VECTORIZATION INSIGHTS WITH THE INTEL® COMPILER FOR IMPROVED PERFORMANCE ON THE INTEL® XEON PHI™ X200 PROCESSOR FAMILY
Introduction to Intel® Advanced Vector Instructions 512 (Intel AVX-512)
Compress and expand loops
Histogram and scatter loops
“Gather to Shuffle”: Converting loops to unit stride
The days of easy performance improvements from steadily increasing clock speed are long gone
- Moore’s law results instead in increased parallelism

Intel® Xeon Phi™ x200 processors provide
- Many cores
- More and wider SIMD registers
- Powerful new instructions
- High bandwidth memory
- ... 

Valuable only if application software makes use of them!
INTEL® ADVANCED VECTOR EXTENSIONS 512 (INTEL® AVX-512) SIMD DATA TYPES FOR THE INTEL® XEON PHI™ X200 PROCESSOR FAMILY

- 16x floats
- 8x doubles
- 16x 32-bit integers
- 8x 64-bit integers
**SIMD: SINGLE INSTRUCTION, MULTIPLE DATA**

double *x, *y, *z;  
for (i=0; i<n; i++)  
z[i] = x[i] + y[i];

- **Scalar mode**  
  - one instruction produces one result  
  - E.g. `vaddsd`, `(vaddss)`

- **Vector (SIMD) mode**  
  - one instruction can produce multiple results  
  - E.g. `vaddpd`, `(vaddps)`

---

Scalar mode example:

\[ x + y = x + y \]

Vector (SIMD) mode example:

\[ \begin{align*}  
x7 + y7 &= x7 + y7 
x6 + y6 &= x6 + y6 
x5 + y5 &= x5 + y5 
x4 + y4 &= x4 + y4 
x3 + y3 &= x3 + y3 
x2 + y2 &= x2 + y2 
x1 + y1 &= x1 + y1 
x0 + y0 &= x0 + y0 
\end{align*} \]
### INTEL® COMPILER SWITCHES TARGETING INTEL® AVX-512

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-xmic-avx512</code></td>
<td>Intel® Xeon Phi™ x200 processor family only</td>
</tr>
<tr>
<td><code>-xcore-avx512</code></td>
<td>Future Intel® Xeon® processor only</td>
</tr>
<tr>
<td><code>-xcommon-avx512</code></td>
<td>Intel® AVX-512 subset common to both. Not a fat binary.</td>
</tr>
<tr>
<td><code>-axmic-avx512   etc.</code></td>
<td>Fat binaries. Targets Intel® Xeon Phi™ x200 and also other Intel® Xeon® processors</td>
</tr>
<tr>
<td><code>-qoffload-arch=mic-avx512</code></td>
<td>Offload to an Intel® Xeon Phi™ x200 coprocessor</td>
</tr>
</tbody>
</table>

Binaries built for earlier Intel® Xeon® processors will run unchanged on the Intel® Xeon Phi™ x200 processor family.

Binaries built for the Intel® Xeon Phi™ x100 coprocessor family will not.
COMPRESS/EXPAND LOOPS VECTORIZE WITH INTEL® AVX-512

- With Intel® AVX2, does not auto-vectorize
  - and vectorizing with an OpenMP* SIMD directive would be unsafe
  - ifort -c -xcore-avx2 -qopt-report-file=stderr -qopt-report=3 -qopt-report-phase=vec compress.f90
    ... LOOP BEGIN at compress.f90(23,3)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization. ...
      remark #15346: vector dependence: assumed ANTI dependence between nb (25:7) and nb (25:7)
    LOOP END
- C code behaves the same

```
nb = 0
do ia=1, na           ! line 23
  if(a(ia) > 0.) then
    nb = nb + 1       ! dependency
    b(nb) = a(ia)     ! compress
  endif
endo
```

```
for (int i; i <N; i++) {
  if (a[i] > 0) {
    b[j++] = a[i];     // compress
    c[i] = a[k++];     // expand
  }
}
// Cross-iteration dependencies via j and k
```
COMPRESS LOOP

- Compile for Intel® AVX-512:
  ```
  ifort -c -qopt-report-file=stderr -qopt-report=3 -qopt-report-phase=vec -xmic-avx512 compress.f90
  ...
  LOOP BEGIN at compress.f90(23,3)
  remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 1
  remark #15457: masked unaligned unit stride stores: 1
  ...
  remark #15478: estimated potential speedup: 14.040
  remark #15497: **vector compress**: 1
  LOOP END
  ```

- Compile with --S to see new instructions in assembly code:
  ```
  grep vcompress compress.s
  vcompressps %zmm4, -4(%rsi,%rdx,4){%k1} #14.7 c7 stall 1
  vcompressps %zmm1, -4(%rsi,%r12,4){%k1} #14.7 c5
  vcompressps %zmm1, -4(%rsi,%r12,4){%k1} #14.7 c5
  vcompressps %zmm4, -4(%rsi,%rdi,4){%k1} #14.7 c7 stall 1
  ```
COMPRESS LOOP: SPEED-UP

- Run for 1,000,000 elements, repeated 1000 times:
  - ifort -xcore-avx2 -qopt-report=3 driver.F90 compress.f90; ./a.out
    - 13 secs
  - ifort -xmic-avx512 -qopt-report=3 driver.F90 compress.f90; ./a.out
    - 0.8 secs
  - Similar for C version

- Speed-up depends on compression factor
  - Less for high compression

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
Motivation for Conflict Detection

- Sparse computations are common in HPC, but hard to vectorize due to race conditions
- Consider the “scatter” or “histogram” problem:

```c
for(i=0; i<16; i++) {
    A[B[i]]++;
}
```

```c
index = vload &B[i] // Load 16 B[i]
old_val = vgather A, index // Grab A[B[i]]
new_val = vadd old_val, +1.0 // Compute new values
vscatter A, index, new_val // Update A[B[i]]
```

- Problem if two vector lanes try to increment the same histogram bin
- Code above is wrong if any values within B[i] are duplicated
  - Only one update from the repeated index would be registered!
- A solution to the problem would be to avoid executing the sequence gather-op-scatter with vector of indexes that contain conflicts
The **VPCONFLICT** instruction detects elements with previous conflicts in a vector of indexes

- Allows to generate a mask with a subset of elements that are guaranteed to be conflict free
- The computation loop can be re-executed with the remaining elements until all the indexes have been operated upon

```c
index = vload &B[i]; // Load 16 B[i]
pending_elem = 0xFFFF; // all still remaining
do {
    curr_elem = get_conflict_free_subset(index, pending_elem)
    old_val = vgather {curr_elem} A, index // Grab A[B[i]]
    new_val = vadd old_val, +1.0 // Compute new values
    vscatter A {curr_elem}, index, new_val // Update A[B[i]]
    pending_elem = pending_elem ^ curr_elem // remove done idx
}
```

**VPCONFLICT instr.**

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPCONFLICT(D,Q) zmm2/mem, zmm1{k1}</td>
</tr>
<tr>
<td>VPTESTNM(D,Q) zmm2, zmm3/mem, zmm2, k2{k1}</td>
</tr>
<tr>
<td>VPBROADCASTM(W2D,B2Q) k2, zmm1</td>
</tr>
<tr>
<td>VPLZCNT(D,Q) zmm2/mem, zmm1 (k1)</td>
</tr>
</tbody>
</table>
With Intel® AVX2, this does not vectorize

- Store to \( h \) is a scatter
- \( ih \) can have the same value for different values of \( i \)
- Vectorization with a SIMD directive would cause incorrect results

for (i=0; i<n; i++) {
    y = sinf(x[i]*twopi);
    ih = floor((y-bot)*invbinw);
    ih = ih > 0 ? ih : 0;
    ih = ih < nbin ? ih : nbin;
    h[ih] = h[ih] + 1;
}

! Accumulate histogram of sin(x) in \( h \\
\text{do } i=1,n \\
y = \text{sin}(x(i)\ast\text{twopi}) \\
ih = \text{ceiling}((y-\text{bot})\ast\text{invbinw}) \\
ih = \text{min}(\text{nbin},\text{max}(1,ih)) \\
h(ih) = h(ih) + 1 \\
\text{enddo}
HISTOGRAMMING WITH INTEL® AVX-512 CD

- Compile for Intel® Xeon Phi™ processor x200 family:
  
  ifort -c -xmic-avx512 histo2.f90 -qopt-report-file=stderr -qopt-report=3 -S

  LOOP BEGIN at histo2.f90(11,4)
  
  remark #15300: LOOP WAS VECTORIZED
  remark #15458: masked indexed (or gather) loads: 1
  remark #15459: masked indexed (or scatter) stores: 1
  remark #15478: estimated potential speedup: 13.930
  remark #15499: histogram: 2

  LOOP END

  vpminsd %zmm20, %zmm5, %zmm3
  vpconflictd %zmm3, %zmm1
  vpgatherdd (%r13,%zmm3,4), %zmm6{%k1}
  vptestmd .L_2il0floatpacket.5(%rip), %zmm1, %k0
  vpaddw %zmm21, %zmm6, %zmm2

  vpbroadcastmw2d %k1, %zmm4
  vpaddw %zmm21, %zmm2, %zmm2{%k1}
  vptestmd %zmm1, %zmm4, %k0{%k1}
  vpscatterdd %zmm2, (%r13,%zmm3,4){%k1}

Some remarks omitted
HISTOGRAMMING WITH INTEL® AVX-512 CD

- Compile for Intel® Xeon Phi™ processor x200 family:
  ifort -c -xmic-avx512 histo2.f90 -qopt-report-file=stderr -qopt-report=3 –S

LOOP BEGIN at histo2.f90(11,4)
  remark #15300: LOOP WAS VECTORIZED
  remark #15458: masked indexed (or gather) loads: 1
  remark #15459: masked indexed (or scatter) stores: 1
  remark #15478: estimated potential speedup: 13.930
  remark #15499: histogram: 2
LOOP END

Some remarks omitted

vpminsmd %zmm20, %zmm5, %zmm3
vpconflictd %zmm3, %zmm1
# work on simd lanes without conflicts
vpgatherdd (%r13,%zmm3,4), %zmm6{%k1} # load h
vpptestmd .L_2il0floatpacket.5(%rip), %zmm1, %k0
vpaddd %zmm21, %zmm6, %zmm2 #increment h
...
vpbroadcastmw2d %k1, %zmm4
vplzcntd %zmm1, %zmm4
vpptestmd %zmm1, %zmm4, %k0{%k1}
kmovw %r10d, %k0
vptestmd %zmm1, %zmm4, %k0{%k1}
kmovw %k0, %r10d
testl %r10d, %r10d
jne ..B1.18
...
vscatterdd %zmm2, (%r13,%zmm3,4){%k1} # final store
HISTOGRAMMING WITH INTEL® AVX-512: SPEED-UP

- Run time for Intel® AVX2 (non-vectorized): 59 secs
  Intel® AVX-512 (vectorized): 6.6 secs

- Speed-up depends on problem details
  - Comes mostly from vectorization of other heavy computation in the loop
    - Not from the scatter itself
  - Speed-up may be (much) less if there are many conflicts
    - E.g. histograms with a singularity or narrow spike
    - Similar behavior for C and Fortran versions

- Other problems map to this
  - E.g. energy deposition in cells in particle transport Monte Carlo simulation
GATHER TO SHUFFLE (“G2S”) OPTIMIZATION
or “Adjacent Gather Optimization”

for (j=0; j<n; j++) {
    y[j] = x[j][1] + x[j][2] + x[j][3] + x[j][4] ...
}

• Elements of x are adjacent in memory, but vector index is in other dimension
• Compiler generates simd loads and shuffles for x instead of gathers
  • Before AVX-512: gather of x[1][1], x[2][1], x[3][1], x[4][1],...
  • With AVX-512: SIMD loads of x[1][1], x[1][2], x[1][3], x[1][4] etc.,
    followed by permutes to get back to x[1][1], x[2][1], x[3][1], x[4][1] etc.
• Message in optimization report:
  remark #34029: adjacent sparse (indexed) loads optimized for speed
• Large arrays of short vectors or structs are very common
G2S EXAMPLE - ARRAY OF STRUCTURES

float sumsq(struct Point *ptvec, int n) {
    float t_sum = 0;
    int i;
    // #pragma omp simd reduction(+:t_sum)
    // #pragma vector nog2s
    for (i = 0; i < n; i++) {
        // loop over points
        t_sum += ptvec[i].x * ptvec[i].x;
        t_sum += ptvec[i].y * ptvec[i].y;
        t_sum += ptvec[i].z * ptvec[i].z;
    
    #ifdef VEC4
        t_sum += ptvec[i].t * ptvec[i].t;
    #endif
    }
    return t_sum;
}

struct Point {
    float x;
    float y;
    float z;
    #ifdef VEC4
        float t;
    #endif
};

Calculate sum of squares of components of a vector

Driver loops 100000 times over a vector of 10000 points
G2S EXAMPLE

- From the version 15 compiler optimization report:  (15.0.7.235)

  $ icc -xmic-avx512 -qopt-report=4 test_g2s.c sumsq.c

  LOOP BEGIN at sumsq.c(9,9)
  
  remark #15415: vectorization support: gather was generated for the variable ptvec: strided by 3  
  [ sumsq.c(10,22) ]
  remark #15415: vectorization support: gather was generated for the variable ptvec: strided by 3  
  [ sumsq.c(10,35) ]

  ...

  remark #15300: LOOP WAS VECTORIZED
  remark #15460: masked strided loads: 6

  LOOP END

- Problem: pt[0].x, pt[1].x and pt[2].x are not adjacent in memory
  - \( \Rightarrow \) gather instructions are generated  (slow)

  - pt[0].x, pt[0].y and pt[0].z are adjacent in memory, but we’re not vectorizing in this dimension
    - Much better to vectorize loop over many points than loop over 3 components

  - Run time 1.5 secs
CAN WE DO BETTER?

SIMD load (unit stride) →

| pt[0].x | pt[0].y | pt[0].z | ...
|--------|--------|--------|-----
| pt[1].x | pt[1].y | pt[1].z | ...
| pt[2].x | pt[2].y | pt[2].z | ...
| ...    | ...    | ...    | ...

Transpose using Intel® AVX-512 permute instructions, vperm...
Enables unit stride SIMD loads as well as SIMD arithmetic
Supported in version 16 and 17 compilers
Compiler options: -xmic-avx512 -qopt-report=4
LOOP BEGIN at sumsq.c(9,9)
  remark #15415: vectorization support: non-unit strided load was generated for the variable `<ptvec->x[i]>`, stride is 3
...
remark #15305: vectorization support: vector length 16
  remark #15300: LOOP WAS VECTORIZED
  remark #15452: unmasked strided loads: 6
...
Report from: Code generation optimizations [cg]
sumsq.c(10,22):remark #34030: adjacent sparse (strided) loads optimized for speed. Details: stride { 12 }, types { F32-V512, F32-V512, F32-V512 }, number of elements { 16 }, select mask { 0x000000007 }.
...
- Run time 0.7 seconds
  - Disable G2S with #pragma nog2s, to conform it is responsible
  - Run time reverts to 1.5 secs
A BIGGER STRUCT

- Suppose our points have 4 dimensions, not 3 (compile with -DVEC4)
  - By default, compiler constructs and vectorizes an inner loop over components

LOOP BEGIN at sumsq.c(9,9)
remark #15542: loop was not vectorized: inner loop was already vectorized
...
LOOP BEGIN at sumsq.c(14,13)
...
remark #15305: vectorization support: vector length 4
remark #15427: loop was completely unrolled
remark #15301: MATERIALIZED LOOP WAS VECTORIZED

- Run time 1.9 secs
  - Inner loop trip count is short; better to vectorize outer loop
To enforce vectorization of outer loop, use:

```c
#pragma omp simd  reduction(+:t_sum)
```

```bash
icc -xmic-avx512 -qopenmp-simd -qopt-report=4 -DVEC4 test_g2s.c sumsq.c
```

LOOP BEGIN at sumsq.c(9,9)

remark #15415: vectorization support: non-unit strided load was generated for the variable <ptvec->x[i]>, stride is 4
remark #15305: vectorization support: vector length 16
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15452: unmasked strided loads: 8

... Report from: Code generation optimizations [cg]
sumsq.c(10,22):remark #34030: adjacent sparse (strided) loads optimized for speed. Details: stride { 16 }, types { F32-V512, F32-V512, F32-V512, F32-V512 }, number of elements { 16 }, select mask { 0x00000000F }.

We have outer loop vectorization back, followed by the G2S optimization

- Run time 0.8 secs
FROM INTEL® SSE OR INTEL® AVX TO INTEL® AVX-512:

Setting Expectations

- Doubling or quadrupling the vector length to 512 bits can boost application performance, but typically by less than 2x or 4x
  - Applications have scalar sections, so subject to Amdahl’s Law
  - Some applications are limited by access to data
    - If throughput bound, high bandwidth memory may help
    - If latency bound, prefetching may help
  - Loops may need larger trip counts to get full benefit
- But gains from newly vectorized loops can be large (new instructions!)
  - SIMD directives not yet applicable to new vector loop types
  - Working on new clauses...
- Application hotspots may change
Additional Resources

https://software.intel.com/articles/getting-ready-for-KNL
https://software.intel.com/xeon-phi/x200-processor
From Knights Corner to Knights Landing:
https://software.intel.com/articles/xeon-phi-software
https://software.intel.com/articles/intel-software-development-emulator
ADDITIONAL RESOURCES (OPTIMIZATION)

Webinars:
https://software.intel.com/articles/further-vectorization-features-of-the-intel-compiler-webinar-code-samples

Vectorization Guide (C):

Explicit Vector Programming in Fortran:
https://software.intel.com/articles/explicit-vector-programming-in-fortran

Initially written for Intel® Xeon Phi™ coprocessors, but also applicable elsewhere:
https://software.intel.com/articles/vectorization-essential

Compiler User Forums at http://software.intel.com/forums
LEGAL 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.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Xeon, Xeon Phi and the Intel logo are trademarks of Intel Corporation in the U.S. and/or 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