Intel® Compiler Features + Performance Tips

Rakesh Krishnaiyer

Principal Engineer

Intel Compiler Lab

Performance Analysis

- Compiler optimization reports are a useful tool to gain insight into:
  - What was done (and not done) by the compiler
  - Important to understand the interactions between multiple optimizations
    - Inlining
    - Openmp parallelization
    - Loop optimizations
    - Vectorization
- Reports are based on static compiler analysis
  - No dynamic information available
  - Hence the reports are most useful when correlated with performance analysis tools that do hotspot analysis and provide other dynamic information
    - Once this information is available, one can study the optimization information for hotspots (functions/loopnests) in compiler reports
    - Compiler can generate multiple versions of loop-nests, important to correlate with the actual executed version at runtime
- Lot of compiler loop optimizations geared for best vectorization
  - Phase ordering of loop opts relative to vectorization and each other
  - Often understanding the loop optimization parameters can help tuning
    - In many cases, finer control available via pragmas/options

Common Optimization Switches

	Windows*	Linux* Mac OS* X
Disable optimization	/Od	-O0
Optimize for speed (no code size increase)	/O1	-O1
Optimize for speed (default) – includes significant level of loop optimizations	/O2	-O2
More aggressive loop optimizations	/O3	-O3
Create symbols for debugging	/Zi	-g
Multi-file inter-procedural optimization	/Qipo	-ipo
Profile guided optimization (multi-step build)	/Qprof-gen /Qprof-use	-prof-gen -prof-use
Optimize for speed across the entire program **warning: -fast def'n changes over time "-fp-model fast=2" implies –complex-limited-range and –fimf-domain-exclusion=15	/fast (same as: /O3 /Qipo /Qprec-div- /QxHost –fp-model fast=2)	-fast (same as: -ipo –O3 -no-prec-div -static –xHost –fp-model fast=2)
OpenMP 4.0 support	/Qopenmp	-openmp
Automatic parallelization	/Qparallel	-parallel

Optimization Notice

Recent Target-specific Compiler Options

- -xMIC-AVX512: Optimizes code for KNL
- -xCORE-AVX512: Optimizes code for Xeon SKX (Skylake Server)
- -axMIC-AVX512 or –axCORE-AVX512
  - Two versions: baseline and another optimized for KNL or Xeon SKX
  - 'baseline': governed by implied –x flag, default sse2
- -axMIC-AVX512,CORE-AVX512
  - Three versions: baseline, KNL optimized, Xeon SKX optimized
- -xCOMMON-AVX512
  - Generates instructions that run on KNL and Xeon SKX
  - Libraries may use KNL/Xeon SKX specific instructions but will be cpu dispatched
  - Possible performance loss but has advantage of one binary that runs on KNL and Xeon SKX
- -xCORE-AVX2: Optimizes code for HSW
- –mmic: Generate code for KNC

Opt-report: Main Compiler Flags of Interest

- -opt-report[=N]
  - Default level is N=2
- -opt-report-phase=<vec,loop,openmp,ipo,…>
  - Default is all, recommend to use this to get full picture
- -opt-report-file= stdout | stderr | filename
- -vec-report[N] -par-report[N] -openmp-report[N]
  - Shorthand for the subset of –opt-report
  - Use only when default report is too verbose

Optimization Report Phases

- The compiler reports optimizations from 9 phases:
  - LOOP: Loop Nest Optimizations
  - PAR: Auto-Parallelization
  - VEC: Vectorization
  - OPENMP: OpenMP
  - OFFLOAD: Offload
  - IPO: Interprocedural Optimizations
  - PGO: Profile Guided Optimizations
  - CG: Code Generation Optimizations
  - TCOLLECT: Trace Analyzer Collection

  LOOP/PAR/VEC share a unified loop structure, a hierarchical output, to display optimizations in an integrated format.

- Selecting phases for compiler optimization reporting is highly customizable to satisfy customers' specific requirements.
  - Single Phase Reporting:
    - Compiler Option: -[Q]opt-report-phase=VEC
  - Multiple Phase Reporting (use a comma separated list):
    - Compiler Option: -[Q]opt-report-phase=VEC, OPENMP, IPO, LOOP
  - Default is "ALL" phases and default reporting verbosity level is 2
    - Want to encourage customers to use integrated HPO report instead of just -vec-report[n]

Optimization Report Levels

- The compiler's optimization report have 5 verbosity levels.
  - Specifying report verbosity level:
    - Compiler Option: –[Q]opt-report=N   where N = level of desired verbosity
    - For each optimization phase, higher verbosity level indicates higher level of detail reported.
    - Each verbosity level is inclusive of lower levels.
  - Example, VEC Phase Levels:
    - Level 1: Reports when vectorization has occurred.
    - Level 2: Adds diagnostics why vectorization did not occur.
    - Level 3: Adds vectorization loop summary diagnostics.
    - Level 4: Adds additional available vectorization support information.
    - Level 5: Adds detailed data dependency information diagnostics.
  - Each phase can support up to 5 levels

Vec/Par/Loop

- Loop Optimization report shows loop nest in hierarchical manner
  - Every loop version created gets its own set of opt-messages (+ a header in some cases)
  - Each message has a unique id for easy “help” access
- Vectorization/Parallelization reports have unified look & feel with Loop Optimization reports
- Caller/Callee info available as part of loop report
- 15.0 messages are more actionable – whenever possible
- Reports a message whenever compiler turns off optimizations when it hits internal limits
  - *Optimization for this routine was skipped to constrain compile time. Consider overriding limits (-qoverride-limits)*
  - *compile time constraints prevent loop vectorization, consider –O3*

Output

- Opt-report output goes to *.optrpt file by default, no longer stderr
  - Output files are always created from scratch (no appending behavior)
- With “-g” (Linux) / ”-Zi” (Windows), ASM code and OBJ code will have extra loop-info
  - In non-debug mode, use option to embed loop-info   -opt-report-embed=T
  - Text-mode output is more complete than loop-info embedded in object file
  - More on this in Vec/Par/Loop section

Annotated Assembly Listings

```
.L11:         # optimization report
              # LOOP WAS INTERCHANGED
              # loop was not vectorized: not inner loop
        xorl      %edi, %edi                              #38.3
        movsd     b.279.0.2(%rax,%rsi,8), %xmm0           #41.32
        unpcklpd  %xmm0, %xmm0                            #41.32
                        # LOE rax rcx rbx rsi rdi r12 r13 r14 r15 edx xmm0
..B1.11:                # Preds ..B1.11 ..B1.10
..L12:        # optimization report
              # LOOP WAS INTERCHANGED
              # LOOP WAS VECTORIZED
              # VECTORIZATION HAS UNALIGNED MEMORY REFERENCES
              # VECTORIZATION SPEEDUP COEFFECIENT 2.250000
        movaps    a.279.0.2(%rcx,%rdi,8), %xmm1           #41.22
        movaps    16+a.279.0.2(%rcx,%rdi,8), %xmm2        #41.22
        movaps    32+a.279.0.2(%rcx,%rdi,8), %xmm3        #41.22
        movaps    48+a.279.0.2(%rcx,%rdi,8), %xmm4        #41.22
        mulpd     %xmm0, %xmm1                            #41.32
        mulpd     %xmm0, %xmm2                            #41.32
<…>
```

- Asm listing produced with “–S –g”

```
L4::        ; optimization report
            ; PEELED LOOP FOR VECTORIZATION
$LN36:
$LN37:
    vaddss   xmm1, xmm0, DWORD PTR [r8+r10*4]     ;4.5

snip snip snip

L5::        ; optimization report
            ; LOOP WAS VECTORIZED
            ; VECTORIZATION HAS UNALIGNED MEMORY REFERENCES
            ; VECTORIZATION SPEEDUP COEFFECIENT 8.398438
$LN46:
    vaddps   ymm1, ymm0, YMMWORD PTR [r8+r9*4]    ;4.5

snip snip snip

L6::        ; optimization report
            ; LOOP WAS VECTORIZED
            ; REMAINDER LOOP FOR VECTORIATION
            ; VECTORIZATION HAS UNALIGNED MEMORY REFERENCES
            ; VECTORIZATION SPEEDUP COEFFECIENT 2.449219
$LN78:
    add      r10, 4                               ;3.3

snip snip snip

L7::        ; optimization report
            ; REMAINDER LOOP FOR VECTORIATION
$LN93:
    inc      rax                                  ;3.3
```

Vec Analysis - Utilizing Full vectors

- Typical vectorized loop consists of:
  - Peel loop - generated as an optimization for aligning some accesses
  - Vector kernel loop - Highest performing, best if all vector execution happen here
  - Remainder loop - required for correctness unless compiler can prove trip-count is multiple of vec-length
- Peel loop and remainder loops are (most likely) vectorized by compiler – less efficiency
  - Any unrolling of kernel vector loop also affects max iterations in remainder
  - Most loads/stores become masked
- Larger vector register means more iterations in peel/remainder – in degenerate cases, all execution will happen in peel/remainder loops significantly reducing benefits from vectorizing loop

Utilizing Full Vectors – Simple Example1

Scellrb5% cat -n t10.c

```
#include <stdio.h>

void foo1(float * restrict a, float *b, float *c, int n)
{
  int i;
  for (i=0; i<n; i++) {
    a[i] += b[i] * c[i];
  }
}

```

scellrb5%:icc -O2 -opt-report4 -opt-report-file=stderr t10.c -restrict -c -mmic

Example1 Pseudo Code

```
Peel loop until A is aligned

if ( B is aligned )

      for () {  // Kernel vectorized loop1, unrolled by 2

         [al64] A = [al64] A + [al64] B * C

         [al64] A = [al64] A + [al64] B * C

else

      for () {  // Alternate alignment kernel loop2, unrolled by 2

         [al64] A = [al64] A + B * C

         [al64] A = [al64] A + [al64] B * C

      }

endif

Remainder loop to execute remaining iterations
```

Peel Loop Report – Example1

Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]

LOOP BEGIN at t10.c(6,3)

<Peeled>

remark #15389: vectorization support: reference a has unaligned access [t10.c(7,5)]

remark #15389: vectorization support: reference a has unaligned access [t10.c(7,5)]

remark #15389: vectorization support: reference b has unaligned access [t10.c(7,5)]

remark #15389: vectorization support: reference c has unaligned access [t10.c(7,5)]

remark #15381: vectorization support: unaligned access used inside loop body

remark #15301: PEEL LOOP WAS VECTORIZED

LOOP END

Kernel Loop Report – Example1

LOOP BEGIN at t10.c(6,3)

remark #15388: vectorization support: reference a has aligned access [t10.c(7,5)]

remark #15388: vectorization support: reference a has aligned access [t10.c(7,5)]

remark #15389: vectorization support: reference b has aligned access [t10.c(7,5)]

remark #15388: vectorization support: reference c has unaligned access [t10.c(7,5)]

remark #15381: vectorization support: unaligned access used inside loop body

remark #15399: vectorization support: unroll factor set to 2

remark #15300: LOOP WAS VECTORIZED

remark #15450: unmasked unaligned unit stride loads: 1

remark #15475: --- begin vector loop cost summary ---

remark #15476: scalar loop cost: 15

remark #15477: vector loop cost: 1.120

remark #15478: estimated potential speedup: 20.270

remark #15488: --- end vector loop cost summary ---

LOOP END

LOOP BEGIN at t10.c(6,3)

<Alternate Alignment Vectorized Loop>

LOOP END

Optimization Notice

Remainder Loop Report – Example1

LOOP BEGIN at t10.c(6,3)
<Remainder>
  remark #15388: vectorization support: reference a has aligned access [t10.c(7,5)]
  remark #15388: vectorization support: reference a has aligned access [t10.c(7,5)]
  remark #15389: vectorization support: reference b has unaligned access [t10.c(7,5)]
  remark #15389: vectorization support: reference c has unaligned access [t10.c(7,5)]
  remark #15388: vectorization support: reference a has aligned access [t10.c(7,5)]
  remark #15388: vectorization support: reference a has aligned access [t10.c(7,5)]
  remark #15389: vectorization support: reference b has unaligned access [t10.c(7,5)]
  remark #15389: vectorization support: reference c has unaligned access [t10.c(7,5)]
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END

Utilizing Full Vectors – What you can do

- If hotspot analysis on performance tool shows lot of time spent in peel or remainder loops, this may be worth looking into
  - Happens often when trip-count is unknown to compiler (say, n1) – but actual value at runtime is small
- Choose algorithm blocking parameters to have high trip-counts for kernel loops (relative to any peel/remainder loops)
- Align arrays – no need for peel loop
- Use loop_count pragma to convey information to compiler
  - Especially useful for low trip-count loops for compiler to make better decisions
- Several controls available:
  - #pragma nounroll (to disable unroll of vector kernel loops)
  - #pragma vector noremainder (to disable vectorization of peel/remainder loops)
  - #pragma vector unaligned (don't generate peel loop)
  - #pragma novector (disable vectorization altogether)
- Use –opt-assume-safe-padding if possible (specific to KNC)

Tips for Low Trip-count Loops

- Ideal for compiler if trip-count and array extents are statically known
  - Such as a Fortran “parameter” (#define in C)
  - Vectorization cost-model decisions are easier for compiler – whether or not to peel, unroll-factor, …
  - Compiler analysis of alignment for vectorization is much more effective
  - Prefetch distances chosen by the compiler are more effective
  - Compiler is able to do outer-loop optimizations much more efficiently
    - PRE (partial redundancy elimination) for address calculations
    - Unroll of outer-loop
    - PDSE (partial dead store elimination) in outer-loop, etc.

Tips for Low Tripcount Loops - 2

- If trip-count and array extents are variables, it may be possible in some cases to make a specialized version via src-changes
- In cases where they will remain as variables, you can help the compiler:
  - Use loop_count pragma/directive to convey min/max/avg values
  - Can also use options such as –unroll0
  - Use alignment clauses per loop or per array
  - Use prefetch distance option to fine-tune
  - In some cases, applying !dir$ simd on the outer-loop may be better
  - Study the opt-report to make sure compiler is making reasonable optimization decisions

Optimization Notice

Blocking Example - NBody

```
for (body1=0; body1<NBODIES; body1++){
  for (body2=0; body2<NBODIES; body2++) {
     OUT[body1] += compute(body1, body2);
  }
}
```

• data (body2) is streamed from memory. Assuming NBODIES is large, we would have no reuse in cache => this application is memory bandwidth bound (app will run at the speed of memory to cpu speeds, less than optimal)

Modified Source Pseudo-code (with 1-D blocking):

```
for (body2=0; body2 <NBODIES; body2 +=BLOCK) {
   for (body1=0; body1 < NBODIES; body1 ++) {
     for (body22=0; body22 < BLOCK; body22 ++) {
         OUT[body1] += compute(body1,
                              body2 + body22);
     }
   }
 }
```

• data (body22) is kept and reused in cache => better performance

// Full source code

```
#define CHUNK_SIZE 8192

#pragma omp parallel private(body_start_index)
for(body_start_index=0; body_start_index<global_number_of_bodies;
body_start_index += CHUNK_SIZE) {
  int i, body_end_index = body_start_index + CHUNK_SIZE;

  #pragma omp for private(i) schedule(guided)
  #pragma unroll_and_jam (4) // unroll-jam done by compiler
  for(i=0; i<global_number_of_bodies; i++) {
   int j;
   TYPE acc_x_0 = 0, acc_y_0 = 0, acc_z_0 = 0;
   for(j=body_start_index; j<body_end_index; j+=1){
     TYPE delta_x_0 = Input_Position_X[(j+0)] - Input_Position_X[i];
     TYPE delta_y_0 = Input_Position_Y[(j+0)] - Input_Position_Y[i];
     TYPE delta_z_0 = Input_Position_Z[(j+0)] - Input_Position_Z[i];

     TYPE gamma_0 = delta_x_0*delta_x_0 +
          delta_y_0*delta_y_0 + delta_z_0*delta_z_0 + epsilon_sqr;
     TYPE s_0 = Mass[j+0]/(gamma_0 * SQRT(gamma_0));

     acc_x_0 += s_0*delta_x_0;
     acc_y_0 += s_0*delta_y_0;
     acc_z_0 += s_0*delta_z_0;
   }

   Output_Acceleration[3*(i+0)+0] += acc_x_0;
   Output_Acceleration[3*(i+0)+1] += acc_y_0;
   Output_Acceleration[3*(i+0)+2] += acc_z_0;
  }
}
```

Data Dependence - Multiversioning

- scellrb5% cat -n t8.c

```
#include <stdio.h>

void foo1(float *a, float *b, float *c, int n)
{
  int i;
#pragma vector aligned nontemporal
  for (i=0; i<n; i++) {
    a[i] *= b[i] + c[i];
  }
}
```

- scellrb5%: icc -O2 -opt-report4 -opt-report-file=stderr t8.c -restrict -c -xmic-avx512

Loop Report – Multiversioning for vec

- LOOP BEGIN at t8.c(7,3)
- <Multiversioned v1>
- remark #25228: Loop multiversioned for Data Dependence
- remark #15388: vectorization support: reference a has aligned access [t8.c(8,5)] …
- remark #15412: vectorization support: streaming store was generated for a [t8.c(8,5)]
- remark #15300: LOOP WAS VECTORIZED
- remark #15448: unmasked aligned unit stride loads: 3
- remark #15449: unmasked aligned unit stride stores: 1
- remark #15467: unmasked aligned streaming stores: 1
- remark #15475: --- begin vector loop cost summary ---
- remark #15476: scalar loop cost: 15
- remark #15477: vector loop cost: 0.430
- remark #15478: estimated potential speedup: 32.140
- remark #15488: --- end vector loop cost summary ---
- LOOP END

- LOOP BEGIN at t8.c(7,3)
- <Remainder, Multiversioned v1>
- remark #15388: vectorization support: reference a has aligned access [t8.c(8,5)] …
- remark #15301: REMAINDER LOOP WAS VECTORIZED
- LOOP END

- LOOP BEGIN at t8.c(7,3)
- <Multiversioned v2>
- remark #15304: loop was not vectorized: non-vectorizable loop instance from multiversioning
- remark #25439: unrolled with remainder by 2
- LOOP END

Avoid Manual Unrolling in Source

- Try not to use manual unroll: keep code simple
  - Common in legacy Fortran codes
- Simple legacy DAXPY Fortran code: (Y = A*X + Y on DP vectors)

```
m = MOD(N,4)
if ( m /= 0 ) THEN
    do i = 1 , m
       Dy(i) = Dy(i) + Da*Dx(i)
    end do
    if ( N < 4 ) RETURN
end if
mp1 = m + 1
do i = mp1 , N , 4
     Dy(i) = Dy(i) + Da*Dx(i)
     Dy(i+1) = Dy(i+1) + Da*Dx(i+1)
     Dy(i+2) = Dy(i+2) + Da*Dx(i+2)
     Dy(i+3) = Dy(i+3) + Da*Dx(i+3)
end do
```

Less than 4 iterations loop

Non-unit accesses everywhere ☹

- Rewriting in a simplest possible way helps:
  - Unit-stride accesses
  - Alignable manually or using peeling and multiversioning
  - Optimizable for all platforms
  - Much more readable

```
do i=1,N
  Dy(i) = Dy(i) + Da*Dx(i)
end do
```

Loop Interchange-Locality&Vectorization

scellrb5% cat –n d2.F90

```
#define np 16
subroutine orig(div)
  real*8, dimension (np,np), intent(inout) :: div
  integer :: i, j, l, k, n, m
    do j=1,np
      do i=1,np
        vtemp(i,j,1)=(Dinv(1,1,i,j)*v(i,j,1) + Dinv(1,2,i,j)*v(i,j,2))
        vtemp(i,j,2)=(Dinv(2,1,i,j)*v(i,j,1) + Dinv(2,2,i,j)*v(i,j,2))
      enddo
    enddo
    do n=1,np
      do m=1,np
        div(m,n)=0
        do j=1,np
          div(m,n)=div(m,n)-(spheremp(j,n)*vtemp(j,n,1)*Dvv(m,j) &
                             + spheremp(m,j)*vtemp(m,j,2)*Dvv(n,j)) &
                              * rrearth
        enddo
      end do
    end do
end subroutine orig
```

scellrb5%: ifort -O3 -xAVX –c -i_keep -qopt-report4 d2.F90

Loop Report – Distribution+Interchange

```
LOOP BEGIN at d2.F90(165,30)
<Distributed chunk1>
   remark #25426: Loop Distributed (2 way)
   remark #15541: outer loop was not auto-vectorized: consider using SIMD
    directive   [ d2.F90(162,11) ]

   LOOP BEGIN at d2.F90(160,8)
   <Distributed chunk1>
      remark #25426: Loop Distributed (2 way)
      remark #25408: memset generated
      remark #15398: loop was not vectorized: loop was transformed to memset or
    memcpy
   LOOP END
LOOP END
```

Loop Report–Distribution+Interchange(2)

```
LOOP BEGIN at d2.F90(165,30)
<Distributed chunk2>
   remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 1 3 2 )
   remark #15542: loop was not vectorized: inner loop was already vectorized
   LOOP BEGIN at d2.F90(163,11)
   <Distributed chunk2>
      remark #15542: loop was not vectorized: inner loop was already vectorized
      LOOP BEGIN at d2.F90(160,8)
         remark #15389: vectorization support: reference div has unaligned access   [ d2.F90(164,14) ]
         remark #15389: vectorization support: reference div has unaligned access   [ d2.F90(164,14) ]
         remark #15389: vectorization support: reference divspherewk_mp_dvv_ has unaligned access
      [ d2.F90(164,14) ]
         remark #15389: vectorization support: reference divspherewk_mp_spheremp_ has unaligned access
      [ d2.F90(164,14) ]
         remark #15389: vectorization support: reference divspherewk_mp_vtemp_ has unaligned access
      [ d2.F90(164,14) ]
         remark #15381: vectorization support: unaligned access used inside loop body
         remark #15301: PERMUTED LOOP WAS VECTORIZED
         remark #15450: unmasked unaligned unit stride loads: 4
         remark #15451: unmasked unaligned unit stride stores: 1
         remark #15475: --- begin vector loop cost summary ---
         remark #15476: scalar loop cost: 22
         remark #15477: vector loop cost: 11.750
         remark #15478: estimated potential speedup: 1.660
         remark #15488: --- end vector loop cost summary ---
         remark #25015: Estimate of max trip count of loop=4
      LOOP END
   LOOP END
LOOP END
```

Make Loop Induction Variables Local

scellrb5% cat –n d1.F90

```
Module divSphereWk
  real*8, Dimension(:,:,:), Allocatable :: v …
  integer :: i, j, l, k, n, m  // Global loop induction variables – bad idea
  public
Contains
  subroutine orig(div)
    real*8, dimension (np,np), intent(inout) :: div
      do n=1,np
         do m=1,np
            div(m,n)=0
            do j=1,np
               div(m,n)=div(m,n)-(spheremp(j,n)*vtemp(j,n,1)*Dvv(m,j) &
                             + spheremp(m,j)*vtemp(m,j,2)*Dvv(n,j)) &
                              * rrearth
            enddo
         end do
      end do
  end subroutine orig
```

scellrb5%: ifort -O3 -xAVX –c -i_keep -qopt-report4 d1.F90

- Global induction variables create imperfect nesting – affects loop opts

remark #25096: Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due to other Compiler Transformations)

Example of matmul 'C'

```
for j=1,1000  // Original loopnest
  for i = 1,1000
    a[j][i] = 0.0
    for k = 1,1000
         a[j][i] = a[j][i]  + b[k][i] *c[j][k];
    end for
  end for
end for
```

```
// Transformed Loopnest pseudo-code
for  j = 1, 1000
  for  i = 1, 1000
    a[j][i] = 0.0   // This 2-level loopnest will be
                    //  converted to a call to memcpy
  end for
end for
// outer three-level loop-blocking not shown
for j=1,1000,4        // unroll-jam by 4
  for k = 1,1000,4  // unroll-jam by 4
    for i = 1,1000    // this loop will be vectorized
      a[j][i] = a[j][i] + b[k][i] * c[j][k];
      a[j][i] = a[j][i] + b[k+1][i] * c[j][k+1];
      a[j][i] = a[j][i] + b[k+2][i] * c[j][k+2];
      a[j][i] = a[j][i] + b[k+3][i] * c[j][k+3];
      a[j+1][i] = a[j+1][i] + b[k][i] * c[j+1][k];
      a[j+1][i] = a[j+1][i] + b[k+1][i] * c[j+1][k+1];
      a[j+1][i] = a[j+1][i] + b[k+2][i] * c[j+1][k+2];
      a[j+1][i] = a[j+1][i] + b[k+3][i] * c[j+1][k+3];
      a[j+2][i] = a[t2+1][i] + b[k][i] * c[j+2][k];
       ...
      a[j+3][i] = a[j+3][i] + b[k][i] * c[j+3][k];
       ...
    end for
  end for
end for
```

Loop Blocking and Unroll-Jam

scellrb5% cat -n m4_single.f

```
       program main
       parameter (n=2048)
       double precision ,  dimension(n,n) :: a,b,c,ctest
       integer i,j,k, nerr
       double precision t,s, temp,freq
       real ( selected_real_kind(14) ) :: t2,t1,TIME,FLOPS,t_call
       freq = 2.67
       fopspercycle = 4
       print*, " Frequency of processor in Ghz  ",freq
        print*, " fp ops per cycle  ",fopspercycle

        do j=1,n
           do i = 1,n
              c(i,j) = 0
              do k = 1,n
                 c(i,j) = c(i,j) + a(i,k) * b(k,j)
              enddo
           enddo
        enddo
        print *, a,b,c
        end
```

scellrb5%: ifort -O3 -qopt-report2 -qopt-report-file=stderr m4_single.f -xmic-avx512

Optimization Notice

Loop Report–Distn+Blocking+Unroll-jam

```
LOOP BEGIN at m4_single.f(12,9)
<Distributed chunk1>
  remark #25426: Loop Distributed (2 way)
  remark #25420: Collapsed with loop at line 13
  remark #25408: memset generated
  remark #15398: loop was not vectorized: loop was transformed to memset or
   memcpy

  LOOP BEGIN at m4_single.f(13,12)
  <Distributed chunk1>
    remark #25426: Loop Distributed (2 way)
    remark #25421: Loop eliminated in Collapsing

  LOOP END
LOOP END
```

Distn+Blocking+Unroll-jam (2)

```
LOOP BEGIN at m4_single.f(12,9)
<Distributed chunk2>
   remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 1 3 2 )
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at m4_single.f(12,9)
   <Distributed chunk2>
      remark #15542: loop was not vectorized: inner loop was already vectorized

      LOOP BEGIN at m4_single.f(12,9)
         remark #15542: loop was not vectorized: inner loop was already vectorized

         LOOP BEGIN at m4_single.f(12,9)
         <Distributed chunk2>
            remark #25442: blocked by 128   (pre-vector)
            remark #25440: unrolled and jammed by 4   (pre-vector)
            remark #15542: loop was not vectorized: inner loop was already vectorized

            LOOP BEGIN at m4_single.f(15,15)
            <Distributed chunk2>
               remark #25442: blocked by 128   (pre-vector)
               remark #25440: unrolled and jammed by 4   (pre-vector)
               remark #15542: loop was not vectorized: inner loop was already vectorized

               LOOP BEGIN at m4_single.f(13,12)
                  remark #25442: blocked by 128   (pre-vector)
                  remark #15301: PERMUTED LOOP WAS VECTORIZED
                  remark #25456: Number of Array Refs Scalar Replaced In Loop: 36
               LOOP END
            LOOP END
```

Distn+Blocking+Unroll-jam (3)

```
        LOOP BEGIN at m4_single.f(15,15)
        <Remainder, Distributed chunk2>
          remark #15542: loop was not vectorized: inner loop was already vectorized

          LOOP BEGIN at m4_single.f(13,12)
            remark #15301: PERMUTED LOOP WAS VECTORIZED
            remark #25456: Number of Array Refs Scalar Replaced In Loop: 3
          LOOP END
        LOOP END
      LOOP END

      LOOP BEGIN at m4_single.f(12,9)
      <Remainder, Distributed chunk2>
        remark #15542: loop was not vectorized: inner loop was already vectorized

        LOOP BEGIN at m4_single.f(15,15)
        <Distributed chunk2>
          remark #15542: loop was not vectorized: inner loop was already vectorized

          LOOP BEGIN at m4_single.f(13,12)
            remark #15301: PERMUTED LOOP WAS VECTORIZED
          LOOP END
        LOOP END
      LOOP END
    LOOP END
  LOOP END
LOOP END
```


Optimization Notice

Pragmas to Fine-tune Loop Opts

- Sampling of loop-level controls available
  - Also useful to suppress particular loop transformations – useful for performance experiments
  - If turning off a particular loop opt hurts, there may be opportunity to fine-tune the parameter

- #pragma simd reduction(+:sum)
  - The loop is transformed as is, no other loop-optimizations will change the simd-loop
- #pragma loop_count min(220) avg (300) max (380)
  - Fortran syntax: **!dir$ loop count(16)**
- #pragma vector aligned nontemporal
  - #pragma novector // to suppress vectorization
- #pragma unroll(4)
  - #pragma unroll(0) // to suppress loop unrolling
- #pragma unroll_and_jam(2)
- #pragma nofusion
- #pragma distribute_point
  - If placed right after the for-loop, distribution will be suppressed for that loop
  - Fortran syntax: **!dir$ distribute point**
- #pragma forceinline (recursive)

Fortran – Unit-Stride is Important

- Fortran language semantics allow unit-stride vectorization for lots of array types such as allocatable arrays, adjustable arrays, explicit arrays, assumed-size arrays, etc.
  - Still requires vector-loop index to be in the last dimension
  - Using F90 array notation can help in cases where it is not
- F90 pointers and assumed-shape arrays get strided access (language semantics)
  - Compiler does versioning for unit-strides, but this is an optimization and may not help all cases
  - Using Fortran 2008 CONTIGUOUS attribute may help

Original src:

```
Do index=1,n
    A(I,j,k,index) = B(I,j,k,index) +
                     C(I,j,k,index) * D
enddo
```

- **Non-unit stride vectorization, since index is not in the innermost dimension**
- **Results in gathers/scatters**

Modified src:

```
Do index=1,n
    A(I:I+VLEN,j,k,index) = B(I:I+VLEN,j,k,index) +
                            C(I:I+VLEN,j,k,index) * D
Enddo
```

- **Use of F90 array notation helps here to vectorize with unit-stride**

Refactor for Efficient Unit-stride Vectors

- Use multi-dimensional arrays carefully to get full-VL unit-strided vectorization for most loops

```
do k=2,kbu
    mi  = mi0 + k
    f1 = mhdtddy*max(v(mi),zero)
    tmp(1:nc,k) = (t(1:nc,mi) + f1*(t(1:nc,mi)))
    htmp(k)     = mhdtddy*onethird*dt/h_new(mi)
enddo
```

- nc value is 2 or 12 - if source-code can be transformed to:

```
!dir$ simd  // Better to vectorize at outer-level with transformation below
do k=2,kbu
    mi  = mi0 + k
    f1 = mhdtddy*max(v(mi),zero)
    tmp(k,1:nc) = (t(mi,1:nc) + f1*(t(mi,1:nc)))
    htmp(k)     = mhdtddy*onethird*dt/h_new(mi)
enddo
```

  - Unit-strided vectorization by vectorizing outer-loop
  - If nc is constant and small, compiler will do complete unroll of inner-loop before vectorization (based on simd pragma on outer loop)
  - Useful even when nc is a variable unknown to compiler

Vectorization with Indirect Accesses

```
for (i = kstart; i < kend; ++i) {
  istart = iend;
  iend = mp_dofStart[i+1];
  float w = xd[i];

  for (j = istart; j < iend; ++j) {
      index = SCS[j];
      xd[index] -= lower[j]*w;
  }
}
```

- Key pre-requisite to vectorization is that the xd values are distinct
  - Otherwise, there are genuine dependences that will make the loop NOT vectorizable (without advanced instructions such as vconflict)
  - If that is the case, the only alternative is to rewrite the algorithm in a vector-friendly way
  - If the xd values are guaranteed (by the user) to be distinct, then one can use the ivdep/simd pragmas (before the inner j-loop) to vectorize
  - The compiler will still generate gather/scatter vectorization
    - If there is an alternative algorithmic formulation where unit-strides can be used, that may be beneficial

Vectorization with indirect access - contd

Whether gather/scatter helps (compared to scalar) will depend on:

- Whether there is any cache-locality for the indirect accesses – KNC hardware will be able to combine them if they happen to be in the same cache-line
- Whether all the data is in cache (as opposed to memory)
  - If they are getting accessed from memory, doing prefetching using intrinsics for the gather/scatter may help depending on the memory access pattern
  - Doing vectorization (or not) may not matter since your bottleneck is the memory access.
- Amount of other “vectorizable computation” inside the loop
  - Example in previous slide has only a simple fma, so not much to gain from vectorizing the “other” part.

Motivation for Conflict Detection

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

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

```
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]]
```

- 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

Conflict Detection Instructions Usage

- 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

VCONFLICT instr.
VPCONFLICT{D,Q} zmm1{k1}, zmm2/mem
VPBROADCASTM{W2D,B2Q} zmm1, k2
VPTESTNM{D,Q} k2{k1}, zmm2, zmm3/mem
VPLZCNT{D,Q} zmm1 {k1}, zmm2/mem

```
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
} while (pending_elem)
```

Report for vectorization using vconflict

```
LOOP BEGIN at t1.c(11,5)
   remark #15389: vectorization support: reference SCS has unaligned access   [ t1.c(12,7) ]
   remark #15389: vectorization support: reference lower has unaligned access   [ t1.c(13,7) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15416: vectorization support: scatter was generated for the variable xd:  indirect access [ t1.c(13,7) ]
   remark #15415: vectorization support: gather was generated for the variable xd:  indirect access [ t1.c(13,7) ]
   remark #15305: vectorization support: vector length 16
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15450: unmasked unaligned unit stride loads: 2
   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: 20
   remark #15477: vector loop cost: 5.060
   remark #15478: estimated potential speedup: 3.490
   remark #15488: --- end vector loop cost summary ---
   remark #15499: histogram: 2
   LOOP BEGIN at t1.c(13,7)
      remark #25460: No loop optimizations reported
   LOOP END
LOOP END
```

Inner-level vs. Outer-level Vectorization

When to consider outer-loop vectorization:

- If the average inner-loop trip-counts are low (not enough to fill up a full vector) and the outer-loop trip-counts are large, then you may get better vectorization-efficiency.
- Are there any expensive operations in the outer-loop (say a divide) that now get vectorized due to outer-loop vectorization?
- Unit-stride vs. gathers/scatters will change for inner vs. outer vectorization
- Potential gains have to be weighed taking any loss of unit-stride efficient vectorization compared to inner-loop vectorization

How to do Outer-level Vectorization

- Compiler may be able to vectorize a subset of these cases by adding the simd pragma (with the right clauses) to the outer-loop
- Directly vectorize at outer loop level by outlining the body of the outer-loop into a vector-elemental function and using the simd pragma
- Strip-mine outer loop iterations and change each statement in the loop body to operate on the strip
  - Intel® Cilk™ Plus array notation extension helps the programmers to express this approach in a natural fashion

Outer-level Vectorization by outlining outer-loop body into elemental function

Sparse-matrix-vector loop-pattern:

```
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]];
    }
}
```

- **Inner-loop vectorization gives:**
- unit-stride load for "Acoefs" and "Acols", and gathers for "local_x“
- "local_y" storage accesses get moved out of the loop and a reduction-temp will be introduced by the compiler

•**Wrong to use simd pragma without reduction clause**

```
#pragma simd
 for(LocalOrdinalType row=ib*BLOCKSIZE; row<top; ++row)
{
      local_y[row]=0.0;
      Inner_loop_elem_function(local_y, row, Acoefs, local_x,
                                             Acols, Arowoffsets);
 }

__declspec(vector(uniform(Arowoffsets, Acoefs, local_x,
                                  Acols, local_y), linear(row))))
Inner_loop_elem_function(float *local_y, int row,
      float *Acoefs, float *local_x, int *Acols, int *Arowoffsets)
{
    for(int i=Arowoffsets[row]; i<Arowoffsets[row+1];  ++i) {
         local_y[row] += Acoefs[i]*local_x[Acols[i]];
    }
}
```

•**Outer-loop vectorization gives:**
- For the inner-loop-body the compiler will generate unit-stride load/store for "local_y“
- "Acoefs" and "Acols" loads become gather
- "local_x" continues to be gather.

Prefetch Directive Support

Prefetch pragma support for C loops

- Apply uniform distance for all arrays in a loop:
  - #pragma prefetch *:hint:distance
- Fine-grained control for each array:
  - #pragma prefetch var:hint:distance
  - #pragma noprefetch var
- You can combine the two forms for the same loop

```
#pragma prefetch *:1:5
#pragma noprefetch A // prefetch only for B and C arrays
  for(int i=0; i<n; i++)  { C[i] = A[B[i]]; }
```

Prefetch directive support for Fortran loops

- Apply uniform distance for all arrays in a loop:
  - CDEC$ prefetch *:hint:distance
- Fine-grained control for each array:
  - CDEC$ prefetch var:hint:distance
  - CDEC$ noprefetch var

Prefetch Distance Tuning Option

-opt-prefetch-distance=n1[,n2]

- n1 specifies the distance for first-level prefetches into L2
- n2 specifies prefetch distance for second-level prefetches from L2 to L1 (use n2 <= n1)
- -opt-prefetch-distance=64,32
- -opt-prefetch-distance=24
  - Use first-level distance=24, second-level distance to be determined by compiler
- -opt-prefetch-distance=0,4
  - Turns off all first-level prefetches, second-level uses distance=4 (Use this if you want to rely on hardware prefetching to L2, and compiler prefetching from L2 to L1)
- -opt-prefetch-distance=16,0
  - First-level distance=16, no second-level prefetches issued
- If option not specified, all distances determined by compiler

Prefetch Performance Tuning

If algorithm is well blocked to fit in L2 cache, prefetching is less critical

For data access patterns where L2-cache misses are common , prefetching is critical

- Default compiler heuristics typically use a first-level prefetch distance of <=8 vectorized iterations
- For bandwidth-bound benchmarks (such as stream), using a larger first-level prefetch (vprefetch1) distance sometimes shows performance improvements
- If you see a performance drop when you turn off compiler-prefetching, the app is a likely candidate that will benefit from fine-tuning of compiler prefetches with options/pragmas

Prefetch Performance Tuning - Contd

Use different first-level (vprefetch1) and second-level prefetch (vprefetch0) distances to fine-tune your application performance

- -opt-prefetch-distance=n1[,n2]
- Useful values to try for n1: 0,4,8,16,32,64
- Useful values to try for n2: 0,1,2,4,8
- Can also use prefetch pragmas to do this on a per-loop basis
- Try –mP2OPT_hpo_pref_initial_vals=100 <large_value>

If your application hot-spots use indirect accesses (gather/scatter) or non-unit-strided accesses, then try enhanced compiler prefetching for such references (described more in later slides)

- Use appropriate pragma for each such loop OR
- Add option –mP2OPT_hlo_pref_indirect_refs=T
- Add option –mP2OPT_hlo_pref_multiple_pfes_strided_refs=T

C++ Example Using Lambda Function

```
typedef double* __restrict__  __attribute__((align_value (64))) Real_ptr;
typedef int Indx_type;

template <typename LOOP_BODY>
inline  __attribute__((always_inline))
void forall(Indx_type begin, Indx_type end, LOOP_BODY loop_body)
{
#pragma simd
#pragma vector aligned
#pragma prefetch *:1:25
#pragma prefetch *:0:2
  for ( Indx_type ii = begin ; ii < end ; ++ii ) {  loop_body( ii );  }
}
void foo8(Indx_type len, Real_ptr out1, Real_ptr out2, Real_ptr out3,
               Real_ptr in1, Real_ptr in2)
{
  forall(0, len, [&] (Indx_type i) {
    out1[i] = in1[i] * in2[i] ;
    out2[i] = in1[i] + in2[i] ;
    out3[i] = in1[i] - in2[i] ;
  } ) ;
}
```

C++ Ex. Using Lambda - Contd

```
$ icpc -c -qopt-report3 -qopt-report-phase=loop,vec star_pf7.cpp -
    std=c++0x -mmic -unroll0

LOOP BEGIN at star_pf7.cpp(12,4) inlined into star_pf7.cpp(17,4)
  remark #15301: SIMD LOOP WAS VECTORIZED
…
  remark #25018: Total number of lines prefetched in=10
  remark #25021: Number of initial-value prefetches=6
  remark #25035: Number of pointer data prefetches=10, dist=8
  remark #25149: Using directive-based hint=1, distance=25 for pointer
    data reference    [ star_pf7.cpp(18,21) ]
  remark #25141: Using second-level distance 2 for prefetching pointer
    data reference   [ star_pf7.cpp(18,21) ]
…
```

- Prefetch pragma using the * syntax to control all arrays inside the loop
- Command-line uses –unroll0 option for illustrative purposes only
    - In general, all unrolled cache-lines are prefetched irrespective of the unroll factor chosen by the compiler for the vectorized loop
- 5 arrays, 2 prefetches per array, 10 cache-lines prefetched inside the loop
- First-level prefetch distance =25 vectorized loop-iterations ahead

Optimization Notice

Loop Prefetch Example2

```
for(int y = y0; y < y1; ++y) {
    float div, *restrict A_cur = &A[t & 1][z * Nxy + y * Nx];
    float *restrict A_next = &A[(t + 1) & 1][z * Nxy + y * Nx];
    float *restrict vvv = &vsq[z * Nxy + y * Nx];
    for(int x = x0; x < x1; ++x) { // Typical trip-count is 192, 12 after vectorization
        div = c0 * A_cur[x]  + c1 * ((A_cur[x + 1] + A_cur[x - 1])
                         + (A_cur[x + _Nx] + A_cur[x - _Nx])
                         + (A_cur[x + Nxy] + A_cur[x - Nxy]))
                         + c2 * ((A_cur[x + 2] + A_cur[x - 2]) + ...
        A_next[x] = 2 * A_cur[x] - A_next[x] + vvv[x] * div;
    }
}
$ icc -O2 -qopt-report3 -qopt-report-phase=loop,vec p3_orig.cpp
...
remark #15301: LOOP WAS VECTORIZED.
remark #25018: Total number of lines prefetched=38
remark #25035: Number of pointer data prefetches=38, dist=8
...
```

- Prefetch coverage is low (dist =8) since typical trip-count is only 12
- Use –opt-prefetch-distance=2,1 (Or add pragmas)
- Or use loop-count directive before inner-loop: #pragma loop_count (192)

Optimization Notice

Adjacent Gather/Scatter Optimization Variants

- Few basic forms (all unmasked)
  - Strided loads and stores
    - Array of Structs
    - Fortran multi-dimensional arrays with last dimension completely unrolled
    - C/C++ pointer-of-pointers with constant last dimension
  - Support for indirect accesses – applications like miniMD
- Replace series of gathers with a series of vector loads of contiguous elements followed by a sequence of permutations/shuffles in the register file
- Compiler Targeting Priorities
  - Simple forms of this optimization enabled in 15.0 for KNC
  - Targeting more cases in 16.0
- Optimization is enabled as part of default O2 optimization level
  - User does not need to add any special options

Opt-report Example – Adjacent Gather

```
#include <stdio.h>

extern float dataf[];
static float resf[];
extern double datad[];
static double resd[];

void adjacent_access_unoptimized()
{
    int i = 0;

    for (i = 0; i< 6200; ++i) {
        float xij = dataf[4 * i];
        float yij = dataf[4 * i + 1];
        float zij = dataf[4 * i + 2];
        float tij = dataf[4 * i + 3];
        resf[i] = xij * xij + yij * yij + zij * zij + tij * tij;
    }
}

void adjacent_access_optimized()
{
    int i = 0;

    for (i = 0; i< 6200; ++i) {
        double xij = datad[3 * i];
        double yij = datad[3 * i + 1];
        double zij = datad[3 * i + 2];
        resd[i] = xij * xij + yij * yij + zij * zij;
    }
}
```

Compiled with:

icc -O2 -opt-report1 -opt-report-file=stderr t1.c -c -mmic –opt-report-phase=cg

CG Opt Report: Level 1

Features

Shows report for optimization of sparse memory accesses for each routine

- Reports whether an optimized instruction sequence is used for the sparse memory access pattern
- Usage of an unoptimized is a question for a compiler improvement
- Compiler report may also say "optimization unprofitable" (not shown)

Example

```
Begin optimization report for: adjacent_access_unoptimized()

    Report from: Code generation optimizations [cg]

t1.c(13,23):remark #34032: adjacent sparse (strided) loads are
        not optimized. Details: stride { 16 }, types { F32-V512, F32-
        V512, F32-V512, F32-V512 }, number of elements { 16 },
        select mask { 0x00000000F }.
==========================================

Begin optimization report for: adjacent_access_optimized()

    Report from: Code generation optimizations [cg]

t1.c(26,24):remark #34030: adjacent sparse (strided) loads
        optimized for speed. Details: stride { 24 }, types { F64-
        V512, F64-V512, F64-V512 }, number of elements { 8 },
        select mask { 0x000000007 }.
==========================================
```

CG Opt Report: KNC Assembly Snippet

- Unoptimized case:

```
..L12:  vgatherdps 4+dataf(%r9,%zmm0,8), %zmm1{%k3}          #13.23
        jkzd      ..L11, %k3    # Prob 50%                   #13.23
        vgatherdps 4+dataf(%r9,%zmm0,8), %zmm1{%k3}          #13.23
        jknzd     ..L12, %k3    # Prob 50%                   #13.23
..L11:  …
..L14: vgatherdps dataf(%r9,%zmm0,8), %zmm3{%k4}             #13.23
        jkzd      ..L13, %k4    # Prob 50%                   #13.23
        vgatherdps dataf(%r9,%zmm0,8), %zmm3{%k4}            #13.23
        jknzd     ..L14, %k4    # Prob 50%                   #13.23
..L13:  …
..L16: vgatherdps 8+dataf(%r9,%zmm0,8), %zmm4{%k2}           #13.23
        jkzd      ..L15, %k2    # Prob 50%                   #13.23
        vgatherdps 8+dataf(%r9,%zmm0,8), %zmm4{%k2}          #13.23
        jknzd     ..L16, %k2    # Prob 50%                   #13.23 …
..L15:  …
..L18: vgatherdps 12+dataf(%r9,%zmm0,8), %zmm5{%k1}          #13.23
        jkzd      ..L17, %k1    # Prob 50%                   #13.23
        vgatherdps 12+dataf(%r9,%zmm0,8), %zmm5{%k1}         #13.23
        jknzd     ..L18, %k1    # Prob 50%                   #13.23
..L17:
```

The four gather sequences remain in generated code – code generator does not create more efficient sequence

- Optimized case:

```
..B2.7:                     # Preds ..B2.7 ..B2.6 Latency 97
        vloadunpackld datad(,%rdx,8), %zmm3           #26.24 c1
        vloadunpackld 64+datad(,%rdx,8), %zmm7        #26.24 c5
        vloadunpackld 128+datad(,%rdx,8), %zmm2       #26.24 c9
        vloadunpackhd 64+datad(,%rdx,8), %zmm3        #26.24 c13
        vloadunpackhd 128+datad(,%rdx,8), %zmm7       #26.24 c17
        vloadunpackhd 192+datad(,%rdx,8), %zmm2       #26.24 c21
        …
        jb        ..B2.7        # Prob 99%            #25.7 c97
```

3-gather sequence converted by compiler to a more efficient sequence with 3 unaligned pair-loads (shown above) + permutes (not shown)

Coral Example–Strided Loads

- **Vectorized using 3 adj-gather: optimized seq generated for AVX,AVX2,KNC,KNL,SKX**

```
void foo(double* restrict min, __int32* restrict pos, const double (*in)[3], const
    double* displ, const int limit){
      int p = 0;       double t = in[0][0] + displ[0];       double m = t*t;
      t = in[0][1] + displ[1];       m += t*t;
      t = in[0][2] + displ[2];       m += t*t;
      for (int j = 1; j < limit; j++) {
            double t = in[j][0] + displ[0];    double v = t*t;
            t = in[j][1] + displ[1];           v += t*t;
            t = in[j][2] + displ[2];           v += t*t;
            if (v < m){  m = v;   p = j;  }
      }
      *min = m;       *pos = p;
}  // icc -O3 –xMIC-AVX512 -restrict -opt-report3 coral_qmcpack.cpp –c
```

Report from: Code generation optimizations [cg]
coral_qmcpack.cpp(11,28):remark #34030: adjacent sparse (strided) loads optimized for speed. Details: stride { 24 }, types { F64-V512, F64-V512, F64-V512 }, number of elements { 8 }, select mask { 0x000000007 }

Optimization Notice

Adjacent Gather/Scatter Optimization for AOS

```
typedef struct {
  double x;
  double y;
  double z;
} s;

s aos[16];

double foo( ){
   int i, j;
   double res = 0;
   for (i=0; i<16; i++) {
       double xt = aos[i].x;
       double yt = aos[i].y;
       double zt = aos[i].z;
       res += xt + yt + zt;
   }
   return res;
}
```

- **Current compiler can successfully detect a series of adjacent gathers/scatters that get generated due to accesses to multiple adjacent structure members and can optimize away the gathers/scatters.**
- **Optimized sequence generated for KNC, KNL, SKX**
- **a3_double.c(13,17):remark #34030: adjacent sparse (strided) loads optimized for speed. Details: stride { 24 }, types { F64-V512, F64-V512, F64-V512 }, number of elements { 8 }, select mask { 0x000000007 }**

Optimization Notice

miniMD: Adjacent Gather on KNC based on Indexed Loads

```
for(int k = 0; k < numneighs; k++) {
   const int j = neighs[k];
   double x = x[j * PAD + 0];
   double y = x[j * PAD + 1];
   double z = x[j * PAD + 2];
   …
}
```

A scenario of gathers in a particular version of miniMD

force_lj.cpp(417,37):remark #34029: adjacent sparse (indexed) loads optimized for speed. Details: types { F64-V512, F64-V512, F64-V512 }, number of elements { 8 }, select mask { 0x000000007 }

- Three gather's of eight double precision elements with offsets 0, 8, 16
- 8 byte padding is present at offset 24
- Full mask, no up-conversion, scale is 8, no NT hint
- 4-byte indices
- Same base; offsets are included into index vectors, so each gather uses its own index vector

1.11x improvement on KNC

Optimization Notice

Fortran Assumed Shape Array Parameter

Assumed shape arrays as parameters

```
subroutine ash(A, B, C)
    real, intent(inout), dimension(:) :: A
    real, intent(in), dimension(:) :: B
    real, intent(in), dimension(:) :: C
    A = A + B * C
    return
end
```

No information is passed explicitly by the programmer

- Implicit interface (dope vector) for extent, stride info
- Populated by the compiler, passed from caller to callee

Can have any stride

- Compiler does not generate packing/unpacking at call site

Assumed Shape Array Vectorization

Any stride is possible for each of the 3 arrays

- Multiversion code to check for stride at runtime
- How many versions? There are 2^3=8 combinations:
  - unitstride(A) & unitstride(B) & unitstride(C)
  - unitstride(A) & unitstride(B) & !unitstride(C)
  - unitstride(A) & !unitstride(B) & unitstride(C)
  - ...
  - !unitstride(A) & !unitstride(B) & !unitstride(C)
- Compiler generates 2 versions:
  - Ver1: All arrays are unitstride
  - Ver2: At least 1 array is non-unitstride
- Version 1 can be vectorized (on KNC) using vmovaps/vloadunpack (alignment)
- Version 2 can be vectorized using vgather

Assumed Shape Array Alignment

Each array can have arbitrary alignment

- User should help compiler with alignment assumptions (as before)
- Without user help, the compiler generates
  - A peel loop that iterates until one array is aligned
    - Preferred array to align is the one we store into (i.e., A)
  - Still (N-1) arrays could be unaligned
  - A multiversion code that checks alignment of B (2nd array)
  - No further multiversioning for array C (too deep version tree)

```
if( A,B,C all unit stride )
        Peel loop until A is aligned (uses vscatter for A)
        if( B is aligned )
                [al64] A = [al64] B + C   //Version 1a
        else
                [al64] A = B + C          //Version 1b
        endif
else
        A = B + C                         //Version 2
endif
```

Assumed Shape Array Multiversioning

Scellrb5% cat -n t7.f90

```
subroutine assumed_shape1(A, B, C)

real, intent(inout), dimension(:) :: A
real, intent(in), dimension(:) :: B, C

A = A + B * C

return
end
```

scellrb5%: ifort -O3 -opt-report5 -opt-report-file=stderr -xmic-avx512 -c t7.f90

Loop Report – Assumed Shape Array (1)

```
LOOP BEGIN at t7.f90(6,7)
<Peeled, Multiversioned v1>
   remark #15389: vectorization support: reference a has unaligned access …
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15301: PEEL LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at t7.f90(6,7)
<Multiversioned v1>
   remark #25233: Loop multiversioned for stride tests on Assumed shape arrays
   remark #15389: vectorization support: reference a has aligned access …
   remark #15389: vectorization support: reference b has unaligned access
   remark #15389: vectorization support: reference c has aligned access
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15399: vectorization support: unroll factor set to 2
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 1 …
LOOP END

LOOP BEGIN at t7.f90(6,7)
<Alternate Alignment Vectorized Loop, Multiversioned v1>
LOOP END

LOOP BEGIN at t7.f90(6,7)
<Remainder, Multiversioned v1> …
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

Loop Report – Assumed Shape Array (2)

```
LOOP BEGIN at t7.f90(6,7)
<Multiversioned v2>
  remark #15416: vectorization support: scatter was generated for the variable a:  strided by non-constant
    value
  remark #15415: vectorization support: gather was generated for the variable a:  strided by non-constant
    value
  remark #15415: vectorization support: gather was generated for the variable b:  strided by non-constant
    value
  remark #15415: vectorization support: gather was generated for the variable c:  strided by non-constant
    value
  remark #15399: vectorization support: unroll factor set to 2
  remark #15300: LOOP WAS VECTORIZED
  remark #15460: masked strided loads: 3
  remark #15462: unmasked indexed (or gather) loads: 1
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 9
  remark #15477: vector loop cost: 10.870
  remark #15478: estimated potential speedup: 1.530
  remark #15488: --- end vector loop cost summary --- …
LOOP END

LOOP BEGIN at t7.f90(6,7)
<Remainder, Multiversioned v2>
LOOP END
```


Optimization Notice

Compiler Strictly Follows Language Rules

- Are “long” and “int” same?
  - Not on GCC based Intel64 platforms
  - Potential Impact: portability and performance
    - Can complicate compiler analysis.
    - May lose vectorization
- 1.0 and sin() are double.
  1.0f and sinf() are float.
  - Hard to catch in the source code.
  - Unintended type converts all over in ASM code
  - Computation unintentionally expensive

```
float x[N], y[N];
for (i = 0; i < N; i++)
  y[i] = sin(x[i] + 1.0)+1.0
```

float | double | float | double

```
float x[N], y[N];
for (i = 0; I < N; i++)
  y[i] = sinf(x[i] + 1.0f)+1.0f
```

Optimization Notice

Software & Services Group

intel Software

Look for Hints of Vector Inefficiency

- Scellrb5% cat -n t12_sin.c

```
#include <stdio.h>

void foo1(float * restrict a, float *b, float *c, int n)
{
  int i;
  for (i=0; i<n; i++) {
    a[i] = sin(b[i] + 1.0);
  }
}

```

- scellrb5%: icc -O2 -qopt-report4 -qopt-report-file=stderr t12_sin.c -restrict -c -xmic-avx512

Kernel Loop Report–Vector Inefficiency

- LOOP BEGIN at t12_sin.c(6,3)
- remark #15389: vectorization support: reference b has unaligned access [t12_sin.c(7,12)]
- remark #15389: vectorization support: reference a has unaligned access [t12_sin.c(7,5)]
- remark #15381: vectorization support: unaligned access used inside loop body
- remark #15399: vectorization support: unroll factor set to 2
- remark #15417: vectorization support: number of FP up converts: single precision to double precision 1 [t12_sin.c(7,12)]
- remark #15418: vectorization support: number of FP down converts: double precision to single precision 1 [t12_sin.c(7,5)]
- remark #15300: LOOP WAS VECTORIZED
- remark #15442: entire loop may be executed in remainder
- remark #15450: unmasked unaligned unit stride loads: 1
- remark #15451: unmasked unaligned unit stride stores: 1
- remark #15475: --- begin vector loop cost summary ---
- remark #15476: scalar loop cost: 113
- remark #15477: vector loop cost: 10.060
- remark #15478: estimated potential speedup: 9.980
- remark #15482: vectorized math library calls: 1
- remark #15487: type converts: 2
- remark #15488: --- end vector loop cost summary ---
- LOOP END

Alignment and Module Data Known Sized Arrays

Example: Global arrays declared in modules with known size.

```
 module mymod
  !dir$ attributes align:64 :: a
  !dir$ attributes align:64 :: b
  real (kind=8) :: a(1000), b(1000)
end module mymod

subroutine add_them()
use mymod
implicit none
!  array syntax shown, could also be explicit loop
!...No explicit directive needed to say that A and B
!  are aligned, the USE brings that information

  a = a + b
end subroutine add_them
```

This saves coding effort AND improves performance!

Fortran Alignment Example

scellrb5% cat -n t2_mod.f90

```
module mymod
  !dir$ attributes align:64 :: a, b, c, d // Alternatively use –align array64byte
  real*8 :: a(1000), b(1000), c(1000), d(1000)
end module mymod
```

scellrb5% cat -n sub3.f90

```
subroutine add_them2(low1, up1)
use mymod
implicit none
integer low1, up1, i

! No explicit directive should be needed to tell the compiler that
! base pointers of A and B are aligned, the USE should bring that information.
! But since the lower bound of the loop is not 1, compiler does
! loop-peeling - once peeling is done for one array, all array
 ! accesses get fully aligned in the kernel loop

   do i=low1, up1
     a(i) = b(i) + c(i) + d(i)
   enddo
 end subroutine add_them2
```


Optimization Notice

Fortran Alignment Example (2)

```
LOOP BEGIN at sub3.f90(12,3)
<Peeled>
  remark #15389: vectorization support: reference a has unaligned access   [ sub3.f90(13,5) ]
   …
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #15301: PEEL LOOP WAS VECTORIZED
  remark #25015: Estimate of max trip count of loop=125
LOOP END
LOOP BEGIN at sub3.f90(12,3)
  remark #15388: vectorization support: reference a has aligned access   [ sub3.f90(13,5) ]
  remark #15388: vectorization support: reference b has aligned access   [ sub3.f90(13,5) ]
  remark #15388: vectorization support: reference c has aligned access   [ sub3.f90(13,5) ]
  remark #15388: vectorization support: reference d has aligned access   [ sub3.f90(13,5) ]
  remark #15399: vectorization support: unroll factor set to 4
  remark #15300: LOOP WAS VECTORIZED …
remark #25015: Estimate of max trip count of loop=31
LOOP END
LOOP BEGIN at sub3.f90(12,3)
<Remainder>
  remark #15388: vectorization support: reference a has aligned access   [ sub3.f90(13,5) ] …
  remark #15301: REMAINDER LOOP WAS VECTORIZED
  remark #25015: Estimate of max trip count of loop=125
LOOP END
```

Appln Example Using Elemental Function

```
#pragma simd // simd pragma for outer-loop at call-site of elemental-function
for (int i = beg*16; i < end*16; ++i)
   particleVelocity_block(px[i], py[i], pz[i], destvx + i, destvy + i, destvz + i, vel_block_start,
vel_block_end);

__declspec(vector(uniform(start,end), linear(velx,vely,velz)))
static void particleVelocity_block(const float posx, const float posy, const float posz,
                                   float *velx,     float *vely,       float *velz,   int start, int end) {
   __assume_aligned(velx,64);     __assume_aligned(vely,64);     __assume_aligned(velz,64);
   for (int j = start; j < end; ++j) {
      const float del_p_x  = posx - px[j];        const float del_p_y  = posy - py[j];
      const float del_p_z  = posz - pz[j];
      const float dxn= del_p_x * del_p_x + del_p_y * del_p_y + del_p_z * del_p_z +pa[j]* pa[j];
      const float dxctaui  = del_p_y * tz[j] - ty[j] * del_p_z;
      const float dyctaui  = del_p_z * tx[j] - tz[j] * del_p_x;
      const float dzctaui  = del_p_x * ty[j] - tx[j] * del_p_y;
      const float dst      = 1.0f/std::sqrt(dxn);
      const float dst3     = dst*dst*dst;
      *velx               -= dxctaui * dst3;
      *vely               -= dyctaui * dst3;
      *velz               -= dzctaui * dst3;
   }
}
```

- Performance improvement over 2X going from inner to outer-loop vectorization

Appln Example using Compress Idiom

```
int index_0 = 0;
for(int k0=0; k0<count0; k0++) {
    TYPE X1 = *(Pos0 + k0);        TYPE Y1 = *(Pos0 + k0 +   count0);
    TYPE Z1 = *(Pos0 + k0 + 2*count0);
    #pragma loop_count min(220) avg (300) max (380)
           for(int k1=0; k1<count1; k1+=1) {
        TYPE X0 = *(Pos1 + k1);
        TYPE Y0 = *(Pos1 + k1 +   count1);
        TYPE Z0 = *(Pos1 + k1 + 2*count1);
        TYPE diff_X = (X0 - X1);
        TYPE diff_Y = (Y0 - Y1);
        TYPE diff_Z = (Z0 - Z1);
        TYPE norm_2 = (diff_X*diff_X) +  (diff_Y*diff_Y) + (diff_Z*diff_Z);

        if ( (norm_2 >= rmin_2) && (norm_2 <= rmax_2))
               Packed[index_0++] = norm_2;
    }
}
```

- Perf gain close to 10X going from no-vec to vec
- Index_0 is getting updated under a condition – not linear
    - Currently this cannot be expressed using simd clauses
    - Extensions to simd-syntax to express this idiom is WIP

Optimization Notice

Mandel Ex. for Vectorizing Outer Loop

Original Src:

```
mandel ( x0, x1, y0, y1,
         width, height,
         max_recur, output)
{
 dx = (x1-x0) / width;
 dy = (y1-y0) / height;

 for( j=0; j<height; j++) {
  for( i=0; i<width; i++) {

    index = j * width + i;
    x = x0 + i * dx;
    y = y0 + j * dy;
    std::complex c(x,y);
    mandel_inner(c, max_recur, &output[index]);
  }
 }
}
main() {
  //read inputs
  ...
  mandel(...);
}
```

- Two loops that go over 2D space of points,
- Call mandel_inner for each point.

```
mandel_inner (
    std::complex<float> c,
    int max_recur,
    char *output )
{

 std::complex z = c;
 int i=0;

 while ( i < max_recur ) {
  if ( z.real()*z.real()+z.imag()*z.imag() > 4.0 )
   break;
  z = z * z + c;
  i++;
 }
 *output = i * (255.0/max_recur);
}
```

- For a given point, calculate how many iterations is needed for that point to go out of bounds.

Mandel Version 2: Elemental Function

```
mandel ( x0, x1, y0, y1,
           width, height,
           max_recur, output)  {
  dx = (x1-x0) / width;
  dy = (y1-y0) / height;

#pragma omp parallel for
 for( j=0; j<height; j++) {
#pragma simd
   for( i=0; i<width; i++) {
     index = j * width + i;
     x = x0 + i * dx;
     y = y0 + j * dy;
     mandel_inner(x,y, max_recur,
      &output[index]);

   }
  }
}
```

- Vectorization of the loop is simple
- The function to be called is already vectorized.

```
__declspec(vector(uniform(max_recur),
     linear(output)))
mandel_inner ( float c_re, float c_im
     int max_recur,
     int * output ) {

 float z_re=c_re, z_im=c_im;
 int i=0;

 while ( i < max_recur ) {
  if ( z_re*z_re+z_im*z_im > 4.0 )
   break;
  tmp = z_re*z_re - z_im*z_im + c_re;
  z_im = 2*z_re*z_im + c_im;
  z_re = tmp;
  i++;
 }

 *output = i * (255.0/max_recur);
}
```

- Declared as a “SIMD enabled function".
- Function is vectorized
- SIMD Loop is vectorized

Mandel Version 3: Array Notation

```
#define VLEN 16  // on MIC

mandel ( x0, x1, y0, y1,
          width, height,
          max_recur, output)  {
 dx = (x1-x0) / width;
 dy = (y1-y0) / height;

 #pragma omp parallel for
 for( j=0; j<height; j++) {
  for( i=0; i<width; i+=VLEN) {
   unsigned int ix[VLEN];
   complex<float> c;
   index = j * width + i;
   ix = __sec_implicit_index(0);
   c[:].re = x0 + (i+ix[:]) * dx;
   c[:].im = y0 + j * dy;
   mandel_inner(c, max_recur,
    &output[index]);
  }
 }
}
```

```
void mandel_inner(struct complex<float> c[VLEN],
      unsigned int max_recur,
      unsigned int output[VLEN]) {
 unsigned int i = 0;  complex<float> z[VLEN];
 int mask[VLEN], result[VLEN];
 result[:] = 0;
 z[:].re = c[:].re;
 z[:].im = c[:].im;
 while (i < max_recurrences) {
  float absq[VLEN];
  absq[:] = z[:].re * z[:].re + z[:].im * z[:].im;
  mask[:] = absq[:] < 4.0;
  if (__sec_reduce_all_zero(mask[:]))
   break;
  result[:] += mask[:];
  float oldz_re[VLEN];
  oldz_re[:] = z[:].re;
  z[:].re = (z[:].re * z[:].re) -
           (z[:].im * z[:].im) + c[0:VLEN].re;
  z[:].im = (oldz_re[:] * z[:].im * TWO) +
           c[0:VLEN].im;
  i++;
 }
 output[0:VLEN] = (result[:]*(255.0/max_recur));
}
```

- Some extra effort by the programmer guarantees vectorization
- Direct translation into vector code

Predicate Optimization

- Hoist affine and invariant conditions

```
do i = 1, n
   if (i>=m)
     S1
   end if
  S2
  end do
```

```
do i = 1, m-1
      S2
enddo
do i= m,n
  S1 S2
enddo
```

Predicate opt example

```
for (i=i1; i<=i2; i++) {
  if (i == val)
    sum++;
  *cnt_ptr=*cnt_ptr*5 +1;
}
```

```
/* Region 1 */
cnt1 = min(i2-i1+1, val-i1);
for (i=1; i<=cnt1; i++) {
  *cnt_ptr = *cnt_ptr*5 + 1;
}
/* Region 2 */
If ( (i1<=val) && (val<=i2) {
   sum++;
  *cnt_ptr = *cnt_ptr*5 + 1;
}
/* Region 3 */
cnt2 = i2-val;
if (val < i1)
  cnt2=i2-i1+1;
for (i=1; i<=cnt2; i++) {
  *cnt_ptr = *cnt_ptr*5 + 1;
}
```

MCDRAM Support in FORTRAN

- !DIR$ attributes fastmem :: *data-object*
  - Says put that data object in KNL “fast memory” aka MCDRAM
  - *data-object* is an allocatable array on heap memory–any type, any shape {up to size limits of HBW memory)
- Compiler will generate calls to routines like hbw_posix_memalign and hbw_free to allocate and free “fast memory”
- Full support available starting with 15.0 Product Update 1

FORTRAN fastmem Example

```
 program main
 Real(8), allocatable, dimension(:,:) :: A, B, C
!DIR$ ATTRIBUTES FASTMEM :: A, B, C
 Integer, parameter :: N=600
 Allocate (A(N,N), B(N,N), C(N,N))
 call test(a,b,c,N)
 print *, a(1,1), b(2,2), c(3,3)
 end

 subroutine test(a,b,c,N)
 integer len, i,j
 Real(8), dimension(N,N) :: A, B, C
 Integer:: N
 Print *, 'start'
 call mic_sub(A, N)
 call mic_sub(B, N)
 call mic_sub(C, N)
 end subroutine test
```

```
subroutine mic_sub(a,len)
real(8) a(len,len)
integer i ,len
do i = 1, len
  do j =1, len
    a(i,j) = 2*(i+j)
  enddo
enddo
end
```

FORTRAN fastmem Example2

```
 module work_array
     TYPE FBLOCK
       INTEGER      :: NXFULL
        REAL, ALLOCATABLE :: work1  (:,:)
        REAL, ALLOCATABLE :: work2  (:,:)
!!!DIR$ ATTRIBUTES FASTMEM :: work2
!DIR$ ATTRIBUTES FASTMEM :: work1
       END TYPE FBLOCK

        integer N1
        TYPE(FBLOCK)    :: PV1
      end module
```

```
program main
  use work_array
  read (input, *) N1
  PV1%NXFULL = N1
  allocate(PV1%work1(N1, 2*N1))
  PV1%work1 = 2

  allocate(PV1%work2(N1, 2*N1))
  PV1%work2 = 3

  do j=1,2*PV1%NXFULL
    do i=1,PV1%NXFULL
      print *, PV1%work1(i, j)
      print *, PV1%work2(i, j)
    enddo
  enddo
  end program
```

Review Sheet for Efficient Vectorization

- Are you using vector-friendly options such as –align array64byte?
- Are all hot loops vectorized and maximizing use of unit-stride accesses?
  - Have you looked into outer-loop vs. inner-loop vec tradeoffs?
- Align the data and Tell the compiler
- Have you studied the opt-report output for hot-loops to ensure these?
- Are there any peel-loop and remainder-loop generated for your key-loops (Have you added loop_count pragma)?
  - Make changes to ensure significant runtime is not being spent in such loops
- Are you able to pad your arrays and get improved performance with –opt-assume-safe-padding (only on KNC)?
- Have you added "#pragma vector aligned nontemporal" for all loops with streaming-store accesses to maximize performance?
- Avoid branchy code inside loops to improve vector-efficiency
  - Avoid duplicates between then and else, use builtin_expect to provide hint, move loop-invariant loads and stores under the branch to outside loops
- Use hardware supported operations only (rest will be emulated)

Review Sheet for Vectorization 2

- Use OMP4.0 or Intel Cilk Plus extensions for efficient and predictable vectorization
  - #pragma omp simd OR #pragma SIMD OR !DEC$ SIMD
  - Short-vector array notation for C/C++
    - Shifts burden to the user to express explicit vectorization
    - High-level and portable alternative to using intrinsics
  - Use simd-enabled functions (C and Fortran) for loops with function calls
    - Can also be used to express outer-loop vectorization
    - #pragma omp declare simd

- Study opportunities for outer-loop vectorization based on code access patterns
  - Use array-notations OR simd-enabled-functions to express it

- Make memory accesses unit-strided in vector-loops as much as possible
  - Important for C and Fortran

- F90 array notation also can be used in short-vector form

Optimization Notice

Review Sheet for Advanced Optimizations

- Are you able to take advantage of –fp-model fast=2?
  - Enables –complex-limited-range – important if using “complex” datatype
  - Enables –fimf-domain-exclusion=15 (significant perf adv on KNC)
- If your algorithm allows it, have you tried more aggressive floating-point options:
  - -fimf-precision=low, -no-prec-div, -no-prec-sqrt, -fast-transcendentals, …
- If your application requires use of –fp-model precise:
  - In some cases, users (who want high-performing vector code) may use:
  - –fp-model precise –fimf-max-error=1 –fast-transcendentals –no-prec-div –no-prec-sqrt
  - This will result in generating vectorized code with as much precision as SVML supports – not IEEE, results may be non-reproducible between –O0 and –O2
- Have you tried prefetch tuning options?
  - For indirect accesses, have you tried prefetching using pragmas?
- Have you maximized use of streaming stores for bandwidth savings?

Optimization Notice

Fortran Vectorization Tips

- https://software.intel.com/en-us/articles/fortran-array-data-and-arguments-and-vectorization
- Use CONTIGUOUS Attribute for pointers and assumed shape arrays
- Use –align array64byte option
- Use –opt-assume-safe-padding (KNC only) wherever possible to change gathers into unit-strided loads

Optimization Notice

Reference Links

- http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture - MIC Compiler tips, lots of useful information for Xeon as well
- http://software.intel.com/en-us/mic-developer - Intel(R) Xeon Phi(TM) page
- https://software.intel.com/en-us/intel-isa-extensions - Intel ISA Extensions
- https://software.intel.com/en-us/blogs/2013/avx-512-instructions - AVX-512 instructions
- http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf - OpenMP 4.0 Application Program Interface

Legal Disclaimer & Optimization Notice

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®

15.0 Key Features: Big Items

- Support for majority of OpenMP* 4.0
  - Cancel, cancellation point, depend, combined offload constructs, workshare parallelization
  - Major item not included: user-defined reductions
- Feature Complete! C++ 11
  - Language features only, library features dependent on the standard C++ library with the platform
- Feature Complete! Fortran 2003
- Fortran 2008 Blocks
- Redesign of compiler Optimization Reports
  - including vec-report, loop optimizations, and inlining reports
- -ansi-alias option enabled as part of default level –O2 (Linux)
- Initial implementation of C++ offload to Intel® Graphics Technology
- New icl/icl++ compilers on OS X for improved compatibility with clang/LLVM

Vec/Par/Loop

- Loop info embedded in ASM/OBJ with –g/-Zi
  - Embedded information is a subset of the output available from the text report
  - Executable size increased slightly (~2%)

Amber loop on KNC example

```
for (int i = 0; i < size; ++i) {
    for (int j = i + 1; j < size; ++j) {
        xij = xi - data[3 * j];
        yij = yi - data[3 * j + 1];
        zij = zi - data[3 * j + 2];
```

```
..L50:
  vgatherdpd 32+data(%rbx,%zmm1,8), %zmm3{%k2}
  jkzd       ..L49, %k2
  vgatherdpd 32+data(%rbx,%zmm1,8), %zmm3{%k2}
  jknzd      ..L50, %k2
..L49:
..L52:
  vgatherdpd 24+data(%rbx,%zmm1,8), %zmm2{%k1}
  jkzd       ..L51, %k1
  vgatherdpd 24+data(%rbx,%zmm1,8), %zmm2{%k1}
  jknzd      ..L52, %k1
..L51:
..L54:
  vgatherdpd 40+data(%rbx,%zmm1,8), %zmm7{%k3}
  jkzd       ..L53, %k3
  vgatherdpd 40+data(%rbx,%zmm1,8), %zmm7{%k3}
  jknzd      ..L54, %k3
..L53:
```

- Vectorized with vector length 8
- Code generated without opt: 3 gather-loops load 3*64 bytes of adjacent data

Optimized KNC Sequence: Data Load

```
for (int i = 0; i < size; ++i) {
    for (int j = i + 1; j < size; ++j) {
        xij = xi - data[3 * j];
        yij = yi - data[3 * j + 1];
        zij = zi - data[3 * j + 2];
```

```
vloadunpacklpd 24+data(,%r15,8), %zmm9
vloadunpacklpd 88+data(,%r15,8), %zmm12
vloadunpacklpd 152+data(,%r15,8), %zmm8
vloadunpackhpd 88+data(,%r15,8), %zmm9
vloadunpackhpd 152+data(,%r15,8), %zmm12
vloadunpackhpd 216+data(,%r15,8), %zmm8
```

- Start address is not aligned
- A pair of loadunpacks is required for each 64-byte chunk load

8	7	6	5	4	3	2	1	t1
16	15	14	13	12	11	10	9	t2
24	23	22	21	20	19	18	17	t3

Full KNC sequence: Summary

- 3 pairs of loadunpacklpd/loadunpackhpd
- 6 cross-lane permutations
- 5 blends
- 2 in-lane shuffles or swizzles
- From 30% to 48% speed-up on KNC B0 for size equal to 10000

Review Sheet (Partial) for Efficient OMP Parallelization

- Is the algorithm able to take advantage of all available threads?
  - In some cases, using OMP collapse may help – but make sure innermost loop gets efficiently vectorized after collapsing
    - May need stripmine loop to make sure innermost loop is not part of collapse
- Reduce any use of barrier synchronization, OMP locks, critical sections
  - In some cases, using nowait clause may help (review example in BKM page)
  - Use reductions where possible
- Are the OMP affinity settings right to cause no oversubscription?
  - Different defaults for native vs. offload
  - For offload, use 'norespect' to use all N*4 threads
  - If you are using <4 threads/core, use –opt-threads-per-core=n
- Affinity tuning
  - Balanced vs. Compact vs. Scatter vs. proclist, Core vs. Fine
- Try different OMP loop scheduling types – static vs. dynamic(<n>)
- Have you tuned your code (to overcome load-imbalance) with KMP_BLOCKTIME=<default,0,50,infinite>
- Explicit tuning of OMP_STACKSIZE?

Intel® Compilers for Intel® Parallel Studio XE 2015

Intel® C++ 15.0 and Intel® Fortran 15.0

Productive language-level vectorization & parallelism models for advanced developers driving application performance

- Common to both
  - New OpenMP 4.0 vectorization simplifies taking advantage of SIMD instructions for great performance on Intel® Xeon® and Xeon Phi™ processors and coprocessors
  - Improved compiler optimization reports help quickly identify optimization opportunities. For Windows-based developers, Visual Studio* 2010, 2012 and 2013 integration is included.
  - Linux*, OS X*, Windows*, Android*
  - Available now in a variety of configurations to suit different development needs. C++ Info Fortran Info
- Intel® C++ Compiler
  - Intel Cilk™ Plus keywords for parallelism simplify implementation of task and data parallelism
  - Complete C++11 support
- Intel® Fortran Compiler
  - Support for the latest Fortran standards
  - Rogue Wave* IMSL* Fortran Numerical Libraries: Performance add-on for Intel® Fortran Windows suites

OpenMP* 4.0 Support

Intel Compilers

- Everything now supported in Intel C++ and Fortran compilers, except user-defined reductions
- CANCEL directive: Requests cancellation of the innermost enclosing region
- CANCELLATION POINT directive: Defines a point at which implicit or explicit tasks check to see if cancellation has been requested
- DEPEND clause on TASK directive: Enforces additional constraints on the scheduling of a task by enabling dependences between sibling tasks in the task region.
- Combined constructs (TEAMS DISTRIBUTE, etc.)

What's New
Intel® C++ Compiler

- Excellent outer-loop optimization capabilities with either OpenMP* or Intel Cilk™ Plus
- Intel® Cilk™ Plus : explicit vectorization. Keyword versions of SIMD pragmas added: _Simd, _Safelen, _Reduction
- Near complete OpenMP* 4.0, including TASK dependency to speed performance by enforcing task scheduling
- Easier to use, more insightful optimization reports, including vectorization, all consolidated under -qopt-report
- Full C++11 language support
- Gcc*-compatible function multiversioning
- Compiler details
  - -ansi-alias enabled by default at –O2 and above on Linux* C++ to enable better performance, including vectorization (matches –fstrict-aliasing defaults on gcc*)
  - Compiler option –no-opt-dynamic-align to disable generation of multiple code paths depending on data alignment
  - aligned_new header
  - -fast/-Ofast enables –fp-model fast=2
- Improved lambda function debugging

GNU Compatibility

Intel® C++ Compiler

- To enable c++11 support you need to use –std=c++11 (or –std=c++0x) option
- We currently support all c++11 features used in the GNU 4.8 versions of the headers enabled when you use the option
- Depending upon the GNU on your system (i.e. g++ in your PATH) you may get different features enabled
- Support of C++11 features requires support from C++ header files included with GNU C/C++ installation – these features vary by version.
- Recommend use of GNU 4.8 or newer packages

Easier Task & Data Parallelism with Intel® Cilk™ Plus (Intel® C++ Compiler)

- Easier task & data parallelism with three simple keywords :
  - Cilk_for, Cilk_spawn, Cilk_sync

- Save time in implementing vectorization. Use Intel® Cilk™ Plus Array Notation and #pragma SIMD

Serial code (left) made parallel with Intel® Cilk™ Plus keywords.
No changes to original code.

Serial	Parallel

int fib (int n)
{
if (n <= 2)
return n;
else {
int x,y;
x =
fib(n-1);
y =
fib(n-2);
return
x+y;
}
}
	int fib (int n)
	{
	if (n <= 2)
	return n;
	else {
	int x,y;
	x = _Cilk_spawn fib(n-
	1);
	y = fib(n-2);
	_Cilk_sync;
	return x+y;
	}
	}

Array notation showing simple vector multiplication

```
A[0:N] = b[0:N] * c[0:N];
```

More sophisticated example of array notation

```
X[0:10:10] = sin(y[20:10:2]);
```

Code snippet before #pragma SIMD

```
for(int i = 2; i < n ;i++)
    y[i] = y[i-2] + 1;
```

Code snippet with #pragma SIMD

```
#pragma simd vectorlength(2)
for(int i = 2; i < n ;i++)
y[i] = y[i-2] + 1;
```

Intel® Cilk™ Plus and OpenMP* 4.0 Differences

Intel® C++ Compiler

Intel® Cilk™ Plus	OpenMP* 4.0
Array Notations Support	No Array Notations support
Support for User implemented vector function using _declspec(vector_variant)implement)…))) This feature enables the users to implement the vector variant of the SIMD enabled function if they aren't happy with the vector code generated by the compiler.	No Support for User implemented vector functions.
#pragma simd has firstprivate(), vecremainder and assert clauses which "omp simd" doesn't support. Can mimic the behavior of aligned() clause in OMP4.0 with __assume_aligned() or __builtin_assume_aligned()	"omp simd" has collapse() and aligned() clauses which is not supported by #pragma simd
_Simd keyword support for explicit vectorization apart from #pragma simd. This keyword support was enabled for enabling threading and vectorization for a single "for" loop which OpenMP4.0 could do with the following: #pragma omp parallel for simd	No _Simd keyword support for explicit vectorization.
Support for __intel_simd_lane() to identify the simd lane on which the current operation is happening.	No support for identifying the individual simd lane.
Supports built-in reduction operations like __sec_reduce_add(), __sec_reduce_max_ind(), __sec_reduce_mul(), __sec_reduce_min_ind(), __sec_reduce_all_zero(), __sec_reduce_all_nonzero(), __sec_reduce_min(), __sec_reduce_max(), __sec_reduce_and(), __sec_reduce_or(), __sec_reduce_xor()	No support for built-in reducer operations.
Supports writing custom reduction functions using: __sec_reduce() and __sec_reduce_mutating()	No Support for custom reduction functions.
Support for Array implicit index using __sec_implicit_index()	No support for Array Implicit index.

What's New
Intel® Fortran Compiler

- Improvements in both automatic and explicit (using Intel or OpenMP directives) vectorization, especially in outer loop vectorization. Provides improved robustness and performance.
- Full Fortran 2003 support including Parameterized Derived Types
- Intel® Fortran supports Fortran 2008 Blocks and much more from Fortran 2008
- Near complete OpenMP* 4.0, including task dependencies. What's left? User-defined reductions.
- Compiler option –no-opt-dynamic-align to ensure run-to-run reproducibility with relatively little impact on performance (compared to –fp-model precise)
- Fortran option –init=snan to initialize all uninitialized SAVEd scalar and array variables of type REAL and COMPLEX to signaling NaNs
- __intel_simd_lane() intrinsic to represent simd lane number in a SIMD vector function callable from Fortran using the interoperability feature
- Support offload of arrays of pointers and non-contiguous array slices (to Intel® Xeon Phi™ coprocessors)
- gdb* debugger supports Intel Fortran (Intel® Debugger removed)
- -fast/-Ofast enables –fp-model fast=2
- Fastmem support for KNL MCDRAM

Fortran OpenMP* Support: WORKSHARE

Intel® Fortran Compiler

- Can go parallel in many uses
- Simple array assignments such as A = B + C parallelize.
- Simple array assignments with overlap such as A = A + B + C parallelize.
- Array assignments with user-defined function calls parallelize such as A = A + F (B). F must be ELEMENTAL.
- Array assignments with array slices on the right hand side of the assignment such as A = A + B(1:4) + C(1:4) parallelize. If the lower bound of the left hand side or the array slice lower bound or the array slice stride on the right hand side is not 1, then the statement does not parallelize.

Intel® Fortran Compiler BLOCK Examples

BLOCK Example

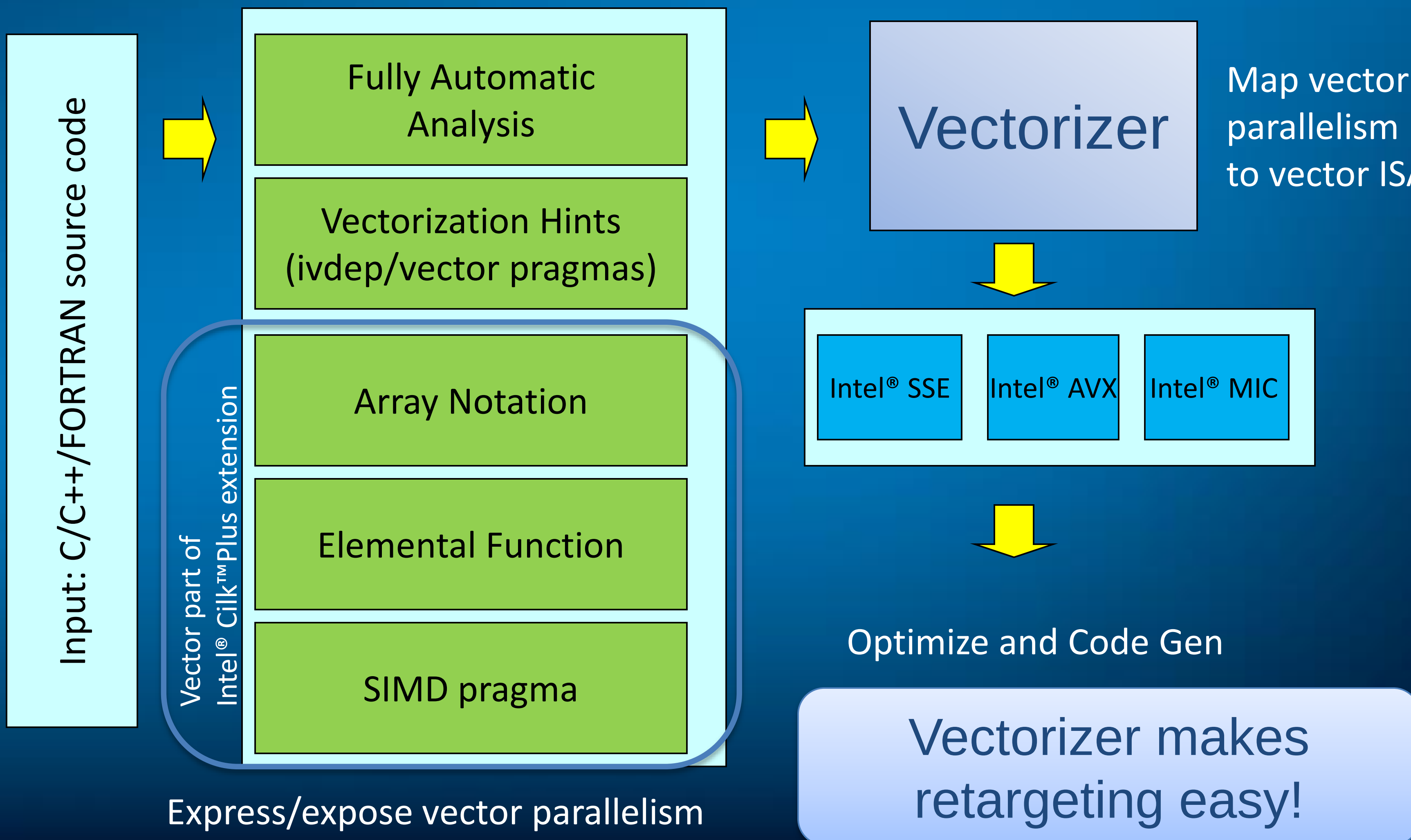
```
IF (swapxy) THEN
  BLOCK
    REAL(KIND(x)) tmp
    tmp = x
    x = y
    y = tmp
  END BLOCK
END IF
```

F08: DO CONCURRENT with BLOCK

```
DO CONCURRENT (I = 1:N)
  BLOCK
    REAL T
    T = A(I) + B(I)
    C(I) = T + SQRT(T)
  END BLOCK
END DO
```

Without BLOCK, no way to create an iteration-local (threadprivate) temporary variable

Vectorizer Architecture



Improved Vectorization

Intel Compilers

- Guaranteed vectorization for entire SIMD loop and SIMD-enabled function while isolating offending code
- Apply partial vectorization escape
  - Serialize execution of offending code section (usually small portion)
  - Works at both statement and expression level
  - Currently enabled for SIMD loops and SIMD-enabled (vector) functions
- Significantly reduced SIMD vectorization bail-outs due to cryptic reasons, e.g. “statement cannot be vectorized”, “operation cannot be vectorized”, “unsupported data type”, etc.
- Customers more satisfied with smaller number of SIMD vectorization bail-outs

Cilk Plus Improvements & OpenMP 4.0 SIMD Support

```
#pragma omp simd aligned(a,c)
for (int i; i <N; i++)  {
   a[i] = b[i]*c[i];
}
```

OpenMP 4.0 syntax support

- For both loops and functions
- Stricter syntax checking
- Alignment specification supported
- Safelen semantics

New public ABI support for vector functions

Support for private/lastprivate arrays and structs

- SOA re-layout for non-escaping local private structs and arrays

```
#pragma simd
for (int i; i <N; i++)  {
   float3 q;  // SIMD private
    q.f1 = a[i] + b[i];
    q.f2 = a[i] - b[i];
}
```

- q.f1 is unit-stride access, not stride 3*unit
- Same done for local arrays

VL agreement rules for caller and callee relaxe

- Longer VL caller may call shorter VL callee [multiple

Updates to MIC Vectorization

- Support for compress/expand idiom
  - Using packstore/loadunpack on KNC
  - Using vcompress/vexpand on KNL
- Improvements to remainder and low trip count masked vectorization
  - Non-masked code path for full mask (if needed)
  - -opt-assume-safe-padding option on KNC to mitigate vector load using gather issue
- Masks handling improvements
- Optimized math functions with controllable precision

```
for (int i; i <N; i++)  {
   if (a[i] > 0)   {
      b[j++]  = a[i];   //
compress
      c[i]  = a[k++];  // expand
   }
}
```

- Cross-iteration dependencies by j and k

Reference Links - 2

- http://software.intel.com/en-us/intro-to-vectorization-using-intel-cilk-plus - Cilk Plus webinar
- http://software.intel.com/en-us/code-samples/intel-c-compiler/ - Intel C++ Compiler code samples
- http://software.intel.com/en-us/articles/getting-started-with-intel-cilk-plus-array-notations/#! – Intel® Cilk™ Plus Array Notation
- http://software.intel.com/en-us/articles/intelr-cilktm-plus-white-paper#! – Intel® Cilk™ Plus White Paper
- http://software.intel.com/en-us/articles/implementing-sepia-filters-with-intelr-ciltkm-plus#! – Improving Sepia filter performance with Intel® Cilk™ Plus
- http://software.intel.com/en-us/articles/improving-averaging-filter-performance-using-intel-cilk-plus#! – Improving Averaging filter performance with Intel® Cilk™ Plus

Reference Links - 3

- http://software.intel.com/en-us/articles/improving-discrete-cosine-transform-performance-using-intelr-cilktm-plus#! – Improving DCT kernel performance using Intel® Cilk™ Plus
- http://software.intel.com/en-us/articles/call-site-dependence-for-elemental-functions-simd-enabled-functions-in-c#! – SIMD-enabled functions explained (call site dependence)
- http://software.intel.com/en-us/articles/usage-of-linear-and-uniform-clause-in-elemental-function-simd-enabled-function-clause#! – SIMD-enabled functions with clauses explained
- http://software.intel.com/en-us/blogs/2013/06/07/resources-about-intel-transactional-synchronization-extensions - Intel(R) TSX

Reference Links - 4

- http://halobates.de/adding-lock-elision-to-linux.pdf - Intel(R) TSX presentation by Andi Kleen
- http://software.intel.com/en-us/intel-isa-extensions - Intel(R) MPX page
- http://software.intel.com/en-us/c-compiler-android - Intel C++ Compiler for Android
- http://software.intel.com/en-us/articles/intel-compilers-for-linux-compatibility-with-gnu-compilers-0 - GNU Compiler Compatibility

Compiler Based Vectorization

Extension Specification

Feature	**SIMD Extension**
Intel® Streaming SIMD Extensions 2 (Intel® SSE2) as available in initial Pentium® 4 or compatible non-Intel processors	sse2
Intel® Streaming SIMD Extensions 3 (Intel® SSE3) as available in Pentium® 4 or compatible non-Intel processors	sse3
Supplemental Streaming SIMD Extensions 3 (SSSE3) as available in Intel® Core™2 Duo processors	ssse3
Intel® SSE4.1 as first introduced in Intel® 45nm Hi-K next generation Intel Core™ micro-architecture	sse4.1
Intel® SSE4.2 Accelerated String and Text Processing instructions supported first by Intel® Core™ i7 processors	sse4.2
Like ssse3 but optimizes for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology	ssse3_atom
Intel® Advanced Vector Extensions (Intel® AVX) as available in 2nd generation Intel® Core™ processor family	avx
Intel® Advanced Vector Extension (Intel® AVX) including instructions offered by the 3rd generation Intel® Core processor	core-avx-I
Intel® Advanced Vector Extension 2 (Intel® AVX2) as provided by a 4th Generation Intel® Core Processors and above	core-avx2

Loops with Lots of Memory Accesses

- In some cases, you can do careful allocation of arrays or choose the extents to add some padding to reduce associativity issues:
- If you have a loop access of the form:

```
#  define FD_REPEAT(x,idx,coeff) +coeff[0] * x##_4[(idx)] \
                    +coeff[1] * (x##_3[(idx)] + x##_5[(idx)]) \
                    +coeff[2] * (x##_2[(idx)] + x##_6[(idx)]) \
                    +coeff[3] * (x##_1[(idx)] + x##_7[(idx)]) \
                    +coeff[4] * (x##_0[(idx)] + x##_8[(idx)])
for (j=0; j<STREAM_ARRAY_SIZE; j++)
          a[j] = scalar*c[j]
           FD_REPEAT(b_ext,j,vscalar);
```


Optimization Notice

Loops with Lots of Memory Accesses-2

And the data allocation was done as follows:

```
a = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
c = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_0 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_1 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_2 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_3 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_4 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_5 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_6 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_7 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_8 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
```

Loops with Lots of Memory Accesses-2

Try this instead:

```
a = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
c = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_0 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
// Shift the start-point of b_ext_* to reduce set-associativity problems.
// Make sure added value is a multiple of 16 (to keep the 64-byte alignment for base-ptr
b_ext_0 = b_ext_0 + 16;
b_ext_1 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_1 = b_ext_1 + 16;
b_ext_2 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_2 = b_ext_2 + 32;
b_ext_3 = _mm_malloc( (STREAM_ARRAY_SIZE) * sizeof(STREAM_TYPE), 64);
b_ext_3 = b_ext_3 + 32;
 …
```

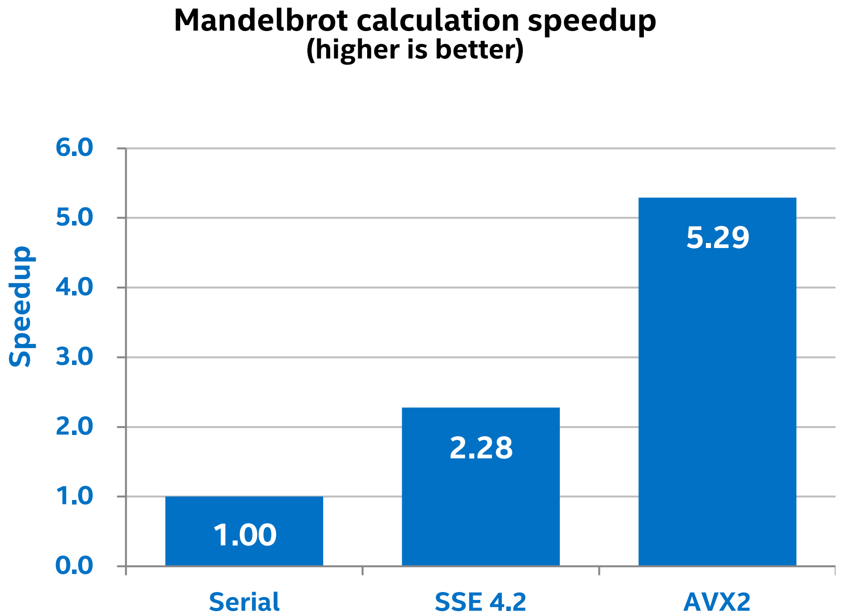

Optimization Notice

Impressive Performance Improvement

Intel® Compiler OpenMP* 4.0 Explicit Vectorization

- Two lines added that take full advantage of both SSE or AVX
- Pragmas ignored by other compilers so code is portable

```
typedef float complex fcomplex;
 const uint32_t max_iter = 3000;
#pragma omp declare simd uniform(max_iter), simdlen(16)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{
    uint32_t count = 1; fcomplex z = c;
    while ((cabsf(z) < 2.0f) && (count < max_iter)) {
       z = z * z + c; count++;
    }
    return count;
}
uint32_t count[ImageWidth][ImageHeight];
 …… ….   …….
      for (int32_t y = 0; y < ImageHeight; ++y) {
        float c_im = max_imag - y * imag_factor;
#pragma omp simd safelen(16)
        for (int32_t x = 0; x < ImageWidth; ++x) {
          fcomplex in_vals_tmp = (min_real + x * real_factor) +
(c_im * 1.0iF);
          count[y][x] = mandel(in_vals_tmp, max_iter);
        }
      }
```



Normalized performance data – higher is better

Configuration: Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz system (4 cores with Hyper-Threading On), running at 3.50GHz, with 32.0GB RAM, L1 Cache 256KB, L2 Cache 1.0MB, L3 Cache 8.0MB, 64-bit Windows* Server 2012 R2 Datacenter. Compiler options:, SSE4.2: -O3 –Qipo –QxSSE4.2 or AVX2: -O3 –Qipo –QxCORE-AVX2.

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. * Other brands and names are the property of their respective owners. Benchmark Source: Intel Corporation

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.

Impressive Performance Improvement

Intel® Compiler OpenMP* 4.0 Explicit Vectorization

- Three lines added that take full advantage of both SSE or AVX
- Pragma's ignored by other compilers so code is portable

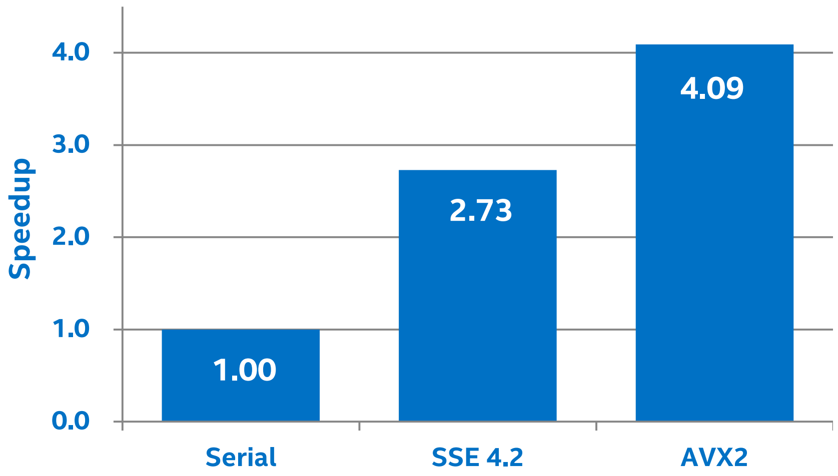
```
#pragma omp declare simd linear(z:40) uniform(L, N, Nmat) linear(k)
float path_calc(float *z, float L[][VLEN], int k, int N, int Nmat)

#pragma omp declare simd uniform(L, N, Nopt, Nmat) linear(k)
float portfolio(float L[][VLEN], int k, int N, int Nopt, int Nmat)
… … …
for (path=0; path<NPATH; path+=VLEN) {
    /* Initialise forward rates */
    z = z0 + path * Nmat;
#pragma omp simd linear(z:Nmat)
   for(int k=0; k < VLEN; k++) {
     for(i=0;i<N;i++) {
        L[i][k] = L0[i];
     }

     /* LIBOR path calculation */
     float temp = path_calc(z, L, k, N, Nmat);
     v[k+path]  = portfolio(L, k, N, Nopt, Nmat);

     /* move pointer to start of next block */
     z += Nmat;
   }
}
```

**Libor calculation speedup
(higher is better)**



Normalized performance data – higher is better

Configuration: Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz system (4 cores with Hyper-Threading On), running at 3.50GHz, with 32.0GB RAM, L1 Cache 256KB, L2 Cache 1.0MB, L3 Cache 8.0MB, 64-bit Windows* Server 2012 R2 Datacenter. Compiler options:, SSE4.2: -O3 –Qipo –QxSSE4.2 or AVX2: -O3 –Qipo –QxCORE-AVX2. For more information go to http://www.intel.com/performance

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. * Other brands and names are the property of their respective owners. Benchmark Source: Intel Corporation

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.

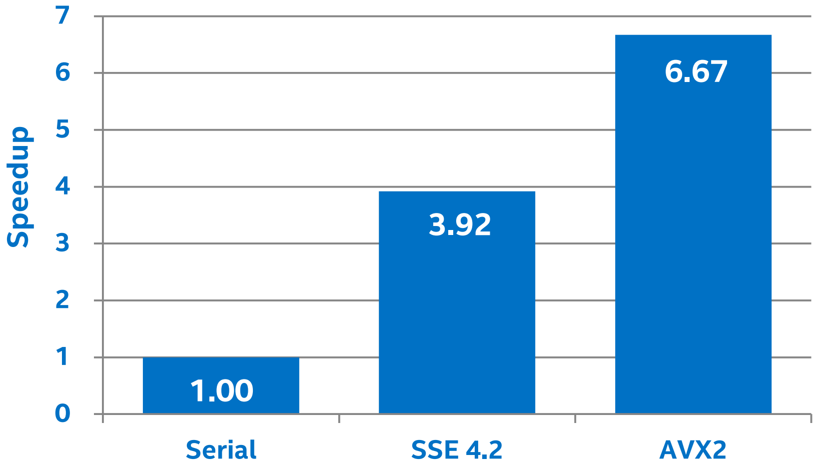
Impressive Performance Improvement

Intel C++ Explicit Vectorization: SIMD Performance

- One line added that take full advantage of both SSE or AVX
- Pragma's ignored by other compilers so code is portable

```
#pragma simd vectorlength(8)
 for (int x = x0; x < x1; ++x) {
     float div = coef[0] * A_cur[x]
                   + coef[1] * ((A_cur[x + 1] + A_cur[x - 1])
                   + (A_cur[x + Nx] + A_cur[x - Nx])
                   + (A_cur[x + Nxy] + A_cur[x - Nxy]))
                   + coef[2] * ((A_cur[x + 2] + A_cur[x - 2])
                   + (A_cur[x + sx2] + A_cur[x - sx2])
                   + (A_cur[x + sxy2] + A_cur[x - sxy2]))
                   + coef[3] * ((A_cur[x + 3] + A_cur[x - 3])
                   + (A_cur[x + sx3] + A_cur[x - sx3])
                   + (A_cur[x + sxy3] + A_cur[x - sxy3]))
                   + coef[4] * ((A_cur[x + 4] + A_cur[x - 4])
                   + (A_cur[x + sx4] + A_cur[x - sx4])
                   + (A_cur[x + sxy4] + A_cur[x - sxy4]));
     A_next[x] = 2 * A_cur[x] - A_next[x] + vsq[s+x] * div;
 }
```

RTM-Stencil Speedup
(higher is better)



Normalized performance data – higher is better

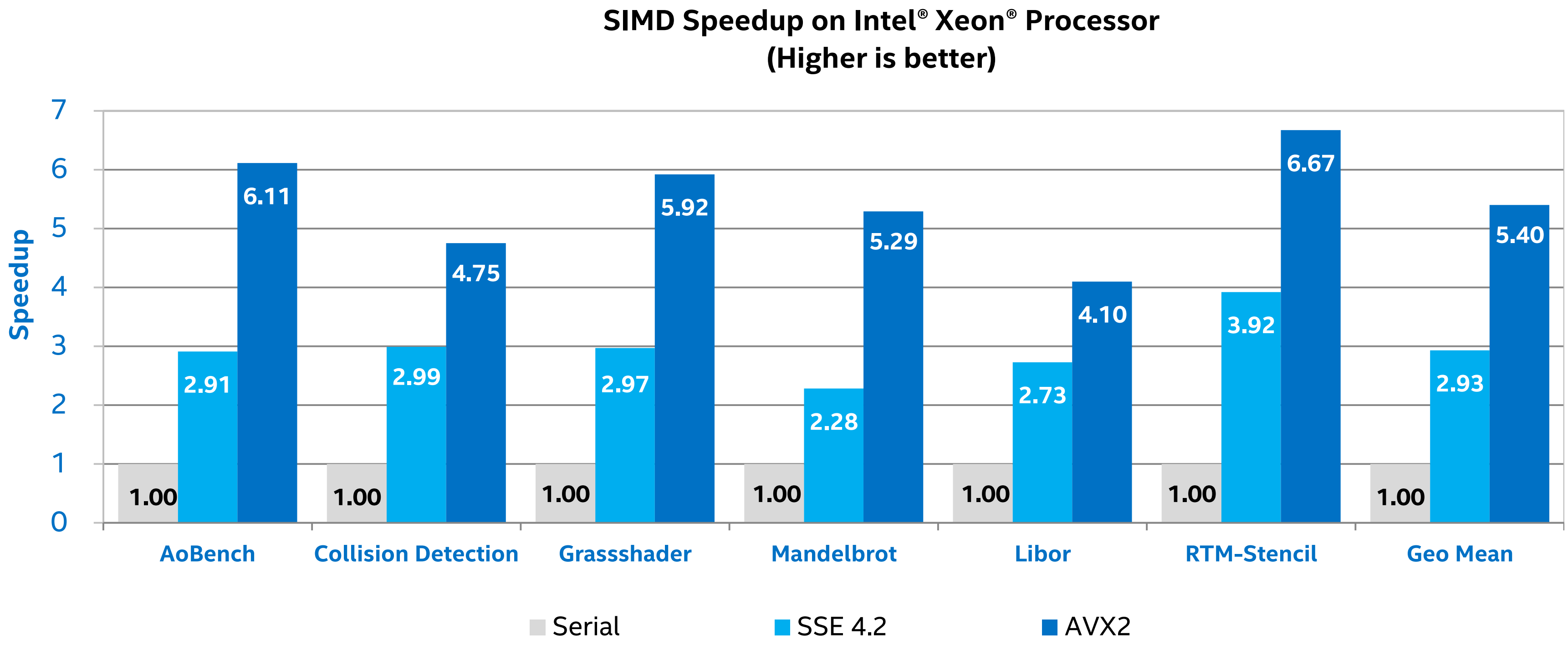
Configuration: Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz system (4 cores with Hyper-Threading On), running at 3.50GHz, with 32.0GB RAM, L1 Cache 256KB, L2 Cache 1.0MB, L3 Cache 8.0MB, 64-bit Windows* Server 2012 R2 Datacenter. Compiler options:, SSE4.2: -O3 –Qipo –QxSSE4.2 or AVX2: -O3 –Qipo –QxCORE-AVX2. For more information go to http://www.intel.com/performance

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.   * Other brands and names are the property of their respective owners.   Benchmark Source: Intel Corporation

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 .

Impressive performance improvement

Intel C++ Explicit Vectorization using OpenMP* 4.0 SIMD or Intel® Cilk™ Plus



Normalized performance data – higher is better

Configuration: Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz system (4 cores with Hyper-Threading On), running at 3.50GHz, with 32.0GB RAM, L1 Cache 256KB, L2 Cache 1.0MB, L3 Cache 8.0MB, 64-bit Windows* Server 2012 R2 Datacenter. Compiler options:, SSE4.2: -O3 –Qipo –QxSSE4.2 or AVX2: -O3 –Qipo –QxCORE-AVX2. For more information go to http://www.intel.com/performance

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. * Other brands and names are the property of their respective owners. Benchmark Source: Intel Corporation

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.