From Serial to Awesome: Advanced Code Vectorization and Optimizations

Anoop Madhusoodhanan Prabha
Martyn J Corden

Agenda

- Search Loop Pattern
- Valarray in Standard C++ Library
- OpenMP4.0 SIMD Constructs Support
- Expand/Compress Loop Pattern
- Histogram Loop Pattern

Changing Hardware Impacts Software

More cores → More Threads → Wider vectors

	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge EP	Intel® Xeon® processor code-named Ivy Bridge EP	Intel® Xeon® processor code-named Haswell EP	Intel® Xeon Phi™ coprocessor Knights Corner	Intel® Xeon Phi™ processor & coprocessor Knights Landing[1]
Core(s)	1	2	4	6	8	12	18	61	60+
Threads	2	2	8	12	16	24	36	244	
SIMD Width	128	128	128	128	256	256	256	512	

*Product specification for launched and shipped products available on ark.intel.com. 1. Not launched or in planning.

High performance software must be both:

- Parallel (multi-thread, multi-process)
- Vectorized

Search Loop Pattern

```
void mysearch(int n1, int *n2, int &retindex, int nsearch) {
      int i;
     for(i = 0; i < nsearch; i++)
      {
            if(n2[i] == n1)
                  break;
      }
      retindex = i;
      if(i >= nsearch) retindex = -1;
}
```

- Search loops are common in applications especially in the High Performance Computing or Big Data space
- Search loops are used either used for finding a substring in a bigger string or search for a number in an unsorted array or vector.
- Example code shows a search loop which searches for the first number in the array which is equal to n1.

Vectorization report and ASM targeting Intel® SSE2

Optimization Report

$ icpc -c search.cc -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout

```
LOOP BEGIN at search.cc(3,2)
<Peeled loop for vectorization>
LOOP END
LOOP BEGIN at search.cc(3,2)
   remark #15305: vectorization support: vector length 4
   remark #15309: vectorization support: normalized vectorization
overhead 0.433
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 21
   remark #15477: vector loop cost: 7.500
   remark #15478: estimated potential speedup: 2.710
   remark #15488: --- end vector loop cost summary ---
LOOP END
LOOP BEGIN at search.cc(3,2)
<Remainder loop for vectorization>
LOOP END
```

Generated Assembly Code

```
# parameter 1: %edi
# parameter 2: %rsi
# parameter 3: %rdx
# parameter 4: %ecx

        movd      %edi, %xmm0                   #1.6
        movl      %esi, %r10d                   #3.2
        pshufd    $0, %xmm0, %xmm0              #1.6
..B1.13:                        # Preds ..B1.14 ..B1.12
        movdqa    (%r8,%r10,4), %xmm1           #5.29
        pcmpeqd   %xmm0, %xmm1                  #5.29
        movmskps  %xmm1, %eax                   #5.29
        testl     %eax, %eax                    #5.29
        jne       ..B1.26       # Prob 20%      #5.29
..B1.14:                        # Preds ..B1.13
        addl      $4, %esi                      #3.26
        addq      $4, %r10                      #3.26
        movl      %esi, %eax                    #3.26
        cmpl      %edx, %esi                    #3.2
        jb        ..B1.13       # Prob 82%      #3.2
```

Computations on Arrays

```
#define SIZE 10000
#define NTIMES 10000
void test(std::array<float,SIZE> &vi, std::array<float,SIZE> &va)
{
   for(int i = 0; i < SIZE; i++)
      vi[i] = 2.f*sin(va[i]);
}

            int main()
{
   std::array<float,SIZE>  vi;
   std::array<float,SIZE>  va;
   float mysum;
   int i,j, size1, ntimes;
   double execTime = 0.0;
   double startTime, endTime;
   size1=SIZE;
   ntimes=NTIMES;
   for (int i=0; i<SIZE; i++) {
              vi[i] = (float) i;
              va[i] = (float)(i+10);
             }
   startTime = clock_it();
   for (j=0; j<ntimes; j++) {
     test(vi, va);
   }
   endTime = clock_it();
..
..
}
```

- Every scientific computing application deals with really huge arrays of either floats or doubles.

- Code snippet computes the sine value of every element in “va” array multiplied by 2 and stores it in corresponding “vi” array location.

```
$ icpc driver1.cc test1.cc -xavx -std=c++11 -o test2
$ ./test2
sum= -4.012442   -1.088042   -0.227893
```
time = 271 ms

Converting the program to use Valarray

```
#define SIZE 10000
#define NTIMES 10000
void test(std::valarray<float> &vi, std::valarray<float> &va)
{
  vi = 2.f*sin(va);
}

              int main()
{
   std::valarray<float>  vi(SIZE);
   std::valarray<float>  va(SIZE);
   float mysum;
   int i,j, size1, ntimes;
   double execTime = 0.0;
   double startTime, endTime;
   size1=SIZE;
   ntimes=NTIMES;
   for (int i=0; i<SIZE; i++) {
               vi[i] = (float) i;
               va[i] = (float)(i+10);
              }
   startTime = clock_it();
   for (j=0; j<ntimes; j++) {
     test(vi, va);
   }
   endTime = clock_it();
..
..
}
```

- Valarray is part of standard C++ library provided by both GNU C++ and Microsoft.
- Valarray provides operator overloaded functions for basic operators and math functions.
- Unlike the previous case, the function test() doesn't need a "for loop" to iterate through the individual elements.
- Introduced mainly to help with huge array based computations in High Performance Computing space.

```
$ icpc driver.cc test.cc -xavx -o test1
$ ./test1
sum= -4.012444   -1.088042   -0.227893
```

time = 136 ms

Speedup w.r.t array version ~ 2x

Use Intel® Integrated Performance Primitives version of Valarray

•Intel® Integrated Performance Primitives provides a more optimal optimization.

•valarray header is provide lers

•Intel® C++ Compiler by de iler on Linux* and Microsoft Compiler on Wir

•To use the Intel optimized e-intel-optimized-headers

$ icpc driver.cc test.cc -xav ers
$ ./test3
sum= -4.012444 -1.088042 -0.227893
time = 94 ms
Speedup w.r.t previous valarray version = 1.46x
Speedup w.r.t array version = 2.91x

System Specifications:

Processor: Intel(R) Core(TM) i5-4670T CPU @ 2.30GHz
RAM: 8GB
Operating System: Ubuntu* 12.04 (Linux* kernel - 3.8.0-29-generic)
GCC Compatibility mode : 4.6.3
Intel® Compiler Version: 16.0

Article URL: https://software.intel.com/en-us/articles/using-improved-stdvalarray-with-intelr-c-compiler

OpenMP* SIMD Constructs Support

- OpenMP* specifications includes support for explicit vector programming.
- Tools offered are:
  - #pragma omp simd [clauses] – Ignores the compiler heuristics and provides developers opportunity to force vectorization
  - #pragma omp declare simd [clauses] – Enables creating vector functions which can take vector arguments, do computations in SIMD mode and return vector result
- One handy use of explicit vectorization tools is to force outer loop vectorization.
- Compiler by default always tries to vectorize the inner loop. If the trip count of the inner loop is minimal when compared to outer loop trip count, it is worth checking the trade off force the outer loop vectorization.

Default Vectorization by Compiler targets inner loops

```
#define BLOCKSIZE 64

void foo(float *__restrict local_y, float *Acoefs, float *local_x, \
       int *Acols, int *Arowoffsets, int ib, int top)
{
      for(int row=ib*BLOCKSIZE; row<top; ++row) {
            local_y[row]=0.0;
      for(int i=Arowoffsets[row]; i<Arowoffsets[row+1]; ++i)
            local_y[row] += Acoefs[i]*local_x[Acols[i]];
      }
}
```

$ icpc -c -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout outer.cc -xavx -qopenmp

```
        LOOP BEGIN at outer.cc(5,2)
<Distributed chunk2>
  remark #15542: loop was not vectorized: inner loop was already vectorized
  LOOP BEGIN at outer.cc(8,6)
    remark #15388: vectorization support: reference Acoefs has aligned access   [
outer.cc(9,9) ]
    remark #15389: vectorization support: reference Acols has unaligned access   [
outer.cc(9,9) ]
    remark #15381: vectorization support: unaligned access used inside loop body
    remark #15305: vectorization support: vector length 8
    remark #15399: vectorization support: unroll factor set to 2
    remark #15309: vectorization support: normalized vectorization overhead 1.133
    remark #15300: LOOP WAS VECTORIZED
    remark #15442: entire loop may be executed in remainder
    remark #15448: unmasked aligned unit stride loads: 1
    remark #15450: unmasked unaligned unit stride loads: 1
    remark #15458: masked indexed (or gather) loads: 1
    remark #15475: --- begin vector loop cost summary ---
    remark #15476: scalar loop cost: 12
    remark #15477: vector loop cost: 3.750
    remark #15478: estimated potential speedup: 2.780
    remark #15488: --- end vector loop cost summary ---
  LOOP END
```

OpenMP* SIMD Constructs in Action

```
#define BLOCKSIZE 64

#pragma omp declare simd uniform(local_y, local_x, Acoefs, Acols,
Arowoffsets),linear(row:1),simdlen(8)
void InnerLoop(float *local_y, float *local_x, float *Acoefs, int *Acols,
int *Arowoffsets, int row);

void InnerLoop(float *local_y, float *local_x, float *Acoefs, int *Acols,
int *Arowoffsets, int row){
      for(int i=Arowoffsets[row]; i<Arowoffsets[row+1]; ++i)
            local_y[row] += Acoefs[i]*local_x[Acols[i]];
}

void foo(float *__restrict local_y, float *Acoefs, float *local_x, \
       int *Acols, int *Arowoffsets, int ib, int top)
{
      #pragma omp simd safelen(8)
      for(int row=ib*BLOCKSIZE; row<top; ++row) {
            local_y[row]=0.0;
            InnerLoop(local_y, local_x, Acoefs, Acols, Arowoffsets, row);
      }
}
```

```
Begin optimization report for: InnerLoop..xN8uuuuul(float
*, float *, float *, int *, int *, int)

    Report from: Vector optimizations [vec]

remark #15389: vectorization support: reference
Arowoffsets has unaligned access   [ omp.cc(7,18) ]
remark #15389: vectorization support: reference
Arowoffsets has unaligned access   [ omp.cc(7,8) ]
remark #15389: vectorization support: reference local_y
has unaligned access   [ omp.cc(8,3) ]
remark #15389: vectorization support: reference local_y
has unaligned access   [ omp.cc(8,3) ]
remark #15381: vectorization support: unaligned access
used inside loop body   [ <compiler generated> ]
remark #15301: FUNCTION WAS VECTORIZED   [
omp.cc(6,101) ]
```

Compress Loop Pattern

```
void foo(double *restrict A, double *B, int N){
  int i, j;
  for (i=0,j=0;i< N;i++){
   __assume_aligned(A, 64);
   __assume_aligned(B, 64);
    if (B[i]>0){
      A[j] = B[i];  // compress positive values of B[]
into A[]
      j++;
    }
  }
}
```

- Compress the positive values in B array into A array.
- Code will not vectorize with Intel® AVX2
- Code will vectorize on Intel® MIC architecture.
- Intel® AVX-512 offers vcompresspd instruction which operates in SIMD mode.
- _mm_mask_compressstoreu_pd/_mm256_mask_compressstoreu_pd are the compiler intrinsics. More documentation available at https://software.intel.com/en-us/node/582687

Vectorization report and ASM targeting Intel® AVX2

Optimization Report

$ icpc -c -S compress.cc -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout –restrict –xCORE-AVX2

```
        LOOP BEGIN at compress.cc(5,3)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed FLOW dependence between j line 8 and j line 7
  remark #15346: vector dependence: assumed ANTI dependence between j line 7 and j line 8
LOOP END

LOOP BEGIN at compress.cc(5,3)
<Remainder>
LOOP END
```

Generated Assembly Code

```
        # parameter 1(A): %rdi
        # parameter 2(B): %rsi

        vxorpd    %xmm0, %xmm0, %xmm0                   #6.14
..B1.4:                 # Preds ..B1.8 ..B1.3
        lea       (%r8,%r8), %r9d                       #6.9
        movslq    %r9d, %r9                             #6.9
        vmovsd    (%rsi,%r9,8), %xmm1                   #6.9
        vcomisd   %xmm0, %xmm1                          #6.14
        jbe       ..B1.6        # Prob 50%              #6.14
..B1.5:                 # Preds ..B1.4
        vmovsd    %xmm1, (%rdi,%rax,8)                  #7.7
        incq      %rax                                  #8.7
```

Optimization Report Generated for Compress Loop

Targeting Intel® Xeon Phi™ Coprocessor

$icpc -mmic -c -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout testcase.cc -S

```
        LOOP BEGIN at testcase.cc(3,2)
   remark #15389: vectorization support: reference A has unaligned access   [ testcase.cc(7,4) ]
   remark #15388: vectorization support: reference B has aligned access   [ testcase.cc(7,4) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15305: vectorization support: vector length 16
   remark #15399: vectorization support: unroll factor set to 4
   remark #15309: vectorization support: normalized vectorization overhead 0.022
   remark #15300: LOOP WAS VECTORIZED
   remark #15457: masked unaligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 26
   remark #15477: vector loop cost: 2.870
   remark #15478: estimated potential speedup: 8.450
   remark #15488: --- end vector loop cost summary ---
   remark #15497: vector compress: 1
LOOP END
```

Targeting Intel® AVX-512

$ icpc -xCOMMON-AVX512 -c -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout testcase.cc -S

```
        LOOP BEGIN at testcase.cc(3,2)
   remark #15388: vectorization support: reference B has aligned access   [ testcase.cc(7,4) ]
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized vectorization overhead 0.120
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15457: masked unaligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 17
   remark #15477: vector loop cost: 1.560
   remark #15478: estimated potential speedup: 10.130
   remark #15488: --- end vector loop cost summary ---
   remark #15497: vector compress: 1
LOOP END
```

Expand Loop Pattern

```
void foo1(double *restrict A, double *B, int N){
__assume_aligned(A, 64);
__assume_aligned(B, 64);
  int i, j;
  for (i=0,j=0;i< N;i++){
    if (B[i]>0){
      B[i] = A[j];  // expand A[] into where B[] is
positive
      j++;
    }
  }
}
```

- Expand the values in A array into locations in B array which has positive values.
- Code will not vectorize with Intel® AVX2
- Code will vectorize on Intel® MIC architecture.
- Intel® AVX-512 offers vexpandpd instruction which operates in SIMD mode.
- _mm_mask_expandloadu_pd/_mm256_mask_expandloadu_pd are the compiler intrinsics. More documentation available at https://software.intel.com/en-us/node/582681

Vectorization report and ASM targeting Intel® AVX2

Optimization Report

$ icpc -c -S expand.cc -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout -restrict -xCORE-AVX2

```
LOOP BEGIN at expand.cc(5,3)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed FLOW dependence between j line 8 and j line 7
  remark #15346: vector dependence: assumed ANTI dependence between j line 7 and j line 8
LOOP END

      LOOP BEGIN at expand.cc(5,3)
<Remainder>
LOOP END
```

Generated Assembly Code

```
        # parameter 1: %rdi
        # parameter 2: %rsi
        # parameter 3: %edx

        vxorpd    %xmm0, %xmm0, %xmm0                   #6.14
..B1.4:                         # Preds ..B1.8 ..B1.3
        lea       (%rcx,%rcx), %r9d                     #6.9
        movslq    %r9d, %r9                             #6.9
        vmovsd    (%rsi,%r9,8), %xmm1                   #6.9
        vcomisd   %xmm0, %xmm1                          #6.14
        jbe       ..B1.6        # Prob 50%              #6.14
..B1.5:                         # Preds ..B1.4
        movq      (%rdi,%rdx,8), %r10                   #7.14
        incq      %rdx                                  #8.7
        movq      %r10, (%rsi,%r9,8)                    #7.7
```

Optimization Report Generated for Expand Loop

Targeting Intel® Xeon Phi™ Coprocessor

```
$ icpc -c -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout
expand.cc -mmic –restrict
LOOP BEGIN at expand.cc(5,3)
  remark #15388: vectorization support: reference B has aligned access   [
expand.cc(7,4) ]
  remark #15415: vectorization support: gather was generated for the variable A:
masked, indirect access    [ expand.cc(7,11) ]
  remark #15305: vectorization support: vector length 16
  remark #15309: vectorization support: normalized vectorization overhead 0.039
  remark #15300: LOOP WAS VECTORIZED
  remark #15462: unmasked indexed (or gather) loads: 1
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 26
  remark #15477: vector loop cost: 4.810
  remark #15478: estimated potential speedup: 5.050
  remark #15488: --- end vector loop cost summary ---
  remark #15498: vector expand: 1
LOOP END
LOOP BEGIN at expand.cc(5,3)
<Remainder loop for vectorization>
  remark #15388: vectorization support: reference B has aligned access   [
expand.cc(7,4) ]
  remark #15305: vectorization support: vector length 16
  remark #15309: vectorization support: normalized vectorization overhead 0.167
  remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

Targeting Intel® AVX-512

```
$ icpc -c -S expand.cc -qopt-report=5 -qopt-report-phase=vec -qopt-report-
file=stdout -restrict -xCOMMON-AVX512
LOOP BEGIN at expand.cc(5,3)
  remark #15388: vectorization support: reference B has aligned access   [
expand.cc(7,7) ]
  remark #15305: vectorization support: vector length 16
  remark #15309: vectorization support: normalized vectorization overhead 0.125
  remark #15300: LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 1
  remark #15455: masked aligned unit stride stores: 1
  remark #15456: masked unaligned unit stride loads: 1
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 17
  remark #15477: vector loop cost: 2.500
  remark #15478: estimated potential speedup: 6.330
  remark #15488: --- end vector loop cost summary ---
  remark #15498: vector expand: 1
LOOP END
LOOP BEGIN at expand.cc(5,3)
<Remainder loop for vectorization>
  remark #15388: vectorization support: reference B has aligned access   [
expand.cc(7,7) ]
  remark #15305: vectorization support: vector length 16
  remark #15309: vectorization support: normalized vectorization overhead 0.304
  remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

Vectorization report and ASM targeting Intel® AVX-512

Optimization Report

```
$ icpc -c -S expand.cc -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout -restrict -xCOMMON-AVX512
LOOP BEGIN at expand.cc(5,3)
   remark #15388: vectorization support: reference B has aligned access   [ expand.cc(7,7) ]
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized vectorization overhead 0.125
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15455: masked aligned unit stride stores: 1
   remark #15456: masked unaligned unit stride loads: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 17
   remark #15477: vector loop cost: 2.500
   remark #15478: estimated potential speedup: 6.330
   remark #15488: --- end vector loop cost summary ---
   remark #15498: vector expand: 1
LOOP END
LOOP BEGIN at expand.cc(5,3)
<Remainder loop for vectorization>
   remark #15388: vectorization support: reference B has aligned access   [ expand.cc(7,7) ]
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized vectorization overhead 0.304
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

Generated Assembly Code

```
        vpxorq    %zmm0, %zmm0, %zmm0                   #6.14
..B1.4:                         # Preds ..B1.4 ..B1.3
        vcmppd    $9, (%rax), %zmm0, %k1                #6.14
        addq      $16, %r10                             #5.3
        vcmppd    $9, 64(%r11,%r12), %zmm0, %k2         #6.14
        kmovw     %k1, %r13d                            #6.14
        kmovw     %k2, %r14d                            #6.14
        movslq    %r8d, %r8                             #7.14
        addq      $128, %r11                            #5.3
        popcnt    %r13d, %r13d                          #7.14
        popcnt    %r14d, %r14d                          #7.14
        vexpandpd (%rbx,%r8,8), %zmm1{%k1}              #7.14
        addl      %r13d, %r8d                           #7.14
        movslq    %r8d, %r8                             #7.14
        vexpandpd (%rbx,%r8,8), %zmm2{%k2}              #7.14
        addl      %r14d, %r8d                           #7.14
        vmovapd   %zmm1, (%rax){%k1}                    #7.7
        vmovapd   %zmm2, 64(%rax){%k2}                  #7.7
        addq      $128, %rax                            #5.3
        cmpq      %r9, %r10                             #5.3
        jb        ..B1.4        # Prob 82%              #5.3
```

Histogram Loop Pattern

```
void foo(float *__restrict A, int *B){
      for(int i=0; i<16; i++)
            A[B[i]]++;
}
```

- Histogram loops are common in image processing and even in genetic engineering algorithms.
- The challenge here that, the vector pulled from B array can have duplicate values. For instance: for a vector length of 4 “1 2 1 2”. In this case, there is a conflict while enabling SIMD increment for A array since there are two increment operations for the same location in A.
- Intel® AVX-512 architecture offers a conflict detection instruction vpconflictd/vpconflictq
- More information on conflict detection instructions and intrinsics are available at https://software.intel.com/en-us/node/582719

Vectorization report and ASM targeting Intel® AVX2

Optimization Report

$ icpc conflict.cc -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout -c -xCORE-AVX2

```
        LOOP BEGIN at conflict.cc(2,2)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed FLOW dependence between A line 3 and A line 3
  remark #15346: vector dependence: assumed ANTI dependence between A line 3 and A line 3
LOOP END
```

Generated Assembly Code

```
    # parameter 1(A): %rdi
    # parameter 2(B): %rsi

movslq   (%rsi), %rax                          #3.6
movss    .L_2il0floatpacket.0(%rip), %xmm0     #3.4
movss    (%rdi,%rax,4), %xmm1                  #3.4
addss    %xmm0, %xmm1                          #3.4
movss    %xmm1, (%rdi,%rax,4)                  #3.4
movslq   4(%rsi), %rdx                         #3.6
movss    (%rdi,%rdx,4), %xmm2                  #3.4
addss    %xmm0, %xmm2                          #3.4
movss    %xmm2, (%rdi,%rdx,4)                  #3.4
```

Vectorization report and ASM targeting Intel® AVX-512

Optimization Report

```
$ icpc conflict.cc -qopt-report=5 -qopt-report-phase=vec -qopt-report-file=stdout -c
-xCOMMON-AVX512
LOOP BEGIN at conflict.cc(2,2)
   remark #15389: vectorization support: reference B has unaligned access   [
conflict.cc(3,4) ]
   remark #15389: vectorization support: reference B has unaligned access   [
conflict.cc(3,4) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15416: vectorization support: scatter was generated for the variable A:
indirect access    [ conflict.cc(3,4) ]
   remark #15415: vectorization support: gather was generated for the variable A:
indirect access    [ conflict.cc(3,4) ]
   remark #15305: vectorization support: vector length 16
   remark #15427: loop was completely unrolled
   remark #15309: vectorization support: normalized vectorization overhead 0.042
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15458: masked indexed (or gather) loads: 1
   remark #15459: masked indexed (or scatter) stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 15
   remark #15477: vector loop cost: 4.430
   remark #15478: estimated potential speedup: 3.240
   remark #15488: --- end vector loop cost summary ---
   remark #15499: histogram: 2
LOOP END
```

Generated Assembly Code

```
        vmovups   (%rsi), %zmm4                         #3.6 c1
        kxnorw    %k1, %k1, %k1                         #3.4 c1
        vpxord    %zmm0, %zmm0, %zmm0                   #3.4 c1
        vmovups   .L_2il0floatpacket.0(%rip), %zmm2     #3.4 c1
        kmovw     %k1, %k2                              #3.4 c3
        vpconflictd %zmm4, %zmm1                        #3.4 c7 stall 1
        vgatherdps (%rdi,%zmm4,4), %zmm0{%k2}           #3.4 c7
        vptestmd  .L_2il0floatpacket.1(%rip), %zmm1, %k0  #3.4 c9
        kmovw     %k0, %eax                             #3.4 c13 stall 1
        vaddps    %zmm2, %zmm0, %zmm3                   #3.4 c13
        testl     %eax, %eax                            #3.4 c15
        je        ..B1.7      # Prob 30%                #3.4 c17
..B1.2:                       # Preds ..B1.1
        vmovups   .L_2il0floatpacket.2(%rip), %zmm0     #3.4 c1
        vptestmd  .L_2il0floatpacket.3(%rip), %zmm1, %k0  #3.4 c1
        vplzcntd  %zmm1, %zmm5                          #3.4 c1
        xorb      %dl, %dl                              #3.4 c1
        kmovw     %k0, %eax                             #3.4 c5 stall 1
        vpsubd    %zmm5, %zmm0, %zmm0                   #3.4 c7
..B1.3:                       # Preds ..B1.5 ..B1.2
        vpbroadcastd %eax, %zmm5                        #3.4 c1
        kmovw     %eax, %k2                             #3.4 c1
        vpermps   %zmm3, %zmm0, %zmm3{%k2}              #3.4 c3
        vaddps    %zmm2, %zmm3, %zmm3{%k2}              #3.4 c5
        vptestmd  %zmm1, %zmm5, %k0{%k2}                #3.4 c7
        kmovw     %k0, %ecx                             #3.4 c11 stall 1
        andl      %ecx, %eax                            #3.4 c13
..B1.7:                       # Preds ..B1.5 ..B1.3 ..B1.1
        vscatterdps %zmm3, (%rdi,%zmm4,4){%k1}          #3.4 c1
        ret                                             #4.1 c3
```

Additional Resources

- Intel® C++ Compiler 16.0 User's Guide
- Intel® Fortran Compiler 16.0 User's Guide
- Using improved std::valarray with Intel® C++ Compiler
- Getting the most out of the Intel Compiler with new Optimization reports
- New Vectorization Features of the Intel Compiler
- Intel® C++ Compiler Samples
- Vectorization Essentials
- Outer Loop Vectorization
- Intel® Architecture Instruction Set Extensions Programming Reference
- Code Samples used in Webinar

Legal Disclaimer & Optimization Notice

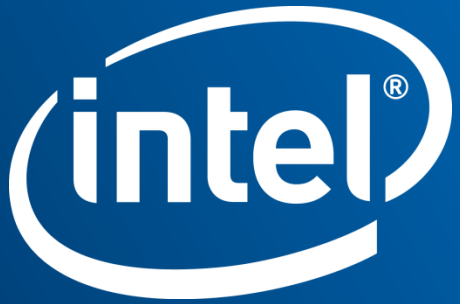
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING 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.

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


intel®

Backup Slides

Vectorization report and ASM for Compress Loop targeting Intel® AVX-512

Optimization Report

```
$ icpc -c -S compress.cc -qopt-report=5 -qopt-report-phase=vec -qopt-report-
file=stdout -restrict -xCOMMON-AVX512
LOOP BEGIN at compress.cc(5,3)
   remark #15388: vectorization support: reference B has aligned access   [
compress.cc(7,7) ]
   remark #15305: vectorization support: vec
   remark #15309: vectorization support: nor
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit str
   remark #15457: masked unaligned unit str
   remark #15475: --- begin vector loop cost
   remark #15476: scalar loop cost: 17
   remark #15477: vector loop cost: 1.560
   remark #15478: estimated potential speedu
   remark #15488: --- end vector loop cost summary ---
   remark #15497: vector compress: 1
LOOP END
LOOP BEGIN at compress.cc(5,3)
<Remainder loop for vectorization>
   remark #15388: vectorization support: reference B has aligned access   [
compress.cc(7,7) ]
   remark #15305: vectorization support: vector length 8
   remark #15309: vectorization support: normalized vectorization overhead 0.625
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

K0 to K7 are new OPMASK registers introduced in AVX-512 instructions which is used to predicate vector data processing. More information available at section 2.5.1 in Intel® Architecture Instruction Set Extensions Programming Reference at https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf

Generated Assembly Code

```
        vpxorq    %zmm0, %zmm0, %zmm0                   #6.14
..B1.4:                         # Preds ..B1.6 ..B1.3
        vmovups   (%rcx,%r14,8), %zmm1                  #6.9
        vmovups   64(%rcx,%r14,8), %zmm2                #6.9
                  %zmm1, %k1                            #6.14
                  %zmm2, %k2                            #6.14
                                                        #6.14
                                                        #6.14
                                                        #6.14
                                                        #6.14
                  0%                                    #6.14
                  B1.4
                                                        #7.7
                                                        #7.7
        popcnt    %ebx, %ebx                            #7.7
        popcnt    %r12d, %r8d                           #7.7
        vcompresspd %zmm1, (%rax,%rdi,8){%k1}           #7.7
        addl      %ebx, %edi                            #7.7
        movslq    %edi, %rdi                            #7.7
        addl      %edi, %r8d                            #7.7
        vcompresspd %zmm2, (%rax,%rdi,8){%k2}           #7.7
..B1.6:                         # Preds ..B1.5 ..B1.4
        addq      $16, %r14                             #5.3
        movl      %r8d, %edi                            #7.7
        cmpq      %r13, %r14                            #5.3
        jb        ..B1.4        # Prob 82%              #5.3
```