SIMD Vectorization with OpenMP

Dr.-Ing. Michael Klemm
Software and Services Group
michael.klemm@intel.com
Credits

“The Tutorial Gang“

Christian Terboven
Michael Klemm
Ruud van der Pas
Eric Stotzer
Bronis R. de Supinski

Members of the OpenMP Language Committee
Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Evolution of Hardware (Intel)

<table>
<thead>
<tr>
<th>64-bit Intel® Xeon® processor</th>
<th>Intel® Xeon® processor 5100 series</th>
<th>Intel® Xeon® processor 5500 series</th>
<th>Intel® Xeon® processor 5600 series</th>
<th>Intel® Xeon® processor E5-2600v2 series</th>
<th>Intel® Xeon Phi™ Co-processor 7120P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>3.6 GHz</td>
<td>3.0 GHz</td>
<td>3.2 GHz</td>
<td>3.3 GHz</td>
<td>2.7 GHz</td>
</tr>
<tr>
<td>Core(s)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Thread(s)</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>SIMD width</td>
<td>128 (2 clock)</td>
<td>128 (1 clock)</td>
<td>128 (1 clock)</td>
<td>128 (1 clock)</td>
<td>256 (1 clock)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>512 (1 clock)</td>
</tr>
</tbody>
</table>

Images not intended to reflect actual die sizes
## Levels of Parallelism

OpenMP already supports several levels of parallelism in today’s hardware

<table>
<thead>
<tr>
<th>Levels</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster</td>
<td>Group of computers communicating through fast interconnect</td>
</tr>
<tr>
<td>Coprocessors/Accelerators</td>
<td>Special compute devices attached to the local node through special interconnect</td>
</tr>
<tr>
<td>Node</td>
<td>Group of processors communicating through shared memory</td>
</tr>
<tr>
<td>Socket</td>
<td>Group of cores communicating through shared cache</td>
</tr>
<tr>
<td>Core</td>
<td>Group of functional units communicating through registers</td>
</tr>
<tr>
<td>Hyper-Threads</td>
<td>Group of thread contexts sharing functional units</td>
</tr>
<tr>
<td>Superscalar</td>
<td>Group of instructions sharing functional units</td>
</tr>
<tr>
<td>Pipeline</td>
<td>Sequence of instructions sharing functional units</td>
</tr>
<tr>
<td>Vector</td>
<td>Single instruction using multiple functional units</td>
</tr>
</tbody>
</table>
SIMD on Intel® Architecture

- Width of SIMD registers has been growing:

  - **SSE**: 128 bit
    - 2 x DP
    - 4 x SP

  - **AVX**: 256 bit
    - 4 x DP
    - 8 x SP

  - **MIC**, **AVX-512**: 512 bit
    - 8 x DP
    - 16 x SP
More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor

vaddpd dest, source1, source2

\[ \begin{array}{cccccccc}
a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
\hline
b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
\hline
\end{array} \]

\[ \begin{array}{cccccccc}
a_7 + b_7 & a_6 + b_6 & a_5 + b_5 & a_4 + b_4 & a_3 + b_3 & a_2 + b_2 & a_1 + b_1 & a_0 + b_0 \\
\hline
\end{array} \]

512 bit
More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor

\[ \text{vfmad}d213pd \text{ source1, source2, source3} \]

\[
\begin{array}{cccccccc}
  a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\
  b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
  c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\
\end{array}
\]

\[
\text{vfmad}d213pd \text{ source1, source2, source3} =
\]

\[
\begin{array}{ccccccccc}
  \text{source1} & \text{source2} & \text{source3} & \text{dest} \\
  a_7*b_7 + c_7 & a_6*b_6 + c_6 & a_5*b_5 + c_5 & a_4 *b_4 + c_4 & a_3*b_3 + c_3 & a_2*b_2 + c_2 & a_1*b_1 + c_1 & a_0*b_0 + c_0 \\
\end{array}
\]
More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor

vaddpd dest{k1}, source2, source3

```
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
| a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  |
```

```
= a7+b7 d6 a5+b5 d4 d3 a2+b2 d1 a0+b0
```

source1
source2
mask
dest
More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor

SIMD Vectorization with OpenMP
Auto-vectorization

Auto vectorization only helps in some cases

→ Increased complexity of instructions makes it hard for the compiler to select proper instructions
→ Code pattern needs to be recognized by the compiler
→ Precision requirements often inhibit SIMD code gen

Example: Intel® Composer XE

-vec (automatically enabled with -03)
-vec-report
-opt-report
Why Auto-vectorizers Fail

- Data dependencies
- Other potential reasons
  - Alignment
  - Function calls in loop block
  - Complex control flow / conditional branches
  - Loop not “countable”
    - e.g., upper bound not a runtime constant
  - Mixed data types
  - Non-unit stride between elements
  - Loop body too complex (register pressure)
  - Vectorization seems inefficient
- Many more … but less likely to occur
Data Dependencies

- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must execute before S2
  → Control-flow dependence
  → Data dependence
  → Dependencies can be carried across loop iterations

- Important flavors of data dependencies

FLOW

\[ \begin{align*}
  s1: & \ a = 40 \\
  & \ b = 21 \\
  s2: & \ c = a + 2
\end{align*} \]

ANTI

\[ \begin{align*}
  & \ b = 40 \\
  s1: & \ a = b + 1 \\
  s2: & \ b = 21
\end{align*} \]
Loop-Carried Dependencies

- Dependencies may occur across loop iterations
  → Loop-carried dependency

- The following code contains such a dependency:

```c
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

- Some iterations of the loop have to complete before the next iteration can run
  → Simple trick: can you reverse the loop w/o getting wrong results?
Can we parallelize or vectorize the loop?

→ Parallelization: no
   (except for very specific loop schedules)

→ Vectorization: yes
   (if vector length is shorter than any distance of any dependency)
Example: Loop not Countable

“Loop not Countable” plus “Assumed Dependencies”

typedef struct {
    float* data;
    size_t size;
} vec_t;

void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
In a Time Before OpenMP 4.0

- Support required vendor-specific extensions
  - Programming models (e.g., Intel® Cilk Plus)
  - Compiler pragmas (e.g., `#pragma vector`)
  - Low-level constructs (e.g., `_mm_add_pd()`)
SIMD Loop Construct

- Vectorize a loop nest
  → Cut loop into chunks that fit a SIMD vector register
  → No parallelization of the loop body

- Syntax (C/C++)
  ```
  #pragma omp simd [clause[[], clause],...]
  for-loops
  ```

- Syntax (Fortran)
  ```
  !$omp simd [clause[[], clause],...]
  do-loops
  [$!$omp end simd]
  ```
Example

void sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp simd reduction(+:sum)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}
**Data Sharing Clauses**

- **private(var-list):**
  Uninitialized vectors for variables in var-list

- **reduction(op:var-list):**
  Create private variables for var-list and apply reduction operator op at the end of the construct
**SIMD Loop Clauses**

- **safelen** *(length)*
  
  - Maximum number of iterations that can run concurrently without breaking a dependence
  
  - In practice, maximum vector length

- **simdlen** *(length)*
  
  - Specify preferred length of SIMD registers used
  
  - Must be less or equal to **safelen** if also present

- **linear** *(list[:linear-step]*)
  
  - The variable’s value is in relationship with the iteration number
  
  - \( x_i = x_{\text{orig}} + i \times \text{linear-step} \)

- **aligned** *(list[:alignment]*)
  
  - Specifies that the list items have a given alignment
  
  - Default is alignment for the architecture

- **collapse** *(n)*
SIMD Worksharing Construct

- Parallelize and vectorize a loop nest
  - Distribute a loop’s iteration space across a thread team
  - Subdivide loop chunks to fit a SIMD vector register

- Syntax (C/C++)
  #pragma omp for simd [clause[,[,] clause],...] for-loops

- Syntax (Fortran)
  !$omp do simd [clause[,[,] clause],...] do-loops
  [$!omp end do simd [nowait]]
void sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp for simd reduction(+:sum)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}
Be Careful What You Wish For...

```c
void sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp for simd reduction(+:sum) \schedule(static, 5)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}
```

- You should choose chunk sizes that are multiples of the SIMD length
  - Remainder loops are not triggered
  - Likely better performance
- In the above example …
  - and AVX2, the code will only execute the remainder loop!
  - and SSE, the code will have one iteration in the SIMD loop plus one in the remainder loop!
The new `simd` modifier automatically adjusts the chunk size to match it with the length of the SIMD register.

- New chunk size becomes \([\text{chunksz}/\text{simdlen}] \times \text{simdlen}\)
- AVX2: new chunk size will be 8
- SSE: new chunk size will be 8
SIMD Function Vectorization

```c
float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
    #pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}
```
SIMD Function Vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

- Syntax (C/C++):
  
  ```
  #pragma omp declare simd [clause[,[,] clause],...]
  [#pragma omp declare simd [clause[,[,] clause],...]]
  [...]
  function-definition-or-declaration
  ```

- Syntax (Fortran):
  
  ```
  !$omp declare simd (proc-name-list)
  ```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
    #pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}

vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}

vec8 distsq_v(vec8 x, vec8 y) {
    return (x - y) * (x - y);
}

vd = min_v(distsq_v(va, vb), vc)
SIMD Function Vectorization

- **simdlen**(length)
  - generate function to support a given vector length

- **uniform**(argument-list)
  - argument has a constant value between the iterations of a given loop

- **inbranch**
  - function always called from inside an if statement

- **notininbranch**
  - function never called from inside an if statement

- **linear**(argument-list[:linear-step])

- **aligned**(argument-list[:alignment])

- **reduction**(operator:list)

Same as before
inbranch & notinbranch

```c
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
    #pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}

vec8 do_stuff_v(vec8 x, mask m) {
    /* do something */
    vmulpd x{m}, 2.0, tmp
    return tmp;
}

for (int i = 0; i < N; i+=8) {
    vcmp_lt &a[i], 0.0, mask
    b[i] = do_stuff_v(&a[i], mask);
}
```