PUTTING VECTOR PROGRAMMING TO WORK WITH OPENMP* SIMD

For Intel® Xeon® Processors, Intel® Xeon Phi™ Coprocessors, and Intel® GPUs

Xinmin Tian, *Principal Engineer;* Hideki Saito, *Principal Engineer;* Milind Girkar, *Senior Principal Engineer;* Serguei Preis, *Principal Engineer;* Sergey Kozhukhov, *Compiler Engineer;* and Alejandro Duran, *Software Engineer,* Intel Corporation

On-die integration of single-instruction/multiple data (**SIMD**) execution units in modern CPUs and GPUs has posed challenges for higher utilization of the underlying SIMD hardware due to its appealing power consumption-to-performance ratio. Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors provide a rich set of SIMD instructions (gather/scatter, shuffles, FMA, permutations, etc.) and wider vector registers (256 bit AVX/AVX2 and 512 bit MIC/AVX512). The compilation challenge for SIMD hardware can be eased with programmer hints; however, there was no industry standard offering a way to express SIMD parallelism. Instead, each compiler vendor has provided its own hints for exploiting vector parallelism, or programmers have relied on the compiler's automatic **vectorization** capability, which is limited due to program factors unknown at compile time.

To alleviate this situation for programmers, the OpenMP language standard committee, based on a proposal by Intel, added SIMD constructs to OpenMP to support vector programming. These new standardized constructs for programmers avoid the need to use nonportable, vendor-specific vectorization intrinsics or directives. In addition, these SIMD constructs provide additional knowledge about the code structure to the compiler and allow for better vectorization that blends well with parallelization.

This article describes the C/C++/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, explicit vector programming guidelines and programming examples are provided to help programmers write efficient SIMD programs. A case study shows how to achieve a ~2000x speedup using OpenMP 4.0 PARALLEL and SIMD constructs on Intel Xeon Phi coprocessors. We also show speedup ranging from 2.28x to 6.67x measured with a set of workloads using SIMD language extensions on a Xeon® processor E3-1270 (Haswell) system.

BLOG HIGHLIGHTS

The Compiler Is to Blame for Everything

BY **ANDREY KARPOV** >

Many programmers are very fond of blaming the compiler for different errors. Let's talk about it.

Are you sure? When a programmer tells you that the compiler causes an error, it is a lie in 99 percent of cases. When you start investigating the problem, you usually find out the following reasons:

- **An array overrun**
- **An uninitialized variable**
- **A misprint**
- **A synchronization error in a parallel program**
- **A nonvolatile variable used**
- **Code leading to undefined behavior**

Many went through fixing such errors. Many read about them. But it doesn't prevent them from blaming the compiler for all sins again and again. Each time it seems that it's exactly it which is guilty.

The compiler, of course, might also contain errors. But this probability is very small unless you use some exotic compiler for a microcontroller. During many years of working with Visual C++* I saw only once that it had generated an incorrect assembler code.

Read more >

Explicit Vector Programming Rationale

Loop **vectorization** is one of the most common ways of exploiting vector-level parallelism among optimizing compilers.[1, 2, 3, 4, 9, 10] However, one major obstacle to vectorizing loops is how to handle function calls without fully inlining these functions into loops that should be vectorized for execution on **SIMD** hardware. The key issue arises when the loop contains a call to a user-defined function `seqfun()` as shown in **Figure 1**. As the compiler does not know what `seqfun()` does, unless the function is inlined at the caller site, the compiler will fail to autovectorize the caller loop as shown in **Figure 1**.

```
extern float seqfun(float);
void callseqfoo(float *restrict a, float *restrict x, int n)
{ int i;
   for (i=0; i<n; i++) a[i] = seqfun(x[i]);
}
   sh-4.1$ icc -nologo -c -restrict -opt-report-phase:vec -opt-report-file=st
   derr vecfun.c
   Begin optimization report for: callseqfun(float * restrict , float *__re
   strict__, int) Report from:
   Vector optimizations [vec]
   LOOP BEGIN at vecfun.c(4,4)
         remark #15543: loop was not vectorized: loop with function call
         not considered an optimization candidate. [ vecfun.c(4,31) ]
   LOOP END
```

1 Example of a loop and user-defined function not vectorized

To address 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 **Figure 2**.

```
#pragma omp declare simd notinbranch
extern float seqfun(float) {
   // function body
}
void callseqfun(float *restrict a, float *restrict x, int n)
{ #pragma omp simd
   for (int i=0; i<n; i++) a[i] = seqfun(x[i]);
}

sh-4.1$ icc -qopenmp -nologo -c -restrict -opt-report-phase:vec -opt-re-
port-file=stderr vecfun.c
   Begin optimization report for: seqfun..xN4v(float)
         Report from: Vector optimizations [vec]
   remark #15301: FUNCTION WAS VECTORIZED    [ vecfun.c(3,1) ]

   Begin optimization report for: callseqfun(float * restrict ,
   float * restrict , int) Report from:
         Vector optimizations [vec]
   LOOP BEGIN at vecfun.c(10,4)
         remark #15301: OpenMP SIMD LOOP WAS
   VECTORIZED LOOP END
```

2 Example of loop and user-defined function that has been vectorized

The functions `callseqfun()` and `seqfun()` do not have to reside in the same compilation unit (or in the same file). However, if the function `callseqfun()` is compiled with a compiler that supports SIMD annotations, the scalar function `seqfun()` needs to be compiled with the compiler that supports SIMD annotation as well to achieve the expected SIMD execution. The vectorization of `callseqfun()` creates a call to `vecfun()` (i.e., simdized `seqfun()`), and the compilation of annotated `seqfun()` needs to provide a vector variant `vecfun()`, in addition to the original scalar `seqfun()`. With the SIMD vector extensions, the function can be vectorized beyond the conventional wisdom of loop-nest-only (or loop-only) vectorization.

Serial execution				SIMD execution
Iteration-0	**Iteration-1**	**Iteration-2**	**Iteration-3**	**Vector Iteration-0**
call seqfun(x[0]) -> r0	call seqfun(x[1] -> r1	call seqfun(x[2]) -> r2	call seqfun(x[3]) -> r3	call vecfun(x[0…3]) -> vecreg0
store a[0], r0	store a[1], r1	store a[2], r2	store a[3], r3	simdstore a[0…3], vecreg0

3 Example of converting a scalar function to a vector function and executing on SIMD hardware

The compiler code transformation of a scalar function call to `float seqfun(float x)` to its vector function call `_m128 vecfun(_m128 vecreg)` in vectorizing the `omp simd` loop of `callseqfun()` is illustrated in **Figure 3**. Essentially, the compiler generates vector variant functions based on the original scalar function and its vector annotations. At the caller site, after vectorization, every four scalar function invocations `(seqfun(x0), seqfun(x1), seqfun(x2), seqfun(x3))` are replaced by one vector function invocation `vecfun(<x0…x3>)` with a vector input argument `<x0…x3>` and vector return value `vecreg0 = <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 `vecfun()` exploits vector-level parallelism.

Another rationale for having explicit vector programming is that while the OpenMP loop construct defines thread-level parallelism on *for* (C/C++) or *do* loops (Fortran), reusing these existing constructs to perform explicit vector programming for vectorization would not work. The example in **Figure 4** 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 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] + expf(y[i][j]);
         y[i][j] = y[i][j+18] + logf(x[i][j]);
   }
}
```

4 Example of using worksharing OMP for and OMP **SIMD** constructs

In **Figure 4**, there are two issues that inhibit vectorization when only the loop `(for)` construct is used:

- The loop construct can't easily be extended to the j-loop, since it would violate the construct's semantics.
- The j-loop contains a loop-carried, lexically backward 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`. Therefore, to address these issues, a set of constructs and clauses are added to OpenMP that enable programmers to exploit vector-level parallelism by annotating SIMD function and loops explicitly in addition to parallel regions and loops.

SIMD Vector Language Extensions

To facilitate explicit vector programming, a new set of pragmas (or directives) was added in the OpenMP 4.0 specification.[8] Ordinary, but annotated, C/**C++** and Fortran 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, and 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 following:

- The function or subroutine body must be a structured block.
- The execution of the function or subroutine, when called from a SIMD loop, can't result in the execution of an OpenMP construct.
- The execution of the function or subroutine can't 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 nonconforming.

Detailed syntax restrictions and language rules of OpenMP SIMD extensions can be found in the OpenMP 4.0 or OpenMP 4.1 (draft version) specification.[8]

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 on vectorizing 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 later in the Clauses for SIMD Constructs section. 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 also be 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. For more details, see Section 2.8.3 and Section 2.10 in the OpenMP 4.0 specification.

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 to process multiple instances of each argument using SIMD instructions from a single invocation of a SIMD loop. There may be multiple declare SIMD directives for a function (C, C++, and 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.

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.

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 one shared by the SIMD lanes; it points to the single instruction to be executed next. To control the 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 SIMD function) in an application, the execution predicate is "all true" for *for* SIMD loop and notinbranch SIMD functions 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 reconverge as soon as possible in the SIMD execution. At a merge point of control flow, we can merge the SIMD executionmask and continue—again, as an as-if execution mode.

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.

Data Sharing Clauses

The private, last private, 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 instance among hardware SIMD lanes (i.e., it is private). The last private 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 scalar variable.

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 quantity, 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.

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 a 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.

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.

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.

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.

Inbranch and notinbranch 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 the 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 must be provided, since the compiler cannot determine that the original scalar function is always called under condition (or not).

The processor clause. The processor clause, which is Intel's SIMD extension for Intel® architecture, directs the compiler to create a vector variant based on original scalar function and vector annotations for the specified target processor. The default processor (or ISA class: XMM, YMM, ZMM) selection is performed based on the vector ABI (Application Binary Interface) specification.[12] The implicit or explicit processor- or architecture-specific flag in the compiler command line (e.g., Intel® compiler –xSSE4.2 option for Linux*) does not have the control on ISA class selections. The target processor clause specifies both the vector ISA that the compiler is allowed to use and the width of the vectors.

```
processor-clause: processor ( processor-name )
processor-name:
   pentium_4
   pentium_4_sse3
   … … … …
   core_i7_sse_4_2   (SSE4.2)
   core_4th_gen_avx  (Haswell)
```

Implementation note: Every compiler and hardware vendor can define and support its own processor-names.

Composite Loop SIMD Construct

The loop SIMD construct (e.g., `#pragma omp for simd`) is a composite construct[8] of the work-sharing loop and SIMD loop, which specifies that the iterations of one or more associated loops first will be distributed across threads that already exist in the team, and then the resulting iterations for each thread will 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 to both constructs separately, except the collapse clause, which is applied once. See an example in **Figure 5**:

```
float a[M][N], b[M][N], c[M][N];
#pragma omp for simd collapse(2)
for (m=0; m<M; m++) {
   for (n=0; n<N; n++) {
   c[m][n] = c[m][n] + a[m][n] * b[m][n];
   }
}
```

5 Example of a composite loop SIMD construct

In **Figure 5**, a doubly nested loop is specified with the composite loop SIMD construct. Semantically, the programmer tells the compiler this loop nest can be executed by multiple threads and SIMD instructions concurrently. By applying the collapse clause, this loop nest is converted to a loop with M*N iterations, which increases the iteration chunk size for parallel and SIMD execution. For instance, if M=640 and N=16, num-threads = 64, each thread will get a 160-iteration chunk. Intel® Xeon® AVX512 processors and Intel Xeon Phi coprocessors can execute this 160-iteration chunk as a SIMD loop of 10 vector iterations with vector length 16. **Figure 6** shows the generated pseudo SIMD code.

```
... ...
simdized_for (k=simd_chunk_start; k<simd_chunk_start+160; k+=16) {
simd_c[k:16] = simd_c[k:16] + simd_a[k:16] * simd_b[k:16];
}
... ...
```

6 Pseudo SIMD code of the composite loop SIMD construct

SIMD Vector Programming Guidelines

This section provides several guidelines for programmers to write correct and high-performing vector programs using SIMD extensions.

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, last private, 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 follow two high-level programming guidelines to ensure that an OpenMP canonical loop can legally be transformed for SIMD execution.

Guideline 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, if the chunk is [*k*, *k+1*, *k+2*, *k+3*], then the results produced by the iteration *k* in statement *S* must not be consumed by the iteration *k+1*, *k+2*, and *k+3* in the statement *S* itself or the statements above *S*.

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

7 Example of an SIMD loop with the safelen clause

The safelen clause in **Figure 7** asserts the loop may be safely SIMDized with any vector length (VL) less than 5. Assuming the VL=4 as 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, and *vr-n* is vector register.

Serial execution				SIMD execution
Iteration-0	**Iteration-1**	**Iteration-2**	**Iteration-3**	**Vector Iteration-0**
load r0, a[0]	load r0, a[1]	load r0, a[2]	load r0, a[3]	simdload vr0, a[0…3]
load r1, b[5]	load r1, b[6]	load r1, b[7]	load r1, b[8]	simdload vr1, b[5…8]
add r0, r1	add r0, r1	add r0, r1	add r0, r1	simdadd vr0, vr1
store a[5], r0	store a[6], r0	store a[7], r0	store a[8], r0	simdstore a[5…8], vr0
Iteration-4	**Iteration-5**	**Iteration-6**	**Iteration-7**	**Vector Iteration-1**
load r0, a[4]	load r0, a[5]	load r0, a[6]	load r0, a[7]	simdload vr0, a[4…7]
load r1, b[9]	load r1, b[10]	load r1, b[11]	load r1, b[12]	simdload vr1, b[9…12]
add r0, r1	add r0, r1	add r0, r1	add r0, r1	simdadd vr0, vr1
store a[9], r0	store a[10], r0	store a[11], r0	store a[12], r0	simdstore a[9…12], vr0

8 Execution samples of a SIMD loop with the safelen clause

In **Figure 8**, from the serial execution of logical iterations of the loop, the result produced by *iteration-0 store a[5]* is carried over to *iteration-5 load a[5]*; the result produced by *iteration-1 store a[6]* is carried over to *iteration-6 load a[6]*, and so on. In addition, the lexical dependency order of store *a[k]* to load *a[k-5]* is backward; thus, this loop has a loop-carried, lexically backward dependency between iteration *iteration-(k)* and *iteration-(k+5)* where *k*=0, 1,2,…,*N*-5. Thus, this loop cannot be vectorized with VL>5. Since the programmer specified safelen(5), the compiler can correctly vectorize the loop, as **Figure 8** shows for the SIMD execution of *vector iteration-0* and *vector iteration-1*.

Guideline 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.

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

9 Example of an SIMD loop with the private clause

In **Figure 9**, the loop may be safely vectorized with any *VL<N* by privatizing the variable *w*. However, if *w* is not privatized for each SIMD lane, there will be a *write-write* and *write-read* conflict involving *w*.

Effective Use of the Uniform and Linear Clauses

The uniform clause can be used for the parameter where all SIMD lanes observe the same value (i.e., uniform valued parameter). It 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.

Guideline 3: The uniform clause is most effective when the (uniform valued) parameter is used as part of the uniform base address computation for memory references or used in evaluating uniform control flow (i.e., when consumed as scalar value). In many other cases, scalar to vector broadcast operation can be hoisted out of the caller side loop by avoiding the use of uniform clause. Think carefully. The linear clause for a scalar parameter (or variable) directs the compiler to generate code to implicitly or explicitly produce a linear sequence of values from a given scalar input value. A linear parameter value is passed via scalar register instead of a vector register.

Guideline 4: Use of a linear clause for variables with linear update in a SIMD loop is not optional. Missing a linear clause when it is needed can lead to an unexpected behavior.

Guideline 5: Use of a linear clause for a parameter of a SIMD function is optional. It is most effective when the (linear valued) parameter is used as part of vector memory reference address computation (unit stride vector load/store and strided vector load/store). In many other cases, vector induction value generation can be performed more efficiently on the caller side by avoiding the use of linear clause.

A typical use case of uniform/linear clauses 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 get good performance gains. Given the example below, the function `SqrtMul` is marked with `pragma omp declare simd` annotation.

```
#pragma omp declare simd uniform(op1) linear(k) notinbranch
processor(core_4th_gen_avx)
double SqrtMul(double *op1, double op2, int k)
{
   return (sqrt(op1[k]) * sqrt(op2));
}
```

10 Example of a SIMD function with uniform and linear clauses

Given the example in **Figure 10**, the compiler generates the following vector `SqrtMul` function on the Core-AVX2 (Haswell) 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 rax register and the initial offset value in the ecx register for the memory address computation, and then uses one `vsqrtpd` instruction to compute four 64-bit floating-point data (a[k], a[k+1], a[k+2], and a[k+3]) to a 256-bit YMM register, as shown in **Figure 11**.

```
;; Generated vector function for
;; #pragma omp declare simd uniform(op1) linear(k:1) notinbranch
processor(core_4th_gen_avx)
   _ZGVYN4uvl_7SqrtMulPddi
   ; parameter 1: rax  ; uniform op1
   ; parameter 2: ymm0 ; ymm0 holds 4 values of op2_1, op2_2, op2_3 and op2_4
   ; parameter 3: ecx  ; linear k with unit stride
   movslq %ecx, %rcx
   vsqrtpd %ymm0, %ymm1    ; vector_sqrt(ymm0)      => ymm1
   vsqrtpd (%rax,%rcx,8),%ymm2 ; vector_sqrt(op1[k:3])    => ymm2 vmulpd
   %ymm2, %ymm1, %ymm0 ; vector multiply: ymm2 * ymm1 => ymm0 ret
```

11 AVX2 code generated for the SIMD function with uniform and linear clauses

If the programmer omits the uniform and linear clauses, the compiler can't 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 YMM registers for passing op1, op2, and k for the scalar function `SqrtMul` under a SIMD execution context, as the example shows in **Figure 12**. For the memory address computation, the compiler generates a SIMD gather instruction vgatherqpd for loading double precision op1_1[k1], op1_2[k2], op1_3[k3], and op1_4[k4] data into the ymm7 register to perform vector execution of `vector_sqrt(ymm7)` using the vector instruction `vsqrtpd`. 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*.

```
;; Generated vector function for #pragma omp declare simd notinbranch
processor(core_4th_gen_avx)
_ZGVYN4vvv_7SqrtMulPddi
; parameter 1: ymm0  ; vector_op1 holds op1_1, op1_2, op1_3 and op1_4
; parameter 2: ymm1  ; vector_op2 holds op2_1, op2_2, op2_3 and op2_4
; parameter 3: ymm2  ; vector_k  holds k1, k2, k3, k4
vpmovsxdq    %xmm2, %ymm3           ; save vector_k => ymm3
vpsllq       $3, %ymm3, %ymm4       ; vector_k*8 is index value => ymm4
vpaddq       %ymm0, %ymm4, %ymm5    ; vector_op1 + vector_k*8 => ymm5
vsqrtpd      %ymm1, %ymm0           ; vector_sqrt(ymm1) => ymm0
vpcmpeqd     %ymm6, %ymm6, %ymm6    ; set ymm6 (base) to zero for vgather
                                      instruction
vxorpd       %ymm7, %ymm7, %ymm7    ; clear up ymm7
vgatherqpd   %ymm6, (,%ymm5) %ymm7 ; vector_gather[op1_1[k1], op1_2[k2],
                                      op1_3[k3], op1_4[k4]
                                      => ymm7
vsqrtpd      %ymm7, %ymm1           ; vector_sqrt(ymm7) => ymm1
vmulpd       %ymm1, %ymm0, %ymm0    ; vector multiply: ymm1 * ymm0 => ymm0
ret
```

12 AVX2 code generated for the SIMD function without uniform and linear clauses

The vector variant function generated from the small kernel program (caller and callee functions are implemented in the different files) in **Figure 13** that uses the uniform and linear clauses produces a 1.78x speedup over one that omits the clauses, when it is compiled with the `–QxCore-AVX2` option and `–DSIMDOPT` using the Intel® C/**C++ Compiler** (on an Intel® Xeon processor E3-1270 [Haswell], 64-bit Windows* platform).

```
#include <stdio.h>
#include <stdlib.h>
#define M 1000000
#define N 1024

// File: fSqrtMul.c 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) processor(core_4th_gen_avx)
#ifdef SIMDOPT
#pragam omp declare simd linear(op1) uniform(op2) simdlen(8)
processor(core_4th_gen_avx)
#endif
float fSqrtMul(float *op1, float op2) {
   return sqrt(*op1)*sqrt(op2);
}

// File main.c
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);
```

13 Example using uniform and linear clauses

Understanding Vector Length Selection

If 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. For example, on Intel® architecture, VL selection follows these 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® Advanced Vector Extensions (Intel® AVX) 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.

Memory Access Alignment

Alignment optimization is important for current SIMD hardware.[9] 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 Intel® Core™ i7 processors, 32-byte alignment for Intel AVX and AVX2 processors, and 64-byte alignment for Intel Xeon Phi coprocessors.

```
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];
   }
}
```

14 Example of a SIMD loop using the aligned clause

In the example shown in **Figure 14**, the array *x* is marked as 32-byte aligned and the pointer *y* is marked as 32-byte aligned. The memory reference offset *n1* is asserted with `mod 8 = 0`. These annotations tell the compiler that all vector memory references are 32-byte aligned.

Structure and Multidimensional 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 multidimensional array accesses normally results in uses of nonunit strided-load/store and gather/scatter instruction supported in the SIMD hardware to handle nonunit stride and irregular memory accesses.

A practical way for programmers to achieve effective vector parallelism through explicit vector programming is to convert array of structs (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 innermost dimension; for Fortran array accesses, SIMD should apply to the outermost dimension. The goal is to reshape arrays or to shuffle array access in order to achieve unit stride memory accesses.

BLOG HIGHLIGHTS

Doctor Fortran in "The Future of Fortran"

BY **STEVE LIONEL** >

In November 2014, I led a session at SC14 (the event formerly known as "Supercomputing") titled "The Future of Fortran." I invited representatives from other vendors and members of the Fortran standards committee to participate, and had some accept, but when it came time for the session, I was up there alone. Oh well.

I had prepared a short, vendor-neutral presentation on the current state of the Fortran standard and what was coming in the next standard, currently called Fortran 2015. (I expect this name to stick.) After that I opened it up for questions from the audience, which was more substantial than I expected (about 70–80 people, despite an awful time slot). At the end, I invited attendees to participate in an online survey of Fortran usage, and the results were interesting.

Read more >

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 multiversioning for simple cases, but the code size and compile time increase. The SIMD construct can be used to address these issues. Application programmers can use the SIMD construct to ensure the compiler that the loop can be vectorized.

```
void star( double *a, double *b, double *c, int n, int *ioff)
{  #pragma omp simd
   for ( int i = 0; i < n; i++) a[i] *= b[i] * c[i+ *ioff];
}

subroutine star(a,b,c,n,ioff_ptr)
   implicit none
   double precision  :: a(:),b(:),c(:)
   integer           :: n, i
   integer, pointer  :: ioff_ptr
   !$omp simd
   do i = 1,n
         a(i) = a(i) * b(i) * c(i+ioff_ptr)
   end do
end subroutine
```

15 Example of a SIMD loop written in C++ and Fortran

The example in **Figure 15** (written in both OpenMP C++ and Fortran) shows that *ioff is unknown at compile time, so the compiler has to assume *ioff could be either a negative or positive integer value for each invocation of the 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.

The second example (written in both OpenMP C/C++ and Fortran) in **Figure 16** shows that the SIMD construct can be used for a recursive function. The function fib is called unconditionally in main and also conditionally and recursively in itself. By default, the compiler would create a masked vector version and a nonmasked vector version while retaining the original scalar version.

```
#include <stdio.h>
#include <stdlib.h>
#define N 45
int a[N], b[N], c[N];

#pragma omp declare simd inbranch
int fib( int n )
{ if (n <= 2)
return n;
   else {
         return fib(n-1) + fib(n-2);
   }
}
int main(void)
{
f  or (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”, N-1, a[N-1]);
   return 0;
}

program Fibonacci implicit
   none integer,parameter  :: N=45
   integer                 :: a(0:N-1), b(0:N-1)
   integer, external       :: fib
   do i = 0,N-1
         b(i) = i
   end do
```

16 A SIMD program using declare SIMD and SIMD construct

```
    !$omp
    simd do
    i=0,N-1
          a(i) = fib(b(i))
    end do
    write(*,*) "Done a(", N-1, ") = ", a(N-1) ! 44 1134903168 end program

 recursive function fib(n) result(r)
 !$omp declare simd(fib) inbranch
    implicit none
    integer     :: n, r if (n <= 2) then
          r = n
    else
          r = fib(n-1) + fib(n-2)
    endif
 end function fib
```

16 Continued

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. Another possibility is that the 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.

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 topic of parallelization is beyond the scope of this article, 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 17 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.

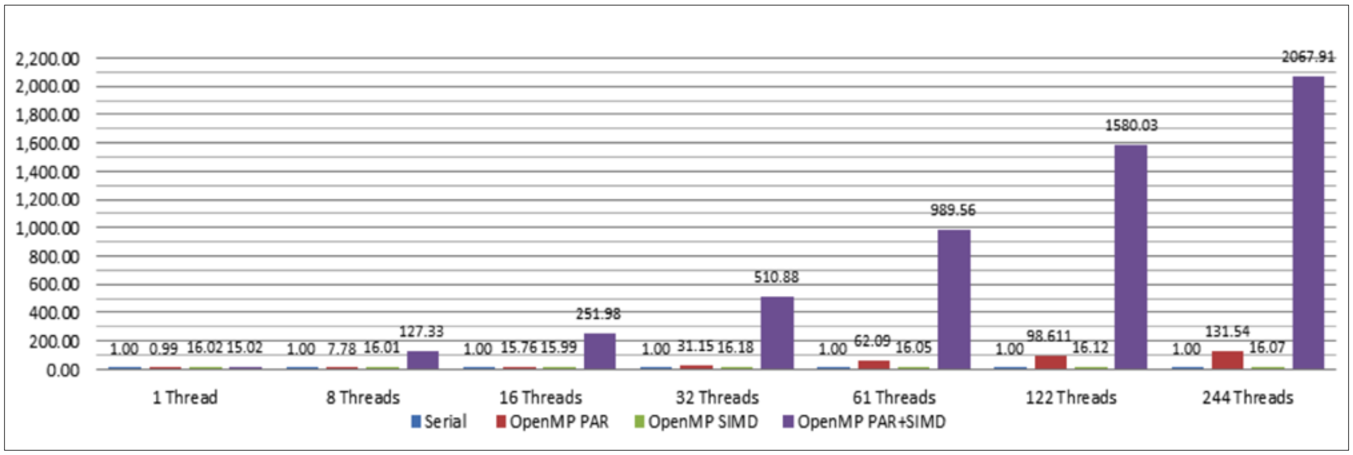
```
#pragma omp declare simd uniform(max_iter) simdlen(32)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{
   // Computes number of iterations(count variable)
   // that it takes for parameter c to be known to
   // be outside mandelbrot set uint32_t count = 1; fcomplex z = c;

while ((cabsf(z) < 2.0f) && (count < max_iter)) {
         z = z * z + c;
         count++;
   }
   return count;
}
```

```
Caller site code:
int main() {
… … …
   #pragma omp parallel for schedule(guided)
   for (int32_t y = 0; y < ImageHeight; ++y) {
         float c_im = max_imag - y * imag_factor;
         #pragma omp simd safelen(32)
         for (int32_t x = 0; x < ImageWidth; ++x) {
         fcomplex in_val;
         in_val = (min_real + x*real_factor) + (c_im*1.0iF);
         count[y][x] = mandel(in_val, max_iter);
         }
   }
   … … …
}
```

17 Example of OpenMP parallel for and SIMD constructs usage

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 doubly 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 17**. The guided scheduling type is used to balance the load across threads since each call to mandel performs a varying amount of work in terms of execution time due to the exit of the while loop. The performance measurement is conducted on an Intel Xeon Phi coprocessor configured with 61 cores (1GHz, 32KB L1, 512 KB L2, 4 hyperthreads per core), 300W power budget, 7936 MB, 16 memory channels, 2.75 GHz memory speed, 352GB/s memory bandwidth, 512-bit SIMD vector length, and 32-bit memory data width.



18 Speedup of Mandelbrot workload of OpenMP* parallel for and SIMD usage

Figure 18 shows that the SIMD vectorization alone `(options –mmic –openmp –std=c99 –O3)` delivers a speedup of approximately 16x over the serial execution. The OpenMP parallelization delivers a 62.09x speedup with **hyperthreading** off (61 threads using 61 cores) and a 131.54x speedup with hyperthreading on (244 threads using 61 cores and 4 hyperthreading 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 to 61 threads is close to linear. In addition, hyperthreading support delivers a performance gain of approximately 2x by comparing the 244-thread speedup with the 61-thread speedup, which is better than the well-known 20 to 30 percent expected performance gain from hyper-threading technology, since the workload has little computing resource contention and the 4 hyperthreading threads hide latency well.

Performance Measurement

Our performance measurement is conducted with a number of workloads to validate the effectiveness of using SIMD extensions in OpenMP C/C++ language and the compiler support in Intel C/C++ 16.0 beta compilers. The SIMD codes produced by the compiler are highly optimized with architecture-specific tunings, and advanced scalar, memory, and loop optimizations assisted with aggressive memory disambiguation using `–O3 –Qopenmp-simd –QxSSE4.2/-QxCore-AVX2` options. The performance measurement is carried out on an Intel Xeon processor E3-1270 (Haswell) system (4 cores with hyperthreading on), running at 3.50GHz, with a 32-GB RAM, 8M L3 cache, 64-bit Windows Server* 2012 R2. In this article, we present both SSE4.2 and AVX2 performance results.

Workloads

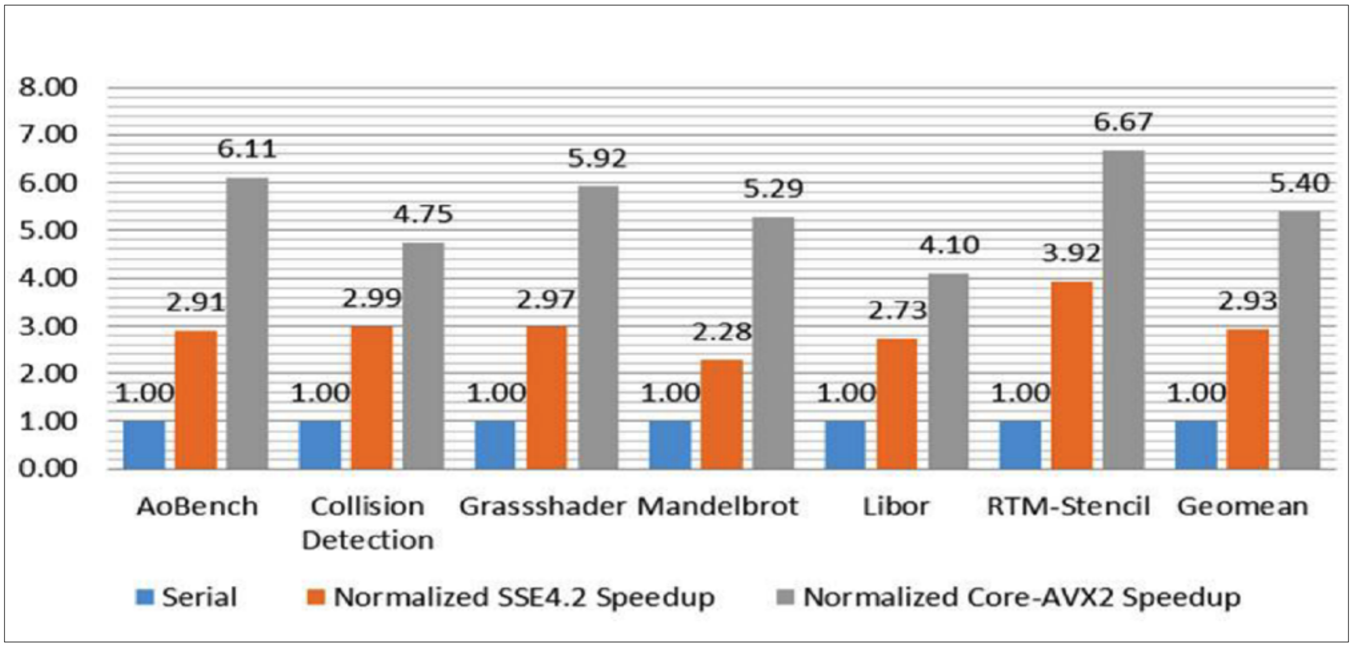
Six workloads are chosen from different application domains to measure the performance of the SIMD vector extensions and compiler implementation. These workloads cover visual computing, graphic simulation, abstract mathematic computation, finance computation, and dynamic system simulation such as AOBench, which is used for rendering ambient occlusion, a shading method that creates nonreflective surfaces to add realism to local reflection models used in 3-D graphics. **Table 1** provides some information on these workloads, including the SIMD vector extension used, input data size, number of lines of code, control flow characteristics, language employed, and a brief description of each workload.

Benchmarks	Benchmark Characteristics					
	Vector Extension Used	Data Size	Lines of Code (LOC)	Control Flow	Language	Description
AOBench	**pragma omp simd** reduction; **pragma omp declare simd** uniform simdlem	512x512	667	Multiple branches in vector functions	C++	Graphics: (ambient occlusion) shader computes attenuation due to occlusion.
Collision Detection	**pragma omp simd; pragma omp declare simd** uniform linear simdlen	3125x32	330	Single branch in vector function	C++	Graphics: updates a scene of moving spheres to correct motion based on collisions at fixes time steps
Grassshader	**pragma omp simd** reduction; **pragma omp declare simd** uniform linear	1000x4x4	862	Multiple branches in vector functions	C++	Graphics: simulate an idyllic nature scene (grass, trees, bushes, etc.) in a realistic fashion
Mandelbrot	**pragma omp simd; pragma omp declare simd** uniform linear simdlen	1000x1000	71	for loop in vector function with early exit	C	Abstract mathematics: computation of complex quadratic recurrence equation
Libor	**pragma omp simd** uniform vectorlength	80x40x15	400	for loop inside vector function	C	Compute Swaption portfolio
RTM Stencil	**pragma omp simd**	400x400x400 10 times	210	Simple nested for loops	C	Seismic: reverse time migration, 3-D stencil over time steps (FDTD)

Table 1. Workload (Benchmark) Information

Performance Results

The performance improvement is measured by the serial execution and SIMD vector execution on the Intel® system described at the beginning of this section. **Figure 19** presents the normalized SIMD speedups of seven chosen workloads. The generated SIMD codes of these workloads achieved speedup ranges from 2.28x to 6.67x with a geomean speedup of 2.93x (using SSE4.2) and 5.40x (using AVX2).



19 SIMD speedup of several workloads with SSE4.2 and AVX2

Conclusion

Driven by the increasing prevalence of SIMD architectures in modern CPU and GPU processors,[5, 6, 7] the OpenMP 4.0 (and 4.1 draft version) leveraged Intel's explicit SIMD extensions[11] to provide an industry-standard set of high-level SIMD vector extensions.[8] These extensions 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 using SIMD intrinsic function or inline ASM code. With these SIMD extensions, compiler vendors can now deliver excellent performance on modern CPU and GPU processors. For GPU accelerators, the OpenMP SIMD extension works seamlessly for Intel GPUs with the 1024-bit vector length support in conjunction of the Intel offloading feature and OpenMP target construct support. Details and performance results for offloading and GPUs support will be presented in another article.

- Intel® C/**C++** and **Fortran compilers** are enhanced to vectorize ordinary C/C++ and Fortran functions for accelerating multimedia, graphics, visual computing, pixel, and embedded applications on SIMD vector architectures, with a seamless integration to threading models beyond traditional loop/loop-nest only SIMD **vectorization**.
- The programmer is presented with general purpose language extensions to express SIMD vector parallelism in C/C++ and Fortran. Intel® compilers provide a productive programming environment to produce SIMD code for the modern SIMD hardware that reduces the burden of directly coding in SIMD intrinsic functions or inline ASM code, which is a lower-productivity activity, whenever the automatic vectorization fails.

Acknowledgments

The authors would like to thank Jennifer Yu, Michael Rice, Clark Nelson, Peter Karam, Shin Lee, and Stan Whitlock for their contributions of extending C/**C++** and **Fortran compiler** front ends to support SIMD extensions to OpenMP C/C++ and Fortran. In particular, we thank all members of the U.S./Novo Vectorizer team, HPO team, ScalarOpt, and PCG team for their contributions in developing high-performance optimizers and code generator for Intel Xeon and Xeon Phi architectures. We also appreciate the management support from Alice S. Chan, Kevin J. Smith, and Geoff Lowney. Thanks also go to all members of Intel compiler groups for delivering Intel® C/C++ and Fortran high-performance compilers.

References

1. A. Bik, M. Girkar, P. M. Grey, and X. Tian. "Automatic Intra-Register Vectorization for the Intel Architecture." *International Journal of Parallel Programming*, (2): pp. 65-98, April 2002.

2. A. Eichenberger, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the CELL Processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

3. D. Nuzman, I. Rosen, and A. Zaks. Auto-Vectorization of Interleaved Data for SIMD. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

4. G. Ren, P. Wu, and D. Padua. A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions. 16th International Workshop of Languages and Compilers for Parallel Computing, October 2003.

5. I. Buck, T. Foley, D. Horn, J. Superman, K. Patahalian, M. Hourston, and P. Hanrahan. *Brook for GPUs: Stream Computing on Graphics Hardware, ACM Transactions on Graphics*, 23(3): pp. 777-786, 2004.

6. Intel Corporation. "Intel® Xeon Phi™ Coprocessor System Software Developers Guide," November 2012, **http://software.intel.com/en-us/mic-developer**.

7. Intel Corporation. Intel® Advanced Vector Extensions Programming Reference, Document Number 319433-011, June 2011.

8. OpenMP Architecture Review Board. "OpenMP Application Program Interface," Version 4.0, July 2013, and 4.1 (draft version), June 2015, **http://www.openmp.org**.

9. P. Wu, A. E. Eichenberger, and A. Wang. Efficient SIMD Code Generation for Runtime Alignment. In *Proceedings of the Symposium on Code Generation and Optimization*, 2005.

10. S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp.145-156, June 2000.

11. X. Tian, H. Saito, M. Girkar, S. Preis, S. Kozhukhov, A.G. Cherkasov, C. Nelson, N. Panchenko, and R. Geva. Compiling C/C++ SIMD Extensions for Function and Loop Vectorization on Multicore-SIMD Processors. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium – Multicore and GPU Programming Models, Language, and Compilers Workshop*, pp. 2349–2358, 2012.

12. X. Tian, H. Saito, S. Kozhkhov, K.B. Smith, R. Geva, M. Girkar, and S. Preis, Vector Function Application Binary Interface, Version. 0.9.5, January 2013, **https://www.cilkplus.org/sites/default/files/open_specifications/Intel-ABI-Vector-Function-2012-v0.9.5.pdf**.