clause. is to be inlined by the Most modern compilers would perform this inlining automatically so fib is always called under the condition, as the example assumes. Either choice would instruct the compiler to generate only the masked vector version, which would reduce compile time and code size.

5. CASE STUDY: SEAMLESS INTEGRATION OF SIMD AND THREADING

OpenMP 4.0 provides an effective model to exploit both thread- and vector-level parallelism to leverage the power of modern processors. For instance, Intel® Xeon Phi™ coprocessors require that both thread- and vector-level parallelisms are exploited in a well-integrated way. While the parallelization topic is beyond the scope of this chapter, we highlight that the SIMD vector extensions can be seamlessly integrated with threading in OpenMP 4.0, either using composite/combined constructs (parallel for simd) or using them separately at different loop levels via OpenMP compiler support.

Figure 5.1 shows a Mandelbrot example that computes a graphical image representing a subset of the Mandelbrot set (a well-known 2-D fractal shape) out of a range of complex numbers. It outputs the number of points inside and outside the set.

```c
#pragma omp declare simd uniform(max_iter) simdlen(32)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{
    // Computes number of iterations(count variable)
    // that it takes for parameter c to be known to
    // be outside mandelbrot set
    uint32_t count = 1; fcomplex z = c;
    while ((cabsf(z) < 2.0f) && (count < max_iter)){
        z = z * z + c;
        count++;
    }
    return count;
}
```
The function `mandel` in the example is a hot function and a candidate for SIMD vectorization, so we can annotate it with `declare simd`. At the caller site, the hot loop is a double nested loop. The outer loop is annotated with `parallel for` for threading, and the inner loop is annotated with `simd` for vectorization as shown in Figure 5.1. The guided scheduling type is used to balance the load across threads since each call to `mandel` does a varying amount of work in terms of execution time due to the exit of the white loop. The performance measurement is conducted on an Intel® Xeon Phi™ Coprocessor with the configuration information provided in Table 5.1.

```
<table>
<thead>
<tr>
<th>System Parameters</th>
<th>Intel® Xeon Phi™ Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chips</td>
<td>1</td>
</tr>
<tr>
<td>Cores/Threads</td>
<td>61 and 244</td>
</tr>
<tr>
<td>Frequency</td>
<td>1 GHz</td>
</tr>
<tr>
<td>Data caches</td>
<td>32 KB L1, 512 KB L2 per core</td>
</tr>
<tr>
<td>Power Budget</td>
<td>300 W</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>79.36 MB</td>
</tr>
<tr>
<td>Memory Technology</td>
<td>GDDR5</td>
</tr>
<tr>
<td>Memory Speed</td>
<td>2.75 (GHz) (5.5 GT/s)</td>
</tr>
<tr>
<td>Memory Channels</td>
<td>16</td>
</tr>
<tr>
<td>Memory Data Width</td>
<td>32 bits</td>
</tr>
<tr>
<td>Peak Memory Bandwidth</td>
<td>352 GB/s</td>
</tr>
<tr>
<td>SIMD vector length</td>
<td>512 bits</td>
</tr>
</tbody>
</table>
```

Table 5.1. Target System Configuration

Figure 5.2 shows that the SIMD vectorization alone (options −mmic −openmp −std=c99 −O3) delivers a ~16x speedup over the serial execution. The OpenMP parallelization delivers a 62.09x speedup with Hyper-Threading OFF (61 threads using 61 cores) and a 131.54x speedup with Hyper-Threading ON (244 threads using 61 cores and 4 HT threads per core) over the serial execution. The combined parallelized and vectorized execution delivers a 2067.9x speedup with 244 threads. The performance scaling from 1-thread to 61-threads is close to linear. In addition, Hyper-Threading support delivers a ~2x performance gain by comparing the 244-thread speedup with the 61-thread speedup, which is better than the well-known 20%-30% expected performance gain from Hyper-Threading technology since the workload has little computing resource contention and the 4 HT threads hide latency well.

6. SUMMARY

A rich body of compiler development explores how to exploit vector-level parallelism for modern CPU [9, 10] and GPU [8] cores with powerful SIMD hardware support [1, 2, 3, 4, 5, 7, 8, 12, 14, 15] through auto-vectorization. However, modern SIMD architectures pose new constraints such as data alignment, masking for control flow, non-unit stride memory accesses and the fixed-length nature of SIMD vectors. Although, significant effort has been directed in the past decade towards these challenges [1, 5, 7, 14], automatic vectorization often still fails to vectorize application programs or to generate optimized SIMD code due to reasons such as compile-time unknown loop trip count, memory access patterns or strides, alignment and control flow complexity. To overcome these reasons, the programmer had to perform low-level SIMD intrinsic programming or to write inline ASM code in order to utilize SIMD hardware resources effectively [6].

Driven by the increasing prevalence of SIMD architectures in modern CPU and GPU processors, OpenMP 4.0 leveraged Intel’s explicit SIMD extensions [11, 16] to provide an industry standard set of high-level SIMD vector extensions [13]. These extensions that form a thin abstraction layer between the programmer and the hardware that the programmer can use to harness the computational
power of SIMD vector units without the low productivity activity of directly writing SIMD intrinsics or inline ASM code. With these SIMD extensions, compiler vendors can now deliver excellent performance on modern CPU and GPU processors.

REFERENCES


