Using Openmp* Effectively on Theta

Carlos Rosales-Fernandez & Ronald W Green
Intel Corporation

2019 ALCF Computational Performance Workshop

Access and getting the files

Find a good working directory. These labs are small and don't create a lot of data. Your /home should suffice, assuming you have not exhausted your quota

To get started, copy the files to a directory of your choosing in the **/projects** area:

$ tar -zxvf /projects/SDL_Workshop/training/UsingOpenMP/labs.tgz

Then change into the **omp** directory:

```
$ cd ./omp
```

Methodology

- Labs are numbered “labX”
- We will work through the labs in numeric order starting with “lab1”
- Each lab has a “readme.txt” to describe the lab
- Each lab has a batch script “ labX.run”
  - If there are multiple runs in a lab, run scripts are named “labX-Y.run”
    - for example, if there are 2 run scripts in lab1, the run scripts are “lab1-1.run” and lab1-2.run”
- Solutions, if needed are in directory “solution/”
- Move through the labs at your own pace OR follow along with the group

Misc

Use latest Intel compiler

```
module swap intel/18.0.0.128 intel/19.0.3.199
```

OpenMP* 5.0 Reference

omp/OpenMPRef-5.0-111802-web.pdf

Getting the most out of your compiler with the Intel Classic Compilers Optimization Report

Ronald W Green, Martyn Corden

Technical Consulting Engineer

Intel Corporation

Objectives

Learn to use the consolidated and enhanced compiler optimization report in Intel Classic Compilers

Control the information provided

Understand what optimizations the compiler performed

Use the information in the report to guide further tuning for improved performance

General

Applicable to Intel® Compiler version 15.0 and newer

- for C, C++ and Fortran
- for Windows*, Linux* and OS X*

  (For readability, options may not be repeated for each OS where spellings are similar. Options apply to all three OS unless otherwise stated.)

Main options (there are a lot of qopt-report-* options):

-qopt-report[=N] (Linux and OS X)

/Qopt-report[:N] (Windows)

N = 1-5 for increasing levels of detail, (default N=2)

-qopt-report-phase=str[,str1,...]

str = loop, par, vec, openmp, ipo, pgo, cg, offload, tcollect, all

-qopt-report-file=[stdout | stderr | filename]

Vectorization – report levels

[-q|/Q]opt-report-phase=vec   [-q|/Q]opt-report=N

N specifies the level of detail; default N=2 if N omitted

Level 0: No vectorization report

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: Additional detail, e.g. on data alignment

Level 5: Adds detailed data dependency information

Report Output

Output goes to a text file by default

- File extension is .optrpt, root name same as object file's
- One report file per object file, in object directory
- created from scratch or overwritten (no appending)

[-q | /Q]opt-report-file:stderr gives to stderr

:filename to change default file name

/Qopt-report-format:vs format for Visual Studio* IDE

For debug builds, (-g on Linux* or OS X*, /Zi on Windows*), assembly code and object files contain loop optimization info

- /Qopt-report-embed to enable this for non-debug builds

Filtering Report Output

The optimization report can be large

Filtering can restrict the content to the most performance-critical parts of an application

[-q | /Q]opt-report-routine:<function1>[,<function2>,…]

“function1” can be a substring of function name
or a regular expression

can also restrict to a particular range of line numbers, e.g.:

icl /Qopt-report-filter=“test.cpp,100-300” test.cpp

ifort –qopt-report-filter=“test.f90,100-300” test.f90

Also select the optimization phase(s) of interest with
-opt-report-phase

Loop, Vectorization and Parallelization Phases

Hierarchical display of loop nest

- Easier to read and understand
- For loops for which the compiler generates multiple versions, each version gets its own set of messages

Where code has been inlined, caller/callee info available

The "Loop" (formerly hlo) phase includes messages about memory and cache optimizations, such as blocking, unrolling and prefetching

- Now integrated with vectorization & parallelization reports

Hierarchically Presented Loop Optimization Report (C/C++)

```
1  double a[1000][1000],b[1000][1000],c[1000][1000];
2
3  void foo() {
4    int i,j,k;
5
6    for( i=0; i<1000; i++) {
7      for( j=0; j< 1000; j++) {
8        c[j][i] = 0.0;
9        for( k=0; k<1000; k++) {
10         c[j][i] = c[j][i] + a[k][i] * b[j][k];
11       }
12     }
13   }
14 }
```

source location

header info

```
LOOP BEGIN at ...\mydir\dev\test.c(7,5)
   Distributed chunk2
   ....
  LOOP BEGIN at ...\mydir\dev\test.c(9,7)
     Distributed chunk2
     ....
    LOOP BEGIN at ...\mydir\dev\test.c(6,3)
       ....
    LOOP END
    LOOP BEGIN at ...\mydir\dev\test.c(6,3)
      ....
    LOOP END
  LOOP END
LOOP END
```

report contents

loop nesting

Hierarchically Presented Loop Optimization Report (Fortran)

```
1 program matrix
2 !...a simple matrix multiply example
3 use iso_fortran_env
4 implicit none
5 integer, parameter :: sp=REAL32
6 integer, parameter :: dp=REAL64
7 integer, parameter :: ROWS=1000,COLS=1000,N=1000 ! square matrix example
8 real (kind=dp) :: a(ROWS,COLS)=2.0_dp, b(ROWS,COLS)=3.0_dp, c(ROWS,COLS)
9 integer :: i, j, k
10
11      c = 0.0_dp
12      do j=1,COLS
13        do i=1,ROWS
14          do k=1,N
15            c(i,j)=c(i,j)+a(i,k)*b(k,j)
16          end do
17        end do
18      end do
19 end program matrix
```

source location

header info

```
LOOP BEGIN at matrix_step0.f90(12,5)
Loopnest Interchanged: ( 1 2 3 ) --> ( 1 3 2 )
   ….
   LOOP BEGIN at matrix_step0.f90(14,9)
   loop was not vectorized: inner loop was vectorized
   …
      LOOP BEGIN at matrix_step0.f90(13,7)
      …
        remark #15301: PERMUTED LOOP WAS VECTORIZED
      …
      LOOP END
   LOOP END
LOOP END
```

report contents

loop nesting

Terminology and Tricks

Compiler Methods to Increase Performance

MULTIVERSIONING

When in doubt, make 2 or more versions of a loop

MULTIVERSION Loops

Consider this:

```
int  foo (   real* array, int n )
...
for ( i=0 ; i < n ; i++){
    ... do some work on array[i] ... }
```

What is the value of 'n'?
I don't know,
nor do you,
nor does the compiler!

What is the value of 'n' assumed by the compiler?

NO ASSUMPTION, could be positive OR negative

Is this worth vectorizing??

MULTIVERSIONING – make 2 or more versions of the loop:

example, 1 serial version, 1 vectorized version

MULTIVERSION Loops

```
#So starting with this:
for ( i=0 ; i < n ; i++){
    … do some work on array[i] … }


# actually create code that would mimic this
(pseudo coded)
if( n > 16 ) {
   # <V1> multiversion loop V1

   #pragma vector always
   for ( i=0 ; i < n ; i++){ … }
} else {

   # <V2> multiverion loop V2

   #pragma novector
   for ( i=0 ; i < n ; i++){ … }
}
```

PEEL, KERNEL, REMAINDER LOOPS

Achieving best data movement

Some Compiler Tricks & Terminology

Consider this:

```
int  foo (   real* array, int n )
#pragma simd vector aligned( array:16 )  // vector length 4
for ( i=0 ; i < n ; i++){
    array[i+1] = array[i+1] +  ... }
```

Fetching array[1], 2, 3, 4 to fill a vector would have to use unaligned loads/stores

Is this worth vectorizing?? Inefficient accesses, maybe not.

PEEL LOOP

#128 bit SSE vectors example
#pragma simd vector aligned(array:16) // vector length 4

```
for ( i=0 ; i < n ; i++){
    array[i+1] = array[i+1] +  … }
```

Address mod 16 = 0

Accesses start here

MAIN MEMORY

array[0]	array[1]	array[2]	array[3]
array[4]	array[5]	array[6]	array[7]
array[8]	array[9]	array[10]	Array[11]
…			
array[n-2]	array[n-1]		

CACHE LINES

array[0]	array[1]	array[2]	array[3]
array[4]	array[5]	array[6]	array[7]
array[8]	array[9]	array[10]	Array[11]
…			
array[n-2]	array[n-1]	unused	unused

PEEL LOOP

PEEL LOOP – do the first 3 iterations with unaligned loads/store. THEN starting with element 4 (aligned on 16 byte boundary) switch to aligned loads/stores.

Bonus points: how do you deal with addresses array[i+offset]?

PEEL – do a loop 3 iterations to do theses 3 element
Use unaligned loads/stores. **PEEL LOOP**

array[0]	array[1]	array[2]	array[3]
array[4]	array[5]	array[6]	array[7]
array[8]	array[9]	array[10]	Array[11]
…			[n-3]
array[n-2]	array[n-1]		

THEN another loop
Do these elements
Aligned load/stores
KERNEL LOOP

And a loop for these 2 elements
Unaligned load/stores
REMAINDER LOOP

Kernel and Remainder Loops

KERNEL LOOP – core of the loop done with 'best possible' vectorization

OR what if the # elements is not a multiple of the vector length?

```
real array[103]  ;
#pragma simd vector aligned( array:16 ) // again, 4 elements per vector
for ( i=0; i<103 ; i++ ) {
   array[i] = .... }
```

REMAINER LOOP – do elements 0..99 in chunks (vectors) of 4 elements, then branch to a serial loop with 3 iterations to "clean up"

Some Compiler Tricks & Terminology

Extra bonus points: what about this?

```
#pragma simd vector aligned( a, b:16 , c:16, d)
for ( i=1 ; i < n -2; i++){
   a[i] = 1.0/3.0 * (c[i-1] + a[i] + d[i+1]) + b[i];  }
```

Question: how do you get alignment here?

Answer – you can't do all of the loads/stores the same

- try to find 'best case' where MOST of the loads/stores are aligned (peel on [i] to get those aligned.
  - Implies c and d will be unaligned loads/stores

Peel loop, remainder loop and kernel

```
LOOP BEGIN at ggFineSpectrum.cc(124,5) inlined into ggFineSpectrum.cc(56,7)
   remark #15018: loop was not vectorized: not inner loop

   LOOP BEGIN at ggFineSpectrum.cc(138,5) inlined into ggFineSpectrum.cc(60,15)
   Peeled
      remark #25460: Loop was not optimized
   LOOP END

   LOOP BEGIN at ggFineSpectrum.cc(138,5) inlined into ggFineSpectrum.cc(60,15)
      remark #15145: vectorization support: unroll factor set to 4
      remark #15002: LOOP WAS VECTORIZED
   LOOP END

   LOOP BEGIN at ggFineSpectrum.cc(138,5) inlined into ggFineSpectrum.cc(60,15)
   Remainder
       remark #15003: REMAINDER LOOP WAS VECTORIZED
   LOOP END
LOOP END
```

Vectorized with Peeling and Remainder

MULTIVERSIONED Loops with Peels, kernels, remainers

Compiler uses both multiversioning and peel/kernel/remainder loops

```
# actually create code that would mimic this (pseudo coded)

if( n > 16 ) {
   # <V1> multiversion loop V1

   # <V1> PEEL loop
     for ( i=0 ; i < 4 ; i++){ … } #pragma novector

   # <V1> KERNEL loop
    for ( i=4 ; i < n-3 ; i++){ … } #pragma vector always

   # <V1> REMAINDER loop
   for ( i=n-2 ; i < n ; i++){ … } #pragma novector
} else {

    # <V2> multiverion loop V2

     #pragma novector
     for ( i=0 ; i < n ; i++){ … }
}
```

Final Remarks on Multiversioning

Multiversioning done when

- Can’t determine trip count
- Can’t determine alignment (have a version for aligned and another version unaligned)
- Can’t determine stride
- offset = indx[i] ; a[i] = a[i + offset]*K;
  - Possibilities: offset negative, offset could be stride 1 or stride 2 or ?
    Indx[i] could be stepping 2, 4, 6, 8, etc (regular stride)
    OR indx[i] could be jumping all over memory (worse case but often the real-world case)
  - Compiler may create version for every possible scenario

Final Remarks on Peel/Kernel/Remainder

- Example shown was for 128bit vector-based processor
- AVX/AVX2 are 256bit.   AVX512 is 512 bit
- Cache line length == max vector length
  - Data moved to/from memory in cache lines == max vector length
- But for PEEL or REMAINDER, what if the # elements is equal to a smaller vector length?
  - Could do PEEL with a smaller SSE or AVX2 instruction on a AVX512 processor
  - OR could do 1 element serial and the rest of the PEEL with a SSE or AVX2 instruction
  - Same for REMAINDER loop – you may see vectorized PEEL or REMAINDER loops but they will be short loops or smaller vector instruction sequences

Follow Along Lab Exercise

Change directories to your lab directory and subdirectory “omp/opt-report-lab-2019/linux”

Choose your language, cd c   or cd fortran

\

C/C++ Inspect the func_step1 function

```
#include <math.h>
void func (float* theta, float* sth) {
  int i;
  for (i=0; i < 128; i++)
    sth[i] = sin(theta[i]+3.1415927);
}
```

```
subroutine func( theta, sth )
implicit none
real  :: theta(:), sth(:)
integer :: i
do i=1,128
   sth(i) = sth(i) + (3.1415927D0 * theta(i))
end do
end
```

Fortran: Inspect the func_step1 function

```
subroutine func( theta, sth )
implicit none
real  :: theta(:), sth(:)
integer :: i
do i=1,128
   sth(i) = sth(i) + (3.1415927D0 * theta(i))
end do
end
```

Compile, Generate Optimization Report phases vec,loop output to stderr

Run script “step1.sh”

./step1.sh

```
icc -c -qopt-report=4 -qopt-report-phase=loop,vec
    -qopt-report-file=stderr func_step1.c
```

```
ifort -c -qopt-report=4 -qopt-report-phase=loop,vec
    -qopt-report-file=stderr func_step1.f90
```

Actionable Messages, C, Step 1

$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr func_step1.c

Begin optimization report for: foo

Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at func_step1.c(4,3)
Multiversioned v1
remark #25231: Loop multiversioned for Data Dependence
remark #15135: vectorization support: reference theta has unaligned access
remark #15135: vectorization support: reference sth has unaligned access
remark #15127: vectorization support: unaligned access used inside loop body
remark #15145: vectorization support: unroll factor set to 2
remark #15164: vectorization support: number of FP up converts: single to double precision 1
remark #15165: vectorization support: number of FP down converts: double to single precision 1
remark #15002: **LOOP WAS VECTORIZED**
remark #36066: unmasked unaligned unit stride loads: 1
remark #36067: unmasked unaligned unit stride stores: 1
.... (loop cost summary) ....
remark #25018: Estimate of max trip count of loop=32
LOOP END

LOOP BEGIN at func_step1.c(4,3)
Multiversioned v2
remark #15006: **loop was not vectorized**: non-vectorizable loop instance from **multiversioning**
LOOP END
===

Arguments theta and sth may be aliased – have to assume this

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 128; i++)
      sth[i] = sin(theta[i]+3.1415927);
}
```

Actionable Messages, Fortran, Step 1

```
Begin optimization report for: FUNC

LOOP BEGIN at func_step1.f90(8,36)

<Peeled, Multiversioned v1>

LOOP END

LOOP BEGIN at func_step1.f90(8,36)

<Multiversioned v1>

   remark #25233: Loop multiversioned for stride tests on Assumed shape arrays

   remark #15388: vectorization support: reference sth has aligned access   [ func_step1.f90(8,3) ]

  remark #15388: vectorization support: reference theta has aligned access   [ func_step1.f90(8,3) ]

   <snip>

LOOP END

LOOP BEGIN at func_step1.f90(8,36)

<Alternate Alignment Vectorized Loop, Multiversioned v1>

   remark #25015: Estimate of max trip count of loop=16

LOOP END

LOOP BEGIN at func_step1.f90(8,36)

<Remainder, Multiversioned v1>
===========================================================================
```

Loop multiversioned due to Assumed Shape arrays

One version assumes contiguous data. This version has PEEL + Kernel + Remainder loops

Another version assumes non-contiguous arrays (strided) – look at the comment "masked strided loads. This has a kernel loop and a remainder loop

Next Steps: run ./step2.sh

C: Eliminate the multi-versioning due to possible alias of arguments ‘sth’ and ‘theta’. Methods:

1. Use compiler option –fargument-noalias
2. Use __restrict__ or C99 (float*restrict theta, …)
   along with –std=c99

What happens if they DO alias?

Fortran: declare the assumed shape arrays are CONTIGUOUS

real, contiguous :: theta(:), sth(:)

What happens if non-contiguous slices are passed?

Actionable Messages: C, step2

$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr \
    -fargument-noalias func_step2.c

(/Qalias-args- on Windows*)

Begin optimization report for: foo
  Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at func_step2.c(4,3)
  remark #15135: vectorization support: reference theta has unaligned access
  remark #15135: vectorization support: reference sth has unaligned access
  remark #15127: vectorization support: unaligned access used inside loop body
  remark #15145: vectorization support: unroll factor set to 2
  remark #15164: vectorization support: number of **FP up converts: single to double precision 1**
  remark #15165: vectorization support: number of **FP down converts: double to single precision 1**
  remark #15002: LOOP WAS VECTORIZED
  remark #36066: unmasked unaligned unit stride loads: 1
  remark #36067: unmasked unaligned unit stride stores: 1
  remark #36091: --- begin **vector loop cost summary** ---
  remark #36092: **scalar loop cost: 114**
  remark #36093: **vector loop cost: 55.750**
  remark #36094: **estimated potential speedup: 2.790**
  remark #36095: lightweight vector operations: 10
  remark #36096: medium-overhead vector operations: 1
  remark #36098: vectorized math library calls: 1
  remark #36103: **type converts: 2**
  remark #36104: --- end vector loop cost summary ---
  remark #25018: Estimate of max trip count of loop=32
LOOP END

```
/* a C99 version.
  compile with –std=c99 */
#include <math.h>
void foo (float *restrict theta, \
              float *restrict sth)   {
  int i;
  for (i = 0; i < 128; i++)
      sth[i] = sin(theta[i]+3.1415927);
}
```

Actionable Messages: Fortran, step2

ifort -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr func_step2.f90
Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at func_step2.f90(7,34)
remark #15388: vectorization support: reference sth has aligned access [func_step2.f90(7,1)]
remark #15388: vectorization support: reference sth has aligned access [func_step2.f90(7,1)]
remark #15388: vectorization support: reference theta has aligned access [func_step2.f90(7,1)]
remark #15399: vectorization support: unroll factor set to 4
remark #15417: vectorization support: number of FP up converts: single precision to double precision 2 [func_step2.f90(7,1)]
remark #15418: vectorization support: number of FP down converts: double precision to single precision 1 [func_step2.f90(7,1)]
remark #15300: LOOP WAS VECTORIZED
remark #15442: entire loop may be executed in remainder
remark #15448: unmasked aligned unit stride loads: 2
remark #15449: unmasked aligned unit stride stores: 1
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 12
remark #15477: vector loop cost: 10.000
remark #15478: **estimated potential speedup: 2.160**
remark #15479: lightweight vector operations: 10
remark #15487: type converts: 3
remark #15488: --- end vector loop cost summary ---
remark #25015: Estimate of max trip count of loop=16
LOOP END

Notice in report we have PEEL, kernel, remainder – no more masked strided version

```
subroutine func( theta, sth )
implicit none
real, contiguous :: theta(:), sth(:)
```

Next Steps: run ./step3.sh

Eliminate the type conversions, double-to-single and back.

C: replace 'sin()' with 'sinf()' and type cast the constant 3.1415927 with **3.1415927f**

Fortran: replace double constant 3.1415927D0 with single precision, use iso_fortran_env to help with readability

```
use iso_fortran_env
implicit none
..
integer, parameter :: sp = REAL32
integer, parameter :: dp = REAL64
do i=1,128
  sth(i) = sth(i) + (3.1415927_sp * theta(i))
end do
```

In Step 3, look in the opt-report for 'estimated potential speedup' – you should be impressed with the perf gain from simply cleaning up sloppy coding

Actionable Messages: C, Step 3

```
$ icc -c  -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias
func_step1.c

Begin optimization report for: foo

    Report from: Loop nest & Vector optimizations [loop, vec]


LOOP BEGIN at func_step1.c(4,3)
remark #15135: vectorization support: reference theta has unaligned access
   remark #15135: vectorization support: reference sth has unaligned access
   remark #15127: vectorization support: unaligned access used inside loop body
   remark #15002: LOOP WAS VECTORIZED
   remark #36066: unmasked unaligned unit stride loads: 1
   remark #36067: unmasked unaligned unit stride stores: 1
   remark #36091: --- begin vector loop cost summary ---
   remark #36092: scalar loop cost: 111
   remark #36093: vector loop cost: 28.000
   remark #36094: estimated potential speedup: 5.400
   remark #36095: lightweight vector operations: 9
   remark #36098: vectorized math library calls: 1
   remark #36104: --- end vector loop cost summary ---
   remark #25018: Estimate of max trip count of loop=32
LOOP END
```

Note no more up/down conversions
Estimated potential speedup:
Step2: 2.790
Step3: 5.400

Actionable Messages: Fortran, Step 3

```
ifort -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr func_step3.f90
   Report from: Loop nest & Vector optimizations [loop, vec]
LOOP BEGIN at func_step3.f90(11,37)
  remark #15388: vectorization support: reference sth has aligned access   [ func_step3.f90(11,3) ]
  remark #15388: vectorization support: reference sth has aligned access   [ func_step3.f90(11,3) ]
  remark #15388: vectorization support: reference theta has aligned access   [ func_step3.f90(11,3) ]
  remark #15399: vectorization support: unroll factor set to 2
  remark #15300: LOOP WAS VECTORIZED
  remark #15442: entire loop may be executed in remainder
  remark #15448: unmasked aligned unit stride loads: 2
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 8
  remark #15477: vector loop cost: 4.000
  remark #15478: estimated potential speedup: 3.220
  remark #15479: lightweight vector operations: 7
  remark #15488: --- end vector loop cost summary ---
  remark #25015: Estimate of max trip count of loop=16
LOOP END

LOOP BEGIN at func_step3.f90(11,37)
<Alternate Alignment Vectorized Loop>
  remark #25015: Estimate of max trip count of loop=16
LOOP END
```

Note no more up/down conversions
Estimated potential speedup:
Step2: 2.160
Step3: 3.220

Next Steps: run ./step4.sh

If data is aligned, which you should do, tell the compiler that sth and theta are aligned. This changes unaligned loads/stores with aligned loads/stores. And in some cases, the compiler won't have to create an aligned version of the loop and an unaligned version.

Alignment on Intel® Xeon Phi™ is key to performance – up to 20x performance improvement.

How to Align Data (C/C++)

Allocate memory on heap aligned to n byte boundary:

```
void* _mm_malloc(int size, int n)
int posix_memalign(void **p, size_t n, size_t size)
```

Alignment for variable declarations:

```
__attribute__((aligned(n)))  var_name
```

or

```
__declspec(align(n))  var_name
```

And tell the compiler…

```
#pragma vector aligned
#pragma omp simd aligned( var [,var…]:<n> )
```

- Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
- May cause fault if data are not aligned

```
__assume_aligned(array, n)
```

- Compiler may assume array is aligned to n byte boundary

n=64 for Intel® Xeon Phi™ coprocessors, n=32 for AVX, n=16 for SSE

How to Align Data (Fortran)

Align array on an "n"-byte boundary (n must be a power of 2)

```
!dir$ attributes align:n :: array
```

- Works for dynamic, automatic and static arrays (not in common)

For a 2D array, choose column length to be a multiple of n, so that consecutive columns have the same alignment (pad if necessary)

`-align array64byte` compiler tries to align all array types

And tell the compiler…

```
!dir$ vector aligned OR
 !$omp simd aligned( var [,var…]:<n>)
```

- Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
- May cause fault if data are not aligned

```
!dir$ assume_aligned array:n   [,array2:n2, …]
```

- Compiler may assume array is aligned to n byte boundary

n=64 for Intel® Xeon Phi™ coprocessors, n=32 for AVX, n=16 for SSE

Actionable Messages: C, Step4

icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias **-qopenmp-simd** func_step4.c

```
   Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at func_step4.c(7,8)
   remark #15388: vectorization support: reference theta has aligned access   [ func_step4.c(8,14) ]
   remark #15388: vectorization support: reference sth has aligned access   [ func_step4.c(8,5) ]
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 111
   remark #15477: vector loop cost: 19.750
   remark #15478: estimated potential speedup: 5.610
   remark #15479: lightweight vector operations: 8
   remark #15481: heavy-overhead vector operations: 1
   remark #15482: vectorized math library calls: 1
   remark #15488: --- end vector loop cost summary ---
   remark #25015: Estimate of max trip count of loop=32
LOOP END
===========================================================================
```

Note aligned accesses
Estimated potential speedup:
Step3: 5.400
Step4: 5.610

```
#pragma omp simd aligned( sth, theta:32)
 for (i=0; i < 128; i++)
   sth[i] = sinf(theta[i]+3.1415927f);
```

Actionable Messages: Fortran, Step4

ifort -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr
-qopenmp-simd func_step4.f90

```
  Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at func_step4.f90(10,21)
   remark #15388: vectorization support: reference sth has aligned access   [ func_step4.f90(12,3) ]
   remark #15388: vectorization support: reference sth has aligned access   [ func_step4.f90(12,3) ]
   remark #15388: vectorization support: reference theta has aligned access   [ func_step4.f90(12,3) ]
   remark #15399: vectorization support: unroll factor set to 8
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: [illegible]
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 8
   remark #15477: vector loop cost: 16.000
   remark #15478: estimated potential speedup: 4.000
   remark #15479: lightweight vector operations: 7
   remark #15488: --- end vector loop cost summary ---
   remark #25015: Estimate of max trip count of loop=4
LOOP END
===============================================================================
```

Note no more version unaligned
Estimated potential speedup:
Step3: 3.220
Step4: 4.000

```
!$omp simd aligned( theta, sth:64 )
do i=1,128
  sth(i) = sth(i) + (3.1415927_sp * theta(i))
end do
!$omp end simd
```

Next Steps: run ./step5.sh

If you don't use a –O option, default optimization is O2

At O2 and O3, the compiler auto-vectorizes your code

BUT it assumes 'lowest common denominator' processor and uses older 128 SSE instructions.

Most modern ("Sandy Bridge and better, post-2011) support 256-bit AVX. AVX-512 is common now in server chips

In Step5 we add –xavx to get 256-bit vector instructions

If you are not using a –x<arch> or –ax<arch> option, you are potentially not gaining on an easy 2-4x performance gain

Actionable Messages: C, Step 5

$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias **-xavx** func_step1.c

```
Begin optimization report for: foo

   Report from: Loop nest & Vector optimizations [loop, vec]
LOOP BEGIN at func_step5.c(5,8)
  remark #15388: vectorization support: reference theta has aligned access   [ func_step5.c(6,14) ]
  remark #15388: vectorization support: reference sth has aligned access   [ func_step5.c(6,5) ]
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 1
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 110
  remark #15477: vector loop cost: 9.870
  remark #15478: estimated potential speedup: 11.130
  remark #15479: lightweight vector operations: 8
  remark #15481: heavy-overhead vector operations: 1
  remark #15482: vectorized math library calls: 1
  remark #15488: --- end vector loop cost summary ---
  remark #25015: Estimate of max trip count of loop=16
LOOP
END===========================================================
```

Note loop trip count went from 32 to 16
Estimated potential speedup:
Step4: 5.610
Step5: 11.130 !!!

Actionable Messages: Fortran, Step 5

ifort -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr
-xavx -qopenmp-simd func_step5.f90

```
   Report from: Loop nest & Vector optimizations [loop, vec]
LOOP BEGIN at func_step5.c(5,8)
  remark #15388: vectorization support: reference theta has aligned access   [ func_step5.c(6,14) ]
  remark #15388: vectorization support: reference sth has aligned access   [ func_step5.c(6,5) ]
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 1
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 110
  remark #15477: vector loop cost: 9.870
  remark #15478: estimated potential speedup: 9.140
  remark #15479: lightweight vector operations: 8
  remark #15481: heavy-overhead vector operations: 1
  remark #15482: vectorized math library calls: 1
  remark #15488: --- end vector loop cost summary ---
  remark #25015: Estimate of max trip count of loop=16
LOOP
END===========================================================
```

Note loop trip count went from 32 to 16
Estimated potential speedup:
Step4: 4.000
Step5: 9.140!!!

Check Point – Progress so far

C:

Step1: estimated potential speedup: 2.790

Step5: estimated potential speedup: 11.130 **~4X speedup!**

Fortran:

Step1: estimated potential speedup: 1.400

Step5: estimated potential speedup: 9.140 **~6.5X speedup!**

step5-avx512.sh

Run step5-avx512.sh

This replaces AVX with AVX512. Potentially can give us 2x

```
FORTRAN example:
LOOP BEGIN at func_step5.f90(10,21)
   remark #15388: vectorization support: reference sth(i) has aligned access   [ func_step5.f90(12,3) ]
   remark #15388: vectorization support: reference sth(i) has aligned access   [ func_step5.f90(12,12) ]
   remark #15388: vectorization support: reference theta(i) has aligned access   [ func_step5.f90(12,37) ]
   remark #15305: vectorization support: vector length 8
   remark #15399: vectorization support: unroll factor set to 8
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #26013: Compiler has chosen to target XMM/YMM vector. Try using -qopt-zmm-usage=high to override
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 8
   remark #15477: vector cost: 0.870
   remark #15478: estimated potential speedup: 9.140
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=2
LOOP END
```

WAIT – speedup is THE SAME as AVX!

What is this option

-qopt-zmm-usage=high ??

Skylake Notes

-xcore-avx512 or –xskylake-avx512 may favor AVX2 instead of AVX512

Override with

-xcore-avx512 –qopt-zmm-usage=high

Or

-xcommon-avx512

Skylake ONLY. Icelake and above will favor AVX512

Run ./step5-skylake.sh to compile with –xskylake-avx512 –qopt-zmm-usage=high

Icelake:

–xicelake-server # don't need –qopt-zmm-usage=high

Check Point – Progress so far

C:

Step1: estimated potential speedup: 2.790

Step5: estimated potential speedup: 11.130 **~4X speedup!**

Step5-skylake est potential speedup: 20.54 **~7.4x speedup!**

Fortran:

Step1: estimated potential speedup: 1.400

Step5: estimated potential speedup: 9.140 **~6.5x speedup!**

Step5-skylake est potential speedup: 18.28 **~13x speedup!**

Other Optimizations: run ./step6.sh

What happens if the loop has a large trip count?

If the code writes out a long vector or array, by default through cache, the data cache is not big enough to hold the data and all existing data is flushed out.

Sometimes you want to 'bypass cache' aka STREAMING STORES

With a fixed, large trip count, the compiler will automatically generate streaming store instructions.

Or you can control with –qopt-streaming-stores <setting>

OR #pragma vector nontemporal   !dir$ vector nontemporal

In this step we change the loop upper bound from 128 to 2,000,00 and look for report to tell us when streaming stores are enabled

Actionable Messages: C, Step6

```
icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias -
qopenmp-simd -xavx func_step6.c
   Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at func_step6.c(5,8)
  remark #15388: vectorization support: reference theta has aligned access   [ func_step6.c(6,14) ]
  remark #15388: vectorization support: reference sth has aligned access   [ func_step6.c(6,5) ]
  remark #15412: vectorization support: streaming store was generated for sth   [ func_step6.c(6,5) ]
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 1
  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: 109
  remark #15477: vector loop cost: 5.06
  remark #15478: estimated potential speedup: 21.53
  remark #15479: lightweight vector operations: 8
  remark #15481: heavy-overhead vector operations: 1
  remark #15482: vectorized math library calls: 1
  remark #15488: --- end vector loop cost summary ---
  remark #25015: Estimate of max trip count of loop=250000
LOOP END
===========================================================================
```

```
for (i = 0; i < 2000000; i++)
   sth[i] = sinf(theta[i]+3.1415927f);
}
```

Actionable Messages: Fortran, Step6

ifort -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -xavx -qopenmp-simd **-qopt-streaming-stores always** func_step6.f90
Report from: Loop nest & Vector optimizations [loop, vec]

```
LOOP BEGIN at func_step6.f90(10,21)
  remark #15388: vectorization support: reference sth has aligned access   [ func_step6.f90(12,3) ]
  remark #15388: vectorization support: reference sth has aligned access   [ func_step6.f90(12,3) ]
  remark #15388: vectorization support: reference theta has aligned access   [ func_step6.f90(12,3) ]
  remark #15412: vectorization support: streaming store was generated for sth   [ func_step6.f90(12,3) ]
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 2
  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: 8
  remark #15477: vector loop cost: 0.43
  remark #15478: estimated potential speedup: 18.280
  remark #15479: lightweight vector operations: 7
  remark #15488: --- end vector loop cost summary ---
  remark #25015: Estimate of max trip count of loop=250000
LOOP END
===========================================================================
```

```
!$omp simd aligned( theta, sth:64 )
do i=1,2000000
  sth(i) = sth(i) + (3.1415927_sp * theta(i))
end do
!$omp end simd
```

Next Steps: run ./step7.sh

So far the loop count has been a constant.

What if the loop trip count is passed as an argument?
force streaming stores with [–q|/Q]opt-streaming-stores always

```
void func (float* theta, float* sth, int n) {
…
for (i=0; i < n; i++)
    sth[i] = sinf(theta[i]+3.1415927f);


subroutine func( theta, sth, n )
…
do i=1,n
  sth(i) = sth(i) + (3.1415927_sp * theta(i))
end do
```

Actionable Messages: C, Step7

icc -c **-qopt-streaming-stores always** -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias -qopenmp-simd -xavx func_step7.c

```
Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at func_step7.c(5,3)
   remark #15388: vectorization support: reference theta has aligned access   [ func_step7.c(6,14) ]
   remark #15388: vectorization support: reference sth has aligned access   [ func_step7.c(6,5) ]
   remark #15412: vectorization support: streaming store was generated for sth   [ func_step7.c(6,5) ]
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   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: 109
   remark #15477: vector loop cost: 5.06
   remark #15478: estimated potential speedup: 18.060
   remark #15482: vectorized math library calls: 1
   remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at func_step7.c(5,3)
<Remainder>
LOOP END
```

Talking point: why do we have a remainder loop now? Why didn't we get it before?
With a variable trip count, how does the compiler know how many iterations in the remainder?

Actionable Messages: Fortran, Step7

ifort -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -xavx -qopenmp-simd **-qopt-streaming-stores always** func_step7.f90

LOOP BEGIN at func_step7.f90(13,1)
remark #15388: vectorization support: reference sth has aligned access [func_step7.f90(14,3)]
remark #15388: vectorization support: reference sth has aligned access [func_step7.f90(14,3)]
remark #15388: vectorization support: reference theta has aligned access [func_step7.f90(14,3)]
remark #15412: vectorization support: streaming store was generated for sth [func_step7.f90(14,3)]
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 2
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: 8
remark #15477: vector loop cost: 0.430
remark #15478: estimated potential speedup: 17.140
R emark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at func_step7.f90(13,1)
<Remainder>
LOOP END

Talking point: why do we have a remainder loop now? Why didn't we get it before?
With a variable trip count, how does the compiler know how many iterations in the remainder?

Next Steps: run ./step8.sh

"–qopt-streaming-stores always " affects the entire source file

To be more strategic, several options:

1. Use #pragma/!dir$ loop count <settings> to give the compiler hints, let it determine when to make streaming stores
2. Use #pragma/!dir$ vector nontemporal to target specific loops
3. Use PGO, the compiler will use observed trip counts to determine when to use streaming stores

Let's use #pragma/!dir$ loop count min option and remove –qopt-streaming-stores

#pragma loop count min(2000000)

!dir$ loop count min=2000000

Actionable Messages: C, Step8

icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias -qopenmp-simd -xavx func_step8.c

```
   Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at func_step8.c(6,3)
   remark #15388: vectorization support: reference theta has aligned access   [ func_step8.c(7,14) ]
   remark #15388: vectorization support: reference sth has aligned access   [ func_step8.c(7,5) ]
   remark #15412: vectorization support: streaming store was generated for sth   [ func_step8.c(7,5) ]
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   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: 109
   remark #15477: vector loop cost: 5.060
   remark #15478: estimated potential speedup: 21.530
   remark #15482: vectorized math library calls: 1
   remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at func_step7.c(5,3)
<Remainder>
LOOP END
```

Actionable Messages: Fortran, Step8

ifort -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -xavx -qopenmp-simd func_step8.f90

Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at func_step8.f90(13,1)
remark #15388: vectorization support: reference sth has aligned access [func_step8.f90(14,3)]
remark #15388: vectorization support: reference theta has aligned access [func_step8.f90(14,3)]
remark #15412: vectorization support: **streaming store was generated for sth** [func_step8.f90(14,3)]
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 1
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: 6
remark #15477: vector loop cost: 0.430
remark #15478: estimated potential speedup: 17.140
remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at func_step7.c(5,3)
<Remainder>
LOOP END

C: Final Comments on This Example

```
1 #include <math.h>
2 void func (float* theta, float* sth) {
3   int i;
4   #pragma omp simd aligned( sth, theta:32)
5   for (i=0; i < 128; i++)
6     sth[i] = sinf(theta[i]+3.1415927f);
7 }
```

```
LOOP BEGIN at func_step5.c(5,8)
  remark #15388: vectorization support: reference theta has aligned access   [ func_step5.c(6,14) ]
  remark #15388: vectorization support: reference sth has aligned access   [ func_step5.c(6,5) ]
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 1
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 110
  remark #15477: vector loop cost: 9.870
  remark #15478: estimated potential speedup: 11.130

  remark #15482: vectorized math library calls: 1
  remark #15488: --- end vector loop cost summary ---
  remark #25015: Estimate of max trip count of loop=16
LOOP END
```

General ops estimate

call to vectorized sinf libsvml

Fortran: Final Comments on Example

```
!.... A slightly more complex expression with SIN
10 !$omp simd aligned( theta, sth:64 )
11 do i=1,128
12  sth(i) = sth(i) + sin((3.1415927_sp * theta(i)))
13  13 end do
14 !$omp end simd
```

```
LOOP BEGIN at func_step5_morecomplex.f90(10,21)
   remark #15388: vectorization support: reference theta has aligned access   [ func_step5_morecomplex.f90(12,21) ]
   remark #15388: vectorization support: reference sth has aligned access   [ func_step5_morecomplex.f90(12,3) ]
   remark #15388: vectorization support: reference sth has aligned access   [ func_step5_morecomplex.f90(12,3) ]
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 110
   remark #15477: vector loop cost: 10.120
   remark #15478: estimated potential speedup: 10.850
   remark #15482: vectorized math library calls: 1
   remark #15488: --- end vector loop cost summary ---
   remark #25015: Estimate of max trip count of loop=16
LOOP END
```

General ops estimate

“vectorized math call” call to vectorized sin function (in libsvml)

Reports On Other Optimization Phases

-qopt-report-phase=

par	auto-parallelization report, structured similarly to vectorization report
openmp	report on OpenMP constructs merged into the loop report
pgo	report on Profile Guided Optimization, including which functions had useful profiles
cg	optimizations during code generation, such as intrinsic function lowering
loop	additional loop and memory optimizations, such as cache blocking, prefetching, scalar replacement, etc.
tcollect	data collection for Intel® Trace Analyzer

Example Code for IPO Report

```
#include <stdio.h>

static void __attribute__((noinline))
bar  (float a[100][100], float b[100][100]) {
  int i, j;
  for (i = 0; i < 100; i++) {
      for (j = 0; j < 100; j++) {
          a[i][j] = a[i][j] + 2 * i;
          b[i][j] = b[i][j] + 4 * j;
      }
   }
 }

static void foo(float a[100][100],
                float b[100][100])  {
 int i, j;
  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      a[i][j] = 2 * i;
        b[i][j] = 4 * j;
    }
  }
  bar(a, b);
 }

extern int main() {
int i, j;
 float a[100][100];
  float b[100][100];

for (i = 0; i < 100; i++) {
     for (j = 0; j < 100; j++) {
        a[i][j] = i + j;
         b[i][j] = i - j;
     }
 }
 foo(a, b);
 foo(a, b);
 fprintf(stderr, "%d %d\n",
                a[99][9], b[1]99]);
}
```

Compiled with:

icc -qopt-report=3
 –opt-report-phase=ipo sm.c

10/1/19

Features of the IPO Report – Inlining

-qopt-report-phase=ipo –opt-report=3

Settings that control the amount of inlining allowed

```
INLINING OPTION VALUES:
  -inline-factor: 100
...
```

Report for function main at line 24 of source file sm.c

foo() is inlined at lines 35 & 36

bar() called from foo at line 21 but not inlined into main

External function fprintf

```
INLINE REPORT: (main) [1]  sm.c(24,19)
  -> INLINE: [35] foo()
    -> [21] bar()
  -> INLINE: [36] foo()
    -> [21] bar()
   ->EXTERN: [37] fprintf
```

User function bar() at line 3 has no function calls

```
INLINE REPORT: (bar) [2] sm(3,81)
```

Static function foo() at line 13 is dead if all calls to it are inlined

```
DEAD STATIC FUNCTION: (foo) sm.c(13,55)
```

Features of the IPO Report – more detail

-qopt-report-phase=ipo –opt-report=4

Whole Program Optimization report

WHOLE PROGRAM (SAFE)
  [EITHER METHOD]: true
 WHOLE PROGRAM (SEEN)
  [TABLE METHOD]: true
 WHOLE PROGRAM (READ)
  OBJECT READER METHOD]: false

% of total routines compiled so far

sz = Size of each inlineable routine in intermediate language units (total = (stmts + exprs))

isz = Increase in size of caller due to inlining

Reasons routines were not inlined

INLINE REPORT: (main) [1/3=33.3%] sm.c(24,19)

 -> INLINE: [35] foo (isz = 40) (**sz** = 47 (25+22))

  -> [21] bar() (isz = 47) (**sz** = 54 (24+30))

   [[Called routine is noinline]]

 -> INLINE: [35] foo (isz = 40) (**sz** = 47 (25+22))

  -> [21] bar() (isz = 47) (**sz** = 54 (24+30))

   [[Called routine is noinline]]

 -> EXTERN: [37] fprintf

Offload Report for Intel® Xeon Phi™ coprocessors

Compile with –opt-report-phase=offload

Separate reports are generated for host and coprocessor

Reports for offloads using Intel® Cilk™ Plus keywords and also for offloads using Intel or OpenMP 4.0 pragmas or directives

Example for OpenMP 4.0 offload pragma:

icc –c -openmp -qopt-report-phase=offload offload_test.c

Offload Report – Example with OpenMP

```
01 #pragma omp declare target
02 int compute(int i)  { return i++; }
03 #pragma omp end declare target
04
05 int do_offload() {
06    int i = 0;
07 #pragma omp target map(tofrom:i)
08      { i = compute(i); }
09    return i;
10 }
```

Host Report

```
offload_test.c(6-6):OFFLOAD:do_offload:  Offload to target MIC 1
 Data sent from host to target
       i, scalar size 4 bytes
 Data received by host from target
       i, scalar size 4 bytes
```

Coprocessor Report

```
offload_test.c(6-6):OFFLOAD:do_offload:  Outlined offload region
 Data received by target from host
       i, scalar size 4 bytes
 Data sent from target to host
       i, scalar size 4 bytes
```

Mapping old switches to new

-vec-report, -par-report and -openmp-report are deprecated.

They do not give the same output as for the version 14 compiler.

Instead, they are mapped to the closest equivalent phase and level of the new optimization report. Reports are not written to stderr unless you set –opt-report-file=stderr or put this into your configuration file.

Users are encouraged to convert do the new, more powerful switches. You may want to delete *.optrpt files in the ”clean” section of your makefiles.

Further Information on vectorization

The Intel® Compiler User Guides:

https://software.intel.com/en-us/compiler_15.0_ug_f

Series of short, audio-visual vectorization tutorials:

https://software.intel.com/en-us/search/site/field_tags/explicit-vector-programming-43556

New Optimization Report (compilers version 15.0+)

https://software.intel.com/en-us/videos/getting-the-most-out-of-the-intel-compiler-with-new-optimization-reports

Other articles:

- Requirements for Vectorizable Loops

http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops

- Explicit Vector Programming in Fortran

https://software.intel.com/en-us/articles/explicit-vector-programming-in-fortran

- Fortran Array Data and Arguments and Vectorization

https://software.intel.com/en-us/articles/fortran-array-data-and-arguments-and-vectorization

LAB 1-1 BEST AFFINITY CONTROL WITH OPENMP*

Logical Processor Mapping 64-core KNL Node

Core 0	Core 1		Core 15	Core 16	Core 17		Core 31
Proc 0	Proc 1		Proc 15	Proc 16	Proc 17		Proc 31
Proc 64	Proc 65	***	Proc 79	Proc 80	Proc 81	***	Proc 95
Proc 128	Proc 129		Proc 143	Proc 144	Proc 145		Proc 159
Proc 192	Proc 193		Proc 207	Proc 208	Proc 209		Proc 223

Core 32	Core 33		Core 47	Core 48	Core 49		Core 63
Proc 32	Proc 33		Proc 47	Proc 48	Proc 49		Proc 63
Proc 96	Proc 97	***	Proc 111	Proc 112	Proc 113	***	Proc 127
Proc 160	Proc 161		Proc 175	Proc 176	Proc 177		Proc 191
Proc 224	Proc 225		Proc 239	Proc 240	Proc 241		Proc 255

Lab 1 - OpenMP* Affinity Control

We will use a simple hand-written matrix-matrix multiplication example to illustrate the effect of affinity on runtime.

To get started, change into the “lab1” **affinity** directory:

```
$ cd omp/lab1
```

Inside this directory you will find a simple **build.sh** script and COBALT submission script – **lab1.run.**

Start by executing the build script:

```
$ ./build.sh
```

This will generate the **mat.omp** executable that you need to complete this exercise.

Lab 1-1 OpenMP* Affinity Control

Examine and then submit the **lab1.run** script to run the example code with a variety of affinity settings and thread counts:

```
$ qsub ./lab1-1.run
```

This will generate an output file, **lab1-1.out**, which contains details of each run configuration and the approximate performance achieved.

Inspect **"lab1-1.out"** and try to answer the following questions:

- What seems to be the best affinity setting combination for this code?
- What is the speedup achieved by using optimal affinity settings?
- Can you modify the submission script to add other affinity settings (or thread counts) and test to see if there are alternatives that work better?

Lab 1-1 Solution

The best combination should be using the following:

- OMP_NUM_THREADS=64
- OMP_PLACES=cores
- OMP_PROC_BIND=spread

Note the following characteristics:

- Since KNL is capable of issuing 2 vector instructions per core per cycle from a single thread ,there may not a need to go over 64 threads to achieve maximum performance in a code of this type - Feel free to try and measure the performance.
- Using a compact affinity setting leaves cores unused and leads to lower overall performance.

LAB1-2 VERIFY YOUR BINDING

OMP_DISPLAY_AFFINITY

At the start of the process, display the binding or affinity of the OMP threads

- Environment variable, default is FALSE

```
export OMP_DISPLAY_AFFINITY=true
```

- On Theta this SHOULD work but seems to be ignored in qsub script (ideas?)
  - Shell env, export in run script, passed with –env on aprun
- Alternative `KMP_AFFINITY=verbose`
- Pass with aprun:
  - `aprun –n 1 –N 1 --env KMP_AFFINITY=verbose –cc none ./mat.omp &>> lab1-2.out`

Lab1-2 Why Bind “close” is slow

- `qsub lab1-2.run` # OMP_PROC_BIND=close

```
grep tid lab1-2.out | sort -n
```

Core 0
Proc 0
Proc 64
Proc 128
Proc 192

Core 1
Proc 1
Proc 65
Proc 129
Proc 193

***

Core 15
Proc 15
Proc 79
Proc 143
Proc 207

Core 16
Proc 16
Proc 80
Proc 144
Proc 208

Core 17
Proc 17
Proc 81
Proc 145
Proc 209

***

Core 31
Proc 31
Proc 95
Proc 159
Proc 223

Core 32
Proc 32
Proc 96
Proc 160
Proc 224

Core 33
Proc 33
Proc 97
Proc 161
Proc 225

***

Core 47
Proc 47
Proc 111
Proc 175
Proc 239

Core 48
Proc 48
Proc 112
Proc 176
Proc 240

Core 49
Proc 49
Proc 113
Proc 177
Proc 241

***

Core 63
Proc 63
Proc 127
Proc 191
Proc 255

LAB1-3 USE ALPS TO CONTROL PROC SET

aprun –j 1 –cc depth –d 64

- qsub lab1-3.run
- With –j 1 we only use 1 Processor (HW thread) per core
- 64 threads for matmult
- 2 run experiments:

1. We set OMP_PROC_BIND=close
2. Then try OMP_PROC_BIND=spread

Compare GFLOPS lab1-3.out lab1-3.out

grep GFLOPS lab1-3.out

Did CLOSE or SPREAD make a difference? Must be +-3% to be above noise. Why/Why not?

LAB 2 - BASIC TASK CONCEPTS

Lab 2 - Basic Task Generation and Execution

In this example you will build a simple code that uses tasks to print out the simple sentence:

```
Hello World from OpenMP!
```

First, change to the basic directory:

```
$ cd ./basic
```

Now edit the provided sequential version **basic.c** so that each of the words in the sentence is printed to screen from a separate task. Remember that you will have to:

- Define a parallel region
- Generate the tasks within a single construct

Compile your new version (don't forget the **-qopenmp** flag) and ensure there are no compilation errors.

Lab 2 - Testing

Now launch the provided **basic.run** script so that you can see the output of your code when using multiple threads:

```
$ qsub ./basic.run
```

The script assumes your executable is called **a.out**, and provides the output in file **basic.out**.

Did the sentence come out correctly? It is unlikely, unless you used any type of synchronization - if you did you are ahead of the game - congratulations!

Now try to come up with **two** implementations that write the output in order while still using the same number of tasks. Do not worry about serialization - this exercise is not about performance, but methodology.

Lab 2 - Solution 1

In solution 1 we simply place a **taskwait** statement in between each printf command, so that the output is serialized.

This is a simple way of ensuring order but, in more complex problems it completely defies the purpose of using OpenMP* in the first place.

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        printf("Hello ");
        #pragma omp taskwait
        #pragma omp task
        printf("World ");
        #pragma omp taskwait
        #pragma omp task
        printf("from");
        #pragma omp taskwait
        #pragma omp task
        printf("OpenMP!");
    }
}
```

Lab 2 - Solution 2

In solution 2 we use the alternative method of defining dependencies among tasks.

In this simple example the result is the same - complete reordering at the expense of full serialization.

But in more complex codes defining dependencies may allow for greater parallel execution opportunities at runtime.

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task depend(out:a)
        a = printf("Hello ");

        #pragma omp task depend(in:a) depend(out:b)
        b = printf("World ");

        #pragma omp task depend(in:b) depend(out:c)
        c = printf("from");

        #pragma omp task depend(in:c)
        printf("OpenMP!");
    }
}
```

LAB 3 - FIBONACCI GENERATOR

Lab 3 - A Simple Fibonacci Number Generator

The Fibonacci series is an integer series defined by having numbers which, after the first one, are the sum of the previous two in the series:

```
1, 1, 2, 3, 5, 8, 13, 21, …
```

A simple Fibonacci generator can be coded as a recursive function:

```
int main(int argc,
         char *argv[])
{
    ...
    answer = fib( number );
    ...
}
```

```
int fib( int n)
{
    if( n < 2 ) return n;
    int i = fib( n - 1 );
    int j = fib( n - 2 );
    return i+j;
}
```

Your mission, should you choose to accept it, is to create a new version of this function that can be executed in parallel using OpenMP* constructs.

The following slides guide you through the process, and point to a solution in case you get stuck.

Lab 3 - Getting started

First go to the Fibonacci directory:

```
$ cd ../fibonacci
```

Inside this directory you will find three subdirectories named ver0, ver1, ver2. They each correspond to a version of the Fibonnaci number generator:

- ver0 - serial implementation, for reference and getting started.
- ver1 - proposed simple tasking solution
- ver2 - proposed refined tasking solution

Start by making a copy of version 0 so that you can work with it and still have a clear reference code to go back to:

```
$ cp ./ver0/* ./
```

Lab 3 - Some Hints

I'm not going to tell you exactly how to do this, but remember two critical things:

1. You MUST initiate the task generation process inside a single region within a parallel OpenMP* region - in this case main would be the right place to do this.

2. If you look inside the fib.c source file you will see that the fib() function either returns immediately or has two independent tasks to perform.

3. Once those tasks are performed their value is added and returned - perhaps an appropriate location for a synchronization point.

Try to use this hints and what you have learned to parallelize this recursive code using OpenMP* tasks.

Next slide has the answer if you get stuck!

Lab 3 - Proposed Solution (ver1)

Our proposed solution has a single task entering the function fib() from main(). It then generates two additional tasks to execute calls to fib() independently for (n-1) and (n-2):

```
int main(int argc,
         char *argv[])
{
    ...
    #pragma omp parallel
    {
         #pragma omp single
         {
              #pragma omp task
              answer = fib( number );
         }
    }
    ...
}
```

```
int fib( int n)
{
    if( n < 2 ) return n;
    int i, j;
    #pragma omp task shared(i)
    {
         i = fib( n - 1 );
    }
    #pragma omp task shared(j)
    {
         j = fib( n - 2 );
    }
    #pragma omp taskwait
         return i+j;
}
```

Lab 3 - Analysis of the Solution

Whether using your own version or the proposed solution in directory **ver1**, submit a quick job to determine how scalable your implementation is:

```
$ qsub ./tasking.run
```

This will save the number of threads and the time taken to determine the 41st number in the Fibonacci series to an output file called **tasking.out**.

- What is the best speedup you can get out of this code, from 4 to 128 threads?
- Is this faster or slower than the original serial implementation?
- Can you think of any way to improve the proposed solution?

Lab 3 - A Better Solution (ver2)

It turns out that the proposed solution in **ver1** works correctly, but generates excessive overhead by generating too many tasks.

Ideally one would include a variable threshold below which a serial function is used rather than a parallelized one. This is what the solution in the directory **ver2** provides.

Try to develop your own version of this hybrid code that enables better workload balance or, if you prefer, look at the solution provided in **ver2** and described in the next slide.

Go to the **ver2** directory (or use your own solution) to submit the **tasking.run** script to complete a new scalability analysis. Can you see the difference in scalability and speedup?

Feel free to change the value of the defined “SPLITTER” variable and observe its effects on performance. Remember you will need to recompile the code each time you make a change to this variable.

Lab 3 - Proposed Solution (ver2)

Our proposed solution does not create a new task once a small enough **n** is reached:

```
int main(int argc,
         char *argv[])
{
    ...
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            answer = fib( number );
        }
    }
    ...
}
```

```
int fib( int n)
{
    if( n < 2 ) return n;
    int i, j;
    #pragma omp task shared(i) if(n>30)
    {
        i = fib( n - 1 );
    }
    #pragma omp task shared(j) if(n>30)
    {
        j = fib( n - 2 );
    }
    #pragma omp taskwait
        return i+j;
}
```

Thank You for attending our OpenMP Hands-On!

Backup

Knights Landing Architectural Diagram

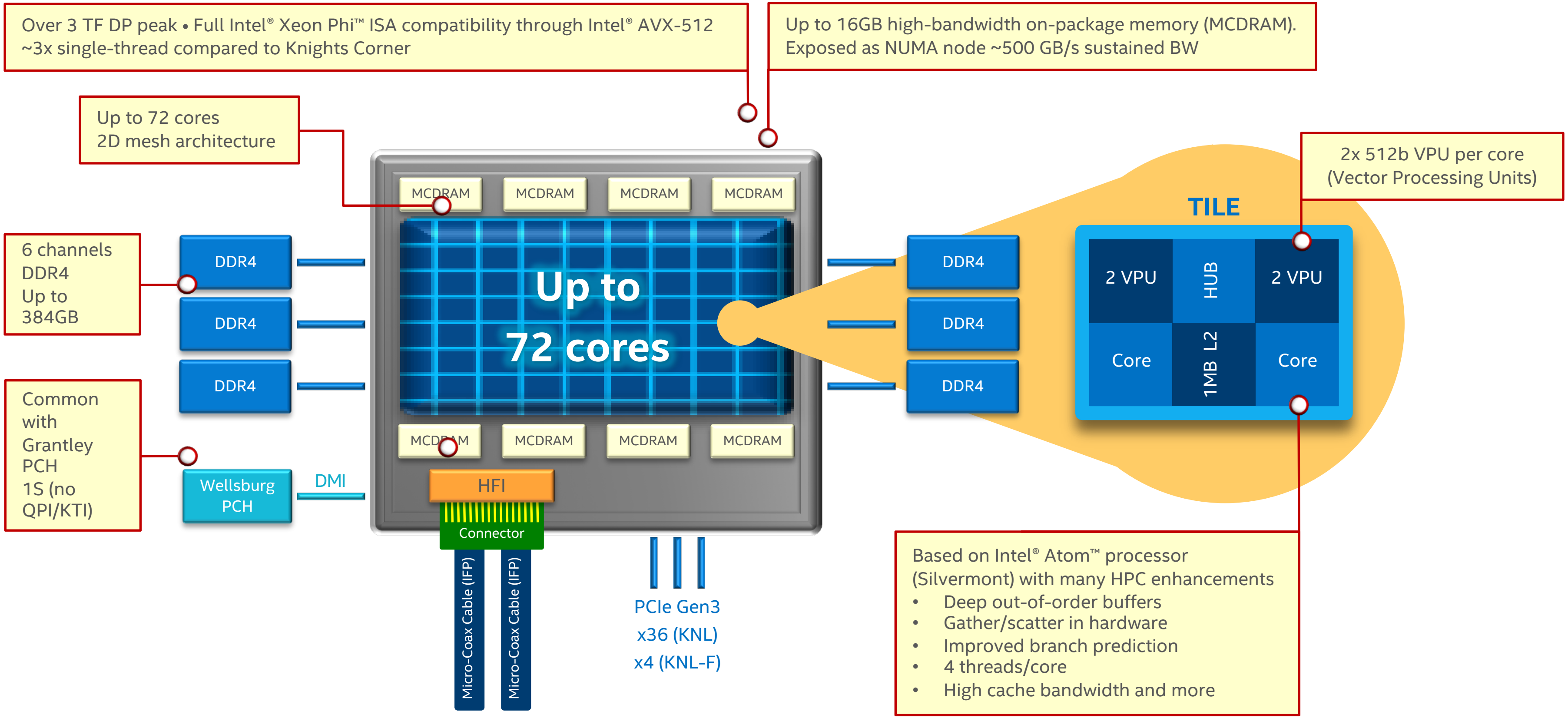
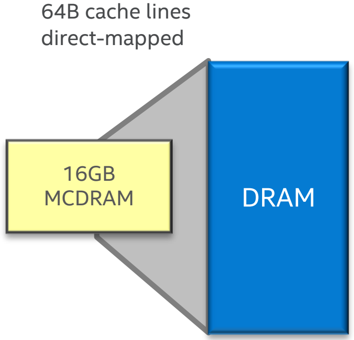
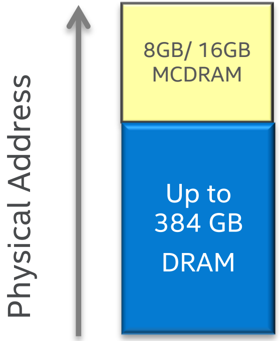


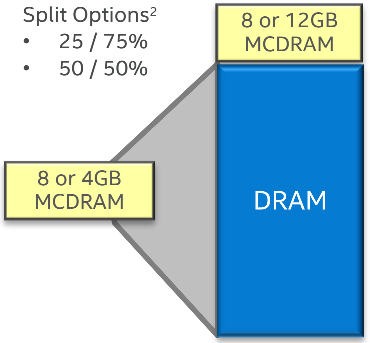
Diagram is for conceptual purposes only and only illustrates a CPU and memory • It is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

Integrated On-Package Memory Usage Models

Model configurable at boot time and software exposed through NUMA[1]







	Cache Model	Flat Model	Hybrid Model
Description	Hardware automatically manages the MCDRAM as a "L3 cache" between CPU and ext DDR memory	Manually manage how the app uses the integrated on-package memory and external DDR for peak perf	Harness the benefits of both Cache and Flat models by segmenting the integrated on-package memory
Usage Model	▪ App and/or data set is very large and will not fit into MCDRAM ▪ Unknown or unstructured memory access behavior	▪ App or portion of an app or data set that can be, or is needed to be "locked" into MCDRAM so it doesn't get flushed out	▪ Need to "lock" in a relatively small portion of an app or data set via the Flat model ▪ Remaining MCDRAM can then be configured as Cache