HLRN Parallel Programming Workshop
Speedup your Code on Intel® Processors at HLRN

October 19th, 2017

OpenMP 4 SIMD Programming

Florian Wende
Motivation

SIMD vectorization is essential to achieve TFLOPS performance on today’s and tomorrow’s CPUs

Amdahl’s law for SIMD
- AVX-512: approx. 80% of your code must vectorize to get 50% peak

**Intel Xeon Phi 7290 (KNL):** 72 CPU cores, **AVX-512**
- \[\text{peak}_{\text{SIMD,32-bit}} = 5990 \text{ GFLOPs} \ (\approx 72 \cdot 2 \cdot 16 \cdot 2 \cdot 1.3)\]
- \[\text{peak}_{\text{no-SIMD,32-bit}} = 375 \text{ GFLOPs}\]
Motivation

How to approach SIMD in your program?

<table>
<thead>
<tr>
<th>Method</th>
<th>Code Invasiveness</th>
<th>Required Skills</th>
<th>Prospect of Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-vectorization</td>
<td>none</td>
<td>none</td>
<td>low</td>
</tr>
<tr>
<td>Explicit vectorization</td>
<td>low</td>
<td>medium</td>
<td>depends</td>
</tr>
<tr>
<td>Manual vectorization</td>
<td>high</td>
<td>expert</td>
<td>high</td>
</tr>
</tbody>
</table>
Motivation

How to approach SIMD in your program?

- Auto-vectorization
  - Code invasiveness: low
  - Required skills: medium
  - Prospect of success: depends

- Explicit vectorization
  - Code invasiveness: low
  - Required skills: medium
  - Prospect of success: depends

- Manual vectorization
  - Code invasiveness: low
  - Required skills: medium
  - Prospect of success: depends
Explicit SIMD vectorization with OpenMP 4

OpenMP as the means for multi-threading (at least in HPC) and...

...SIMD vectorization as of version 4.0

<table>
<thead>
<tr>
<th></th>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop vectorization:</td>
<td>#pragma omp simd</td>
<td>!$omp simd</td>
</tr>
<tr>
<td>function vectorization:</td>
<td>#pragma omp declare simd</td>
<td>!$omp declare simd (func)</td>
</tr>
</tbody>
</table>
Explicit SIMD vectorization with OpenMP 4

OpenMP as the means for multi-threading (at least in HPC) and...

...SIMD vectorization as of version 4.0

- quasi-standard for compiler directive based vectorization
- supported by major compiler vendors
  - GNU, Clang, Intel, Cray, ...
- need to adapt the code base just once to affect different platforms
  - standard CPU, many-core (Intel Xeon Phi), GPGPU, ...
Explicit SIMD vectorization with OpenMP 4

Loop vectorization

for (int i = 0; i < n; ++i)
    y[i] = 2.0 * x[i];

do i = 1, n
    y(i) = 2.0 * x(i)
endo
Explicit SIMD vectorization with OpenMP 4

Loop vectorization

```c
#pragma omp simd
for (int i = 0; i < n; ++i)
    y[i] = 2.0 * x[i];
```

```c
!$omp simd
do i = 1, n
    y(i) = 2.0 * x(i)
endo
```
Explicit SIMD vectorization with OpenMP 4

Loop vectorization

```c
#pragma omp simd aligned(x, y)
for (int i = 0; i < n; ++i)
    y[i] = 2.0 * x[i];
```

```c
!$omp simd aligned(x, y)
do i = 1, n
    y(i) = 2.0 * x(i)
enddo
```

x and y point to aligned memory
Explicit SIMD vectorization with OpenMP 4

Loop vectorization

```c
#pragma omp simd safelen(3)
for (int i = 3; i < n; ++i)
    y[i] = 2.0 * x[i - 3];
```

```c
!$omp simd safelen(3)
do i = 4, n
    y(i) = 2.0 * x(i - 3)
enddo
```

SIMD vectorization is safe only within a window of 3 successive loop iterations (data dependences)
Explicit SIMD vectorization with OpenMP 4

Loop vectorization

```c
#pragma omp simd simdlen(32)
for (int i = 0; i < n; ++i)
    y[i] = 2.0 * x[i];
```

operate 32 successive loop iterations in SIMD fashion
(kind of dealing with logical SIMD vectors of size larger than what the hardware supports natively)
Explicit SIMD vectorization with OpenMP 4

Loop vectorization

```c
#pragma omp simd reduction(+:sum)
for (int i = 0; i < n; ++i)
    sum += 2.0 * x[i];
```

```c
!$omp simd reduction(+:sum)
do i = 1, n
    sum = sum + 2.0 * x(i)
enddo
```

reduce on sum
Explicit SIMD vectorization with OpenMP 4

Loop vectorization

```c
#pragma omp simd collapse(2)
for (int j = 0; j < m; ++j)
    for (int i = 0; i < n; ++i)
        y[j][i] = x[i] * x[j];
```

```c
!$omp simd collapse(2)
do j = 1, m
do i = 1, n
    y(i, j) = x(i) * x(j)
enddo
enddo
```

collapse the two loops into a single loop with trip count \( m \times n \):
might be beneficial in case of small \( n \) and/or \( m \) values
Explicit SIMD vectorization with OpenMP 4

Loop vectorization

double c;
#pragma omp simd private(c)
for (int i = 0; i < n; ++i) {
    c = log(x[i]);
    y[i] = c * x[i];
}

variable c is private per SIMD lane

real(8) :: c
!$omp simd private(c)
do i = 1, n
    c = log(x(i))
    y(i) = c * x(i)
enddo
Explicit SIMD vectorization with OpenMP 4

Loop vectorization

double c;
#pragma omp simd lastprivate(c)
for (int i = 0; i < n; ++i) {
    c = log(x[i]);
    y[i] = c * x[i];
}

real(8) :: c
!$omp simd lastprivate(c)
do i = 1, n
    c = log(x(i))
    y(i) = c * x(i)
enddo

variable c is private per SIMD lane and its value after the SIMD loop execution is equal to that after executing the loop in scalar fashion
Explicit SIMD vectorization with OpenMP 4

Function vectorization

```c
#pragma omp declare simd
double foo(const double x) {
    return log(x);
}

#pragma omp simd
for (int i = 0; i < n; ++i)
    y[i] = foo(x[i]);
```

```c
function foo(x)
    !$omp declare simd (foo)
    real(8), intent(in) :: x
    real(8) :: foo
    foo = log(x)
end function foo

do i = 1, n
    y(i) = foo(x(i))
enddo
```
Explicit SIMD vectorization with OpenMP 4

Function vectorization

```c
#pragma omp declare simd
double foo(const double x) {
  return log(x);
}

#pragma omp simd
double x[1000]
for (int i = 0; i < n; ++i)
  y[i] = foo(x[i]);
```

you can add clauses similar to loop vectorization:
aligned, simdlen, uniform, linear, ...

```c
!$omp declare simd (foo)
real(8), intent(in) :: x
real(8) :: foo
global foo = log(x)
end function foo

!$omp simd
do i = 1, n
  y(i) = foo(x(i))
enddo
```
Explicit SIMD vectorization with OpenMP 4

Function vectorization

```c
#pragma omp declare simd uniform(c)
double foo(const double x, const double c) {
    return c + log(x);
}

#pragma omp simd
for (int i = 0; i < n; ++i)
    y[i] = foo(x[i], c);

broadcast c across all SIMD lanes
```

```c
function foo(x, c)
$omp declare simd (foo) uniform(c)
  real(8), intent(in) :: x, c
  real(8) :: foo
  foo = c + log(x)
end function foo

!$omp simd
do i = 1, n
    y(i) = foo(x(i), c)
endo
```
Explicit SIMD vectorization with OpenMP 4

Function vectorization

```c
#pragma omp declare simd uniform(x) linear(i:1)
double foo(const double* x, const int i) {
    return x[i];
}

#pragma omp simd
for (int i = 0; i < n; ++i)
    y[i] = foo(x, i);
```

variable $i$ changes from one loop iteration to the other
by 1 in the scalar case, and by $VL \times 1$ in the SIMD case
Explicit SIMD vectorization with OpenMP 4

Function vectorization

subroutine foo(x, y)
!$omp declare simd (foo) linear(ref(x, y))
    real(8), intent(in) :: x
    real(8), intent(out) :: y
    y = log(x)
end subroutine foo

!$omp simd
do i = 1, n
    call foo(x(i), y(i))
enddo

Fortran gives arguments to functions/subroutines by reference per-default

In the SIMD case a vector of reference will be created, resulting in gather/scatter operations

Use linear(ref(..)) to tell the compiler that references in the vector point to successive positions in main memory!
Explicit SIMD vectorization with OpenMP 4

Function vectorization

```fortran
subroutine foo(x, y)
!$omp declare simd (foo) linear(ref(x, y))
    real(8), intent(in) :: x
    real(8), intent(out) :: y
    y = log(x)
end subroutine foo

!$omp simd
do i = 1, n
    call foo(x(i), y(i))
endo
```

Full documentation of all the OpenMP 4 SIMD clauses and features:
www.openmp.org/mp-documents
OpenMP 4 SIMD – libmvec

GNU: SIMD math functions are available with glibc-2.21 and later through libmvec

If your glibc is older, update it

1.) build glibc: configure --prefix=<glibc_path> && make && make install
2.) export glibc install path (bash shell): export GLIBCROOT=<glibc_path>
3.) compile your program with --ftree-vectorize --ffast-math -I$GLIBCROOT/include
4.) link your program with -Wl,-rpath,$GLIBCROOT/lib -Wl,-dynamic-linker,
   $GLIBCROOT/lib/1d-linux-x86-64.so.2 -L$GLIBCROOT/lib -lm

→ you can check vectorization successes via -fopt-info-vec-optimized compile flag
GNU: you have libmvec available, but the following loop is not vectorized

```fortran
!$omp simd
do i = 1, n
  y(i) = log(x(i))
enddo
```

The GNU compiler seems not to use libmvec’s SIMD log version
GNU: you have libmvec available, but the following loop is not vectorized

```fortran
!$omp simd
do i = 1, n
   y(i) = log(x(i))
enddo
```

The GNU compiler seems not to use libmvec’s SIMD log version

Anyway, we tell the compiler that it exists somewhere!
GNU: you have libmvec available, but the following loop is not vectorized

```
use simd
!$omp simd
do i = 1, n
  y(i) = simd_log(x(i))
enddo
```

```
module simd
  interface
    function simd_log(x) bind(c, name="__log_finite")
      real(c_double) :: x
      real(c_double), value, intent(in) :: x
    end function simd_log
  end interface
end module simd
```

```
use simd
!$omp simd
do i = 1, n
  y(i) = simd_log(x(i))
enddo
```
GNU: you have libmvec available, but the following loop is not vectorized

```
use simd
!
!$omp simd
do i = 1, n
  y(i) = simd_log(x(i))
enddo

module simd
  interface
    function simd_log(x) bind(c, name="__log_finite") !$omp declare simd (simd_log)
      use iso_c_binding
      real(c_double) :: simd_log
      real(c_double), value, intent(in) :: x
    end function simd_log
  end interface
end module simd

__log_finite is libmvec’s logarithm function
!$omp declare simd.. creates SIMD version _ZGVdN4v__log_finite when building for AVX2
```
GNU: you have libmvec available, but the following loop is not vectorized

```fortran
use simd
!

module simd
    interface
        function simd_log(x) bind(c, name="__log_finite")
    !$omp declare simd (simd_log)
        use iso_c_binding
        real(c_double) :: simd_log
        real(c_double), value, intent(in) :: x
    end function simd_log
    end interface
end module simd
```

```fortran
use simd
!

do i = 1, n
    y(i) = simd_log(x(i))
enddo
```

approx. 7.5x performance
gain over scalar log version
on AVX2 machine
OpenMP 4 SIMD – Missing math intrinsics

GNU/Cray: Intel-specific SIMD intrinsics for SVML calls are not available

Build them yourself!

```c
#if !defined(__INTEL_COMPILER)
__m512d __mm512_log_pd(const __m512d& x) {
    double tmp[8] __attribute__((aligned(64)));
    __mm512_store_pd(tmp, x);
    #pragma omp simd
    for (int i = 0; i < 8; ++i)
        tmp[i] = log(tmp[i]);
    return __mm512_load_pd(tmp);
}
#endif
```