Programming, Compiling and Optimizing for the Intel® Xeon Phi™ Coprocessor

based on the same, familiar tools and methods that you already use for multi-core systems

Martyn Corden
Technical Consulting Engineer
Intel Corporation
Objectives of the Tutorial

Learning to use the Intel® compiler to build and optimize native and offload applications for Intel® Xeon Phi™ coprocessors

Appreciation of the importance of vectorization; understanding how to go about getting loops to vectorize; pros and cons of alternative approaches

Familiarity with the OpenMP environment on the coprocessor

Learning about memory and cache optimizations

Understanding of what can give rise to variations in floating-point results and how to minimize these
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

>50 in-order cores
- Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family
- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core
- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency

Reminder!
Programming Model - Introduction

The programming model and compiler make it easy to develop or port code to run on a system with an Intel® Xeon Phi™ coprocessor

Full integration into both C/C++ and Fortran

Enables use of Intel’s optimizing compilers on both host and coprocessor

- Vectorization
- Parallel programming with TBB, Intel® Cilk™ Plus, OpenMP, MPI, OpenCL

Enables *co-operative* processing between host and coprocessor

- Use both simultaneously for parallel processing
- Use host for serial code, coprocessor for data-parallel code

Using the Intel® Xeon Phi™ coprocessor is a simple extension of programming for Intel® Xeon® processors
Programming Models

An Intel® Xeon Phi™ coprocessor is accessed via the host system, but may be programmed either as a coprocessor(s) or as an autonomous processor.

The appropriate model may depend on application and context.

**Heterogeneous (Offload)**
- Better serial processing
- More memory
- Better file access
- Makes fuller use of available resources

**Native (Autonomous)**
- Simpler programming model
- Easier or no code porting
- More constraints
- Maybe quicker route for initial testing of key kernels
Native model may be appropriate if app:

Contains very little serial processing
Has a modest memory footprint
Has a very complex code structure and/or
does not have well-identified hot kernels than can be offloaded without substantial data transfer overhead
Does not perform extensive I/O

Data parallelism, use of parallel algorithms and application scalability are criteria for targeting Intel® MIC Architecture, but not for distinguishing between native or offload modes.
Building Native Applications

Cross compiler only, (same one as used for offload)

- Set environment in the usual way
  
  ```
  source /opt/intel/compilerxe/bin/compilervars.sh intel64
  ```

Build on host with `–mmic` (this sets the `__MIC__` macro)

Either

- copy executable, small data, dependencies to coprocessor with `scp`
  - e.g. `libiomp5.so`
  - Files are not permanent (in RAM) – recopy after reboot

Or

- Mount host file system on coprocessor using NFS
Running Native Applications

Either

- login to coprocessor with “ssh coprocessor_name” (e.g. mic0)
- Set environment, e.g. LD_LIBRARY_PATH, ulimit -s;
- run program

Or

- remotely submit a shell script that sets environment and runs app with ssh mic0 './myscript.sh'
- Useful for performance analysis with Intel® Vtune™ Amplifier XE

The Intel® Xeon Phi™ Coprocessor runs a reduced form of Linux*

- Many familiar commands are available
- top, time, ...
Building Native Libraries

Shared Libraries

Use the standard method for creating shared objects and also include \(-\text{mmic}\)

- \$ \texttt{icc }-\text{mmic }\texttt{-c }-\text{fpic }\texttt{mylib.c} // Creates \texttt{mylib.o} by default
- \$ \texttt{icc }-\text{mmic }-\text{shared }-\text{o }\texttt{libmylib.so }\texttt{mylib.o} // Creates the shared library
- \$ \texttt{icc }-\text{mmic }\texttt{main.c libmylib.so} // Link the application

Static Libraries

Use \texttt{xiar} to create native static libraries

- \$ \texttt{icc }-\text{mmic }\texttt{-c }-\text{fpic }\texttt{mylib.c} // Creates \texttt{mylib.o} by default
- \$ \texttt{xiar }\texttt{crs libmylib.a mylib.o} // Creates the static library
- \$ \texttt{icc }-\text{mmic }\texttt{main.c libmylib.a} // Link the application
Parallel programming is the same on coprocessor and host
Language Extensions for Offload (pragmas)

Offload **pragma/directive** for data marshalling

- **#pragma offload <clauses>** in C/C++
  
  Offloads the following OpenMP block or Intel® Cilk™ Plus construct or function call or compound statement

- **!dir$ offload <clauses>** in Fortran
  
  Offloads the following OpenMP block or subroutine/function call

\[
\text{RESULT} = \text{FUNC}(A, B) \! \text{ but not } \text{RESULT} = \text{SCALE} \times \text{FUNC}(A, B)
\]

- **!dir$ offload begin <clauses>...**
  **!dir$ end offload** to offload other block of code

- Offloaded data must be scalars, arrays, bit-wise copyable structs (C/C++) or derived types (Fortran)
  
  - no embedded pointers or allocatable arrays
  - Excludes all but simplest C++ classes
  - Excludes most Fortran 2003 object-oriented constructs
  - All data types can be used within the target code
  - Data copy is explicit
Rules For Data Transfer (pragmas)

Automatically detected and transferred as INOUT

- Named arrays in lexical scope
- Scalars in lexical scope

User can override automatic transfer with explicit IN/OUT/INOUT clauses

Not automatically transferred

- Memory pointed to by pointers
  - This also needs a length parameter
- Global variables used in functions called within the offloaded construct
- User must specify IN/OUT/INOUT clauses
When are alloc_if and free_if clauses needed?

Needed for pointers or allocatable arrays

- Default is to always allocate and free memory for pointers that are within the lexical scope of the offload, not otherwise
  - use free_if(0) if you want to memory and data to persist until next offload
  - Need alloc_if(1) for globals that are not lexically visible and are NOCOPY
  - Or use alloc_if(expression) to make dependent on runtime data

Not needed for statically allocated data

- These are statically allocated and persistent on the coprocessor, even for arrays that are not lexically visible or have a NOCOPY clause.

Syntax:

- #pragma offload nocopy(myptr:length(n):alloc_if(expression))
- !DIR$ OFFLOAD IN(FPTR:length(n):free_if(.false.))
  (or a logical expression in Fortran)
Suggestion: Simplify use of alloc_if and free_if

For Readability define macros (unless condition is variable)

- `#define ALLOC alloc_if(1)`
- `#define FREE free_if(1)`
- `#define RETAIN free_if(0)`
- `#define REUSE alloc_if(0)`

```c
#pragma offload target(mic) in(p:length(l))
...`

- Allocate and do not free
  ```c
  #pragma offload target(mic) in(p:length(l) ALLOC RETAIN)
  ...`

- Reuse memory allocated above and do not free
  ```c
  #pragma offload target(mic) in(p:length(l) REUSE RETAIN)
  ...`

- Reuse memory allocated above and free
  ```c
  #pragma offload target(mic) in(p:length(l) REUSE FREE)
  ```
Data Transfer: non-shared memory

- Buffer
- CPU
- input1
- input2
- output

PCIe
Transfer to
coprocessor

- Buffer
- Coprocessor
- input1
- input2
- output
Memory Regions

A: COI buffer (4KB pages). All offloaded data (except B below) arrive here, then get re-copied to region C. Buffer size is dynamic (from 13.0 beta)

B: 2MB buffers/pages. C/C++ & Fortran pointer variables with size \( \text{MIC\_USE\_2MB\_BUFFERS} \) (off if unset) grows and shrinks dynamically. Data not recopied

C: dynamically allocated memory (malloc, ALLOCATE, ...) grows and shrinks dynamically with malloc, free,..

D: statically allocated data (.text, .data, .bss segments) size program-dependent but fixed.
Support for Multiple Coprocessors

```c
#pragma offload target(mic [ :<expr> ] ) ...

coprocessor # = <expr> % number_of_devices
```

Code must run on coprocessor #, aborts if not available (counts from 0)

If -1, runtime chooses coprocessor, aborts if not available

If not present, runtime chooses coprocessor or runs on host if none available

**APIs:**

- `#include offload.h` (C/C++); `USE MIC_LIB` (Fortran)
  - `int _Offload_number_of_devices()` (C/C++)
  - `result = OFFLOAD_NUMBER_OF_DEVICES()` (Fortran)
    - Returns # of coprocessors installed, or 0 if none
  - `int _Offload_get_device_number()` (C/C++)
  - `result = OFFLOAD_GET_DEVICE_NUMBER()` (Fortran)
    - Returns coprocessor number where executed, (-1 for CPU)
    - Can use to share work explicitly by coprocessor number
Asynchronous Offload

New synchronization clauses  SIGNAL(&x) and WAIT(&x)

- Argument is a unique address (usually of the data being transferred)

Data:

- #pragma offload_transfer target(mic:n) IN(....) signal(&s1)
  - Standalone data offload
- #pragma offload_wait target(mic:n) wait(&s1)
  - Standalone synchronization, host waits for transfer completion (blocking)

Computation:

- #pragma offload target(mic:n) wait(&s1) signal(&s2)
  - Offload computation when data transfer has completed
  - Computation on host then continues in parallel
- #pragma offload_wait target(mic:n) wait(&s2)
  - Host waits for signal that offload computation completed

There is also a non-blocking API to test signal value
Asynchronous Offload Example

```c
float *T;
int    N_OFFLOAD = 2*NTOT/3;

#pragma offload target(mic:0) in(T:length(N_OFFLOAD)) ... out(Result:length(N_OFFLOAD)) \ 
    signal (&T)
{
#pragma omp parallel for
    for(int opt = 0; opt < N_OFFLOAD; opt++) {
        ...  // do first 2/3 of work on coprocessor
    }

#pragma omp parallel for
    for(int opt = N_OFFLOAD; opt < NTOT; opt++) {
        ...
            // do remainder of work on host
    }

    //  synchronization before continuing on host using results of offload
    #pragma offload_wait target(mic:0) wait (&T)

    //  easily extended to offload work to multiple coprocessors, using different signals
```
Compiler Usage and Output Files

Offload compiler is invoked automatically if the host compiler detects any offload language extensions (keywords or pragmas) in the source

- This can be prevented by –no-offload

Offload options set by `-offload-option,mic,<tool>,"<option list>"`

where `<tool>` is compiler, as or ld.

e.g. `-offload-option,mic,compiler,"-vec-report2"

```
icc -c a.c creates a.o aMIC.o  (aMIC.o is generated behind the scenes)
xiar ... a.o b.o creates lib.a and libMIC.a
        (libMIC.a is generated behind the scenes)
xild ... lib.a c.o creates a.out
icc ... a.c b.c creates a.out
```

coprocessor executable is embedded inside a.out

ifort behaves the same as icc
Compiler Usage

Most compiler options for the host carry over to the offload

- Including preprocessor macro definitions
- -openmp is **not** enabled by default, but needed for OpenMP offloads
  - Else OpenMP directive is not recognized
  - Some parts of the OpenMP runtime may be linked by default for offload builds, but not for native

These defaults may be overridden, e.g.

```
icc -vec-report0 -offload-option,mic,compiler,"-vec-report2" a.c
```

(generates vectorization report for the offload without generating the corresponding report for the host)

- -opt-report-phase=offload will report which variables are offloaded
- -offload-attribute-target=mic flag all global variables and functions for offload
  (roughly like setting the offload attribute wherever it is allowed)
Preprocessor Macros

__INTEL_OFFLOAD

- Set automatically unless disabled by –no-offload (or –mmic)
- Set for the host compilation but not the target (coprocessor) compilation
- Use to protect code on the host that is specific for offload
  e.g. `omp_num_set_threads_target()` family of APIs
  but must remember to set –no-offload for host-only builds

__MIC__

- NOT set for host compilation in an offload build
- Set automatically for target (coprocessor) compilation in offload build
- Also set automatically when building native coprocessor application
- Use to protect code that is compiled & executed only on coprocessor
  e.g. `_mm512` intrinsics
Intel® MIC-specific environment variables

<table>
<thead>
<tr>
<th>MIC Env Variable</th>
<th>Default Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIC_ENV_PREFIX</td>
<td>string</td>
<td>none</td>
<td>Environment variables (except those below) are stripped of this prefix and underscore sent to coprocessor. Often set to “MIC”.</td>
</tr>
<tr>
<td>MIC_&lt;card #&gt;_ENV</td>
<td>string</td>
<td>none</td>
<td>List of environment variables to set on card #</td>
</tr>
<tr>
<td>MIC_LD_LIBRARY_PATH</td>
<td>string</td>
<td>set by compiler vars script</td>
<td>Search paths for coprocessor shared libraries</td>
</tr>
<tr>
<td>MIC_USE_2MB_BUFFERS</td>
<td>integer B/K/M/G/T</td>
<td>Don’t use</td>
<td>Use 2MB pages for pointer data where (size &gt; MIC_USE_2M_BUFFERS)</td>
</tr>
<tr>
<td>MIC_STACKSIZE</td>
<td>integer B/K/M/G/T</td>
<td>12M</td>
<td>Main thread stack size limit for pthreads</td>
</tr>
<tr>
<td>MIC_PROXY_IO</td>
<td>integer</td>
<td>1 (enabled)</td>
<td>Proxy coprocessor I/O to host (stdout, stderr)</td>
</tr>
<tr>
<td>MIC_PROXY_FS_ROOT</td>
<td>string</td>
<td>none</td>
<td>Host root directory for proxy file I/O for COI</td>
</tr>
<tr>
<td>OFFLOAD_REPORT</td>
<td>integer</td>
<td>none</td>
<td>1 Report execution time on host and coprocessor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 Also reports on data transfers between host and coprocessor</td>
</tr>
</tbody>
</table>
SIMD Data Types for Intel® MIC Architecture

- **16x floats**
- **8x doubles**
- **16x 32-bit integers**
- **8x 64-bit integers**
- **64x 8-bit bytes**
- **32x 16-bit shorts**
Support for SIMD Parallelism

For good performance, it’s not sufficient to use all the cores, you need to use the 512 bit SIMD registers and instructions

Vector classes and intrinsics are supported for C/C++
• See micvec.h and zmmmintrin.h in the include/mic directory
• Just include <immintrin.h>, the compiler takes care of the rest

Auto-vectorization for Intel® MIC architecture works just like for Intel® SSE or Intel® AVX on the host
• Data alignment should be to 512 bits, instead of 128 or 256

Because of the greater SIMD width, vectorization is even more important on Intel® MIC architecture than on Intel® Xeon® processors. The Intel® compiler now supports:

Explicit Vector Programming
• Via Intel® Cilk™ Plus language extensions
• Via the SIMD constructs from OpenMP 4.0
Auto-vectorization

The vectorizer for Intel® MIC architecture works just like for Intel® SSE or Intel® AVX on the host, for C, C++ and Fortran

- Enabled at default optimization level (-O2)
- Data alignment should be to 64 bytes, instead of 16 (see later)
- More loops can be vectorized, because of masked vector instructions, gather/scatter instructions, fused multiply-add (FMA)
- Try to avoid 64 bit integers (except as addresses)

Vectorized loops may be recognized by:

- Vectorization and optimization reports (simplest),
  - e.g. -vec-report2 or -opt-report-phase hpo
- Unmasked vector instructions (there are no separate scalar instructions; masked vector instructions are used instead)
- Gather & scatter instructions
- Math library calls to libsvml
Vectorization Reports

By default, both host and target compilations may generate messages for the same loop, e.g.

```
icc -vec-report2 test_vec.c
```

```
test_vec.c(10): (col. 1) remark: LOOP WAS VECTORIZED.
test_vec.c(10): (col. 1) remark: *MIC* LOOP WAS VECTORIZED.
```

To get a vectorization report for the offload target compilation, but not for the host compilation:

```
icc –vec-report0 –offload-option,mic,compiler,”-vec-report2” test_vec.c
```

```
test_vec.c(10): (col. 1) remark: *MIC* LOOP WAS VECTORIZED.
test_vec.c(20): (col. 1) remark: *MIC* loop was not vectorized: existence of vector dependence.
test_vec.c(20): (col. 1) remark: *MIC* PARTIAL LOOP WAS VECTORIZED.
```
Common vectorization messages

“Loop was not vectorized” because:

• “Low trip count”

• “Existence of vector dependence”
  • Possible dependence of one loop iteration on another, e.g.
    
    ```
    for (j=1; j<MAX; j++)  a[j] = a[j] + c * a[j-n];
    ```

• "vectorization possible but seems inefficient"

• “Not Inner Loop”

It may be possible to overcome these using switches, pragmas, source code changes or explicit vector programming
Vector instructions

Compile with –S to see assembly code (if you like)

A vectorized loop contains instructions like

```
vfmadd213ps %zmm23, %zmm8, %zmm2   # fma instruction
vaddps       %zmm25, %zmm2, %zmm0   # single precision add
```

In a scalar loop, these instructions will be masked, e.g.

```
vfmadd213ps %zmm17, %zmm20, %zmm1{%k1}
vaddps       %zmm23, %zmm1,   %zmm0{%k1}
```

Example of vectorized math function for Intel® MIC architecture:

```
call   __svml_sinf16       # calculates sin(x) for 16 floats
```

```
call   __svml_sinf16_mask
```
Requirements for Auto-Vectorization

Innermost loop of nest (a few simple exceptions)
Straight-line code (masked assignments OK)

Avoid:
  • Function/subroutine calls (unless inlined or vector)
  • Data-dependent loop exit conditions
    • Iteration count should be known at entry to loop
  • Loop carried data dependencies (Reduction loops OK)
  • Non-contiguous data (indirect addressing; non-unit stride)
    • Inefficient
  • Inconsistently aligned data

Directives/pragmas can help:
  • #pragma ivdep ...... ignore potential dependencies
  • #pragma vector always ignore efficiency heuristics
  • aligned assume data are aligned
  • Compiler can generate runtime alignment and dependency tests, for simple loops only, (but less efficient)

See http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/
## Vectorizable math functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>ceil</td>
<td>fabs</td>
<td>round</td>
</tr>
<tr>
<td>acosh</td>
<td>cos</td>
<td>floor</td>
<td>sin</td>
</tr>
<tr>
<td>asin</td>
<td>cosh</td>
<td>fmax</td>
<td>sinh</td>
</tr>
<tr>
<td>asinh</td>
<td>erf</td>
<td>fmin</td>
<td>sqrt</td>
</tr>
<tr>
<td>atan</td>
<td>erfc</td>
<td>log</td>
<td>tan</td>
</tr>
<tr>
<td>atan2</td>
<td>erfinv</td>
<td>log10</td>
<td>tanh</td>
</tr>
<tr>
<td>atanh</td>
<td>exp</td>
<td>log2</td>
<td>trunc</td>
</tr>
<tr>
<td>cbrt</td>
<td>exp2</td>
<td>pow</td>
<td></td>
</tr>
</tbody>
</table>

Also float versions, such as `sinf()`

Uses short vector math library, `libsvml`
Problems with Pointers

Hard for compiler to know whether arrays or pointers might be aliased (point to the same memory location)

- Aliases may hide dependencies that make vectorization unsafe

In simple cases, compiler may generate vectorized and unvectorized loop versions, and test for aliasing at runtime.

Otherwise, compiler may need help:

- `-fargument-noalias` & similar switches
- Use Intel® Cilk™ Plus array notation
- "restrict" keyword with `-restrict` or `-std=c99` or by inlining
  - and now `__restrict__`
- `#pragma ivdep` asserts no potential dependencies
- Compiler still checks for proven dependencies
- `#pragma simd` asserts no dependencies, period (see later)

```c
void saxpy(float *x, float *y, float *restrict z, float *a, int n) {
  #pragma ivdep
  for (int i=0; i<n; i++) z[i] = *a*x[i] + y[i];
}
```
Intel® Compilers:
some useful loop optimization pragmas/directives

- IVDEP 
  ignore vector dependency
- LOOP COUNT 
  advise typical iteration count(s)
- UNROLL 
  suggest loop unroll factor
- DISTRIBUTE POINT 
  advise where to split loop
- VECTOR 
  vectorization hints
  - Aligned 
    assume data is aligned
  - Always 
    override cost model
  - Nontemporal 
    advise use of streaming stores
- NOVECTOR 
  do not vectorize
- NOFUSION 
  do not fuse loops
- INLINE/FORCEINLINE  
  invite/require function inlining
- SIMD ASSERT 
  “vectorize or die” 😊 (see later)

Use where needed to help the compiler, guided by optimization reports
How to Align Data (C/C++)

Allocate memory on heap aligned to n byte boundary:

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

Alignment for variable declarations:

```c
__attribute__((aligned(n))) var_name or __declspec(align(n)) var_name
```

**And tell the compiler...**

```c
#pragma vector aligned
```

- 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

```c
__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
• 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
Vectorization and Intel® Cilk™ Plus

Vectorization is so important ➞ consider explicit vector programming

Intel® Cilk™ Plus array notation

• Compiler can assume LHS does not alias RHS (unlike Fortran)
  \[ r[n:n] = \text{sqrtf}(x[0:n] \times x[0:n] + y[0:n] \times y[0:n]) \]

and simd-enabled functions

• Allow vectorization over function calls, without inlining

\[ \text{__attribute__((vector)) float myfun(float a,float x,float y)} \{\ldots\} \]
  \[ z[:] = \text{myfunl(a[:], b[:], c[:])} \]

#pragma simd

• Directs compiler to vectorize if at all possible
• Overrides all dependencies and cost model
  • More aggressive than pragmas ivdep and vector always
• Semantics modeled on OpenMP parallel pragmas
  • Private and reduction clauses required for correctness
Explicit Vector Programming:
Intel® Cilk™ Plus Array notation

Array notation asks the compiler to vectorize
- asserts this is safe (for example, \( x < 0 \))
- Improves readability

```c
void addit(double* a, double* b, int m, int n, int x)
{
    for (int i = m; i < m+n; i++)
    {
        a[i] = b[i] + a[i-x];
    }
}
```

```c
void addit(double* a, double * b, int m, int n, int x)
{
    // I know x<0
    a[m:n] = b[m:n] + a[m-x:n];
}
```

loop was not vectorized: existence of vector dependence.

LOOP WAS VECTORIZED.
Explicit Vector Programming:
Intel® Cilk™ Plus pragma example

Using  #pragma simd (C/C++)  or  !DIR$ SIMD (Fortran)
or  #pragma omp simd  (OpenMP 4.0)

void addit(double* a, double* b, int m, int n, int x)
{
    for (int i = m; i < m+n; i++) {
        a[i] = b[i] + a[i-x];
    }
}

loop was not vectorized:
existence of vector dependence.

void addit(double* a, double* b, int m, int n, int x)
{
    #pragma simd  // I know x<0
    for (int i = m; i < m+n; i++) {
        a[i] = b[i] + a[i-x];
    }
}

SIMD LOOP WAS VECTORIZED.

• Use when you **KNOW** that a given loop is safe to vectorize
• The Intel Compiler will vectorize if at all possible
  (will ignore dependency or efficiency considerations)
• Minimizes source code changes needed to enforce vectorization
Clauses for SIMD directives

The programmer (i.e. you!) is responsible for correctness

- Just like for race conditions in OpenMP loops

Available clauses (both OpenMP and Intel versions)

PRIVATE |  
LASTPRIVATE | --- like OpenMP
REDUCTION |  
COLLAPSE | (OpenMP 4.0 only; for nested loops)
LINEAR (additional induction variables)
SAFELEN (OpenMP 4.0 only)
VECTORLENGTH (Intel only)
ALIGNED (OpenMP 4.0 only)
ASSERT (Intel only; default for OpenMP 4.0)
SIMD-enabled Function

Compiler generates vector version of a scalar function that can be called from a vectorized loop:

```c
__attribute__((vector(uniform(y, xp, yp))))
float func(float x, float y, float xp, float yp) {
    float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);
    denom = 1./sqrtf(denom);
    return denom;
}
```

```
#pragma simd  private(x)  reduction(+:sumx)
for (i=1; i<nx; i++)  {
    x = x0 + (float)i * h;
    sumx = sumx + func(x, y ,xp, yp);
enddo
```

These clauses are required for correctness, just like for OpenMP.

SIMD LOOP WAS VECTORIZED.

y, xp and yp are constant, x can be a vector
Clauses for SIMD-enabled Functions

__attributes__((vector)) (Intel)
#pragma omp declare simd (OpenMP 4.0)

Available clauses (both OpenMP and Intel versions)

- **LINEAR** (additional induction variables)
- **UNIFORM** (arguments that are same for each SIMD lane)
- **REDUCTION**
- **PROCESSOR** (Intel)
- **VECTORLENGTH** (Intel)
- **MASK / NOMASK** (Intel)
- **INBRANCH / NOTINBRANCH** (OpenMP 4.0)
- **SIMDLEN** (OpenMP 4.0)
- **ALIGNED** (OpenMP 4.0)
SIMD Summary

The importance of SIMD parallelism is increasing

• Moore’s law leads to wider vectors as well as more cores
• Don’t leave performance “on the table”
• Be ready to help the compiler to vectorize, if necessary
  • With compiler directives and hints
  • Using information from vectorization and optimization reports
  • With explicit vector programming
  • Use Intel® VTune™ Amplifier XE to find the best places (hotspots) to focus your efforts - see accompanying session
• No need to re-optimize vectorizable code for new processors
  • Typically a simple recompilation
Prefetching - automatic

Compiler prefetching is on by default for the Intel® Xeon Phi™ coprocessor at -O2 and above

- Prefetches issued for regular memory accesses inside loops
- But not for indirect accesses a[index[i]]
- More important for Intel Xeon Phi coprocessor (in-order) than for Intel® Xeon® processors (out-of-order)
- Very important for apps with many L2 cache misses

Use the compiler reporting options to see detailed diagnostics of prefetching per loop

- opt-report-phase hlo -opt-report 3 e.g.

Total #of lines prefetched in main for loop at line 49=4
Using noloc distance 8 for prefetching unconditional memory reference in stmt at line 49
Using second-level distance 2 for prefetching spatial memory reference in stmt at line 50

- opt-prefetch=n (4 = most aggressive) to control
- opt-prefetch=0 or -no-opt-prefetch to disable
- opt-prefetch-distance =<n1> [,<n2>] to tune how far ahead to prefetch
Prefetching - manual

Use intrinsics

```c
_mm_prefetch((char *) &a[i], hint);
```

See xmmmintrin.h for possible hints (for L1, L2, non-temporal, ...)

```c
MM_PREFETCH(A, hint)
```

for Fortran

• But you have to figure out and code how far ahead to prefetch
• Also gather/scatter prefetch intrinsics, see zmmmintrin.h and compiler
  user guide, e.g. _mm512_prefetch_i32gather_ps

Use a pragma / directive (easier):

```c
#pragma prefetch a [:hint[:distance]]
```

`!DIR$ PREFETCH A, B, ...`

• You specify what to prefetch, but can choose to let compiler figure out
  how far ahead to do it.

Hardware L2 prefetcher is also enabled by default

• If software prefetches are doing a good job,
  then hardware prefetching does not kick in
Streaming Stores

Write directly to memory bypassing cache
- for “nontemporal” data that are not read and will not be reused
- avoids “read for ownership” to get line into cache
  - Reduces memory bandwidth requirements
  - Keeps cache available for useful work, avoids “pollution”

```c
#pragma vector nontemporal (v1, v2, ...) hint to compiler
```

- No Streaming Stores:
  448 Bytes read/write per iteration
- With Streaming Stores:
  320 Bytes read/write per iteration
- -vec-report6 shows what the compiler did

```c
for (int chunk = 0; chunk < OptPerThread; chunk += CHSIZE) {
    #pragma simd vectorlength(CHSIZE)
    #pragma vector aligned
    #pragma vector nontemporal (CallResult, PutResult)
    for (int opt = chunk; opt < (chunk+CHSIZE); opt++) {
        float T = OptionYears[opt];
        float X = OptionStrike[opt];
        float S = StockPrice[opt];
        …. 
        CallVal = S * CNDD1 - XexpRT * CNDD2;
        PutVal = CallVal + XexpRT - S;
        CallResult[opt] = CallVal;
        PutResult[opt] = PutVal;
    }
}
```

bs_test_sp.c(215): (col. 4) remark: vectorization support: streaming store was generated for CallResult.
bs_test_sp.c(216): (col. 4) remark: vectorization support: streaming store was generated for PutResult.
More reports

-opt-report-phase hlo
summarizes loop optimizations including loop interchange, fusion, distribution, unrolling, multi-versioning, cache blocking, prefetching, etc.

-opt-report-phase ipo_inl
summarizes function inlining

-opt-report-phase offload
gives a compile time summary of which data are copied to and from the coprocessor

OFFLOAD_REPORT=2 (environment variable on host)
gives run-time summary of data copied to and from the coprocessor and computation time on the coprocessor

Main optimization opportunity for the data offload:
Don’t transfer data that you don’t need! (e.g., copy in but not out)
OpenMP on the Coprocessor

The basics work just like on the host CPU

- For both native and offload models
- Need to specify -openmp

There are 4 hardware thread contexts per core

- Need at least 2 x ncore threads for good performance
- For all except the most memory-bound workloads
- Often, 3x or 4x (number of available cores) is best
- Very different from hyperthreading on the host!
- -opt-threads-per-core=n advises compiler how many threads to optimize for
- If you don’t saturate all available threads, be sure to set KMP_AFFINITY to control thread distribution
OpenMP defaults

OMP_NUM_THREADS defaults to

• 1 x ncore for host  (or 2x if hyperthreading enabled)
• 4 x ncore for native coprocessor applications
• 4 x (ncore-1) for offload applications
  • one core is reserved for offload daemons and OS (typically the highest numbered)
• Defaults may be changed via environment variables or via API calls on either the host or the coprocessor
Target OpenMP environment (offload)

Use target-specific APIs to set for coprocessor target only, e.g.

```c
omp_set_num_threads_target()  // (called from host)
omp_set_nested_target()       // etc.
```

- Protect with `#ifdef __INTEL_OFFLOAD`, undefined with `-no-offload`
- Fortran: `USE MIC_LIB` and `OMP_LIB`
  
  ```c
  C: #include <offload.h>
  ```

Or define coprocessor – specific versions of env variables using

```c
MIC_ENV_PREFIX=PHI       // (no underscore)
```

- Values on coprocessor **no longer default to values on host**
- Set values specific to coprocessor using

```c
export PHI_OMP_NUM_THREADS=120  // (all coprocessors)
export PHI_2_OMP_NUM_THREADS=180  // for coprocessor #2, etc.
export PHI_3_ENV="OMP_NUM_THREADS=240|KMP_AFFINITY=balanced"
```
Stack Sizes for Coprocessor

For the main thread, (thread 0), default stack limit is 12 MB
- In offloaded functions, stack is used for local or automatic arrays and compiler temporaries
- To increase limit, export MIC_STACKSIZE (e.g. =100M)
  - default unit is K (Kbytes)
- For native apps, use ulimit –s (default units are Kbytes)

For worker threads: default stack size is 4 MB
- Space only needed for those local variables or automatic arrays or compiler temporaries for which each thread has a private copy
- To increase limit, export OMP_STACKSIZE=10M (or as needed)
- Or use dynamic allocation (may be less efficient)

Typical error message if stack limits exceeded:
  offload error: process on the device 0 was terminated by SEGFAULT
Thread Affinity Interface

Allows OpenMP threads to be bound to physical or logical cores

• export environment variable KMP_AFFINITY=
  • physical use all physical cores before assigning threads to other logical cores (other hardware thread contexts)
  • compact assign threads to consecutive h/w contexts on same physical core (e.g. to benefit from shared cache)
  • scatter assign consecutive threads to different physical cores (e.g. to maximize access to memory)
  • balanced blend of compact & scatter (currently only available for Intel® MIC Architecture)

• Helps optimize access to memory or cache
• Particularly important if all available h/w threads not used
  • else some physical cores may be idle while others run multiple threads

• See compiler documentation for (much) more detail
Example – share work between coprocessor and host using OpenMP

```c
omp_set_nested(1);
#pragma omp parallel private(ip)
{
    #pragma omp sections
    {
        #pragma omp section
        /* use pointer to copy back only part of potential array,
           to avoid overwriting host */
        #pragma omp offload target(mic) in(xp) in(yp) in(zp) out(ppot:length(np1))
        #pragma omp parallel for private(ip)
        for (i=0; i<np1; i++) {
            ppot[i] = threed_int(x0, xn, y0, yn, z0, zn, nx, ny, nz, xp[i], yp[i], zp[i]);
        }
    }
    #pragma omp section
    #pragma omp parallel for private(ip)
    for (i=0; i<np2; i++) {
        pot[i+np1] = threed_int(x0, xn, y0, yn, z0, zn, nx, ny, nz, xp[i+np1], yp[i+np1], zp[i+np1]);
    }
}
```

Top level, runs on host
Runs on coprocessor
Runs on host
Debugging with OpenMP

Test your OpenMP threaded app on the host before moving to the coprocessor

Debug with –O0 –openmp

- Unlike most other optimizations, OpenMP threading is not disabled at -O0
- Compare to running with OMP_NUM_THREADS=1
- Compare to building with –O0 –openmp-stubs (and –auto for Fortran)

If debugging with print statements

- print out the thread number with omp_get_thread_num()
- the internal I/O buffers are threadsafe (with –openmp), but the order of print statements from different threads is undetermined.
- Order of print statements from host and from coprocessor is undefined

Use Intel® Inspector XE on the host to detect race conditions and other threading and memory errors
Floating-Point Behavior on Intel® Xeon Phi™ Coprocessors

Trapping of floating-point exceptions in vector instructions is not supported

The bits of the SIMD floating-point control word that mask/unmask floating-point exceptions are protected

• If you try to unmask exceptions, your app will seg fault
• Unmasking by compiler switches such as –fp-trap or –fpe0 is disabled for native builds or for the target part of an offload build
• The exception flags still get set, and you can test on these
• Otherwise, the computation just continues with QNaNs, infinities, etc.
• -fp-model except or –fp-model strict preserves exception semantics
  • Generates x87 instead of vector instructions, big performance impact
  • May be useful for debugging

Denormals are supported

• Needs -no-ftz or –fp-model precise (like on host)
Floating-Point Behavior on Intel® Xeon Phi™ Coprocessors

-fp-model fast=2 enables some more aggressive optimizations

• Faster inlined versions of some math functions
  • May not give standard behavior for extreme or exceptional arguments

Floating-point results on Intel® Xeon Phi™ may not be bit-for-bit identical to results obtained on Intel® Xeon® processors

• Most common cause is fused multiply-add (FMA) instructions
  • Not disabled by –fp-model precise
  • Can disable for testing with –no-fma
    • With some impact on performance

• Implementation of math functions might also differ

• To get close, try –fp-model precise –no-fma
  • But parallel reductions or math functions may still cause differences
Questions?
Additional Resources

http://software.intel.com/mic-developer
- Developer’s Quick Start Guide
- Programming Overview
- User Forum at


Intel® Composer XE 2013 SP1 for Linux* User and Reference Guides

Intel Premier Support  https://premier.intel.com
Legal Disclaimer & Optimization Notice

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

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

Copyright © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

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
Will it vectorize?

Assume a, b and x are known to be independent.

```c
for (j=1; j<MAX; j++)  a[j]=a[j-n]+b[j];

for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[i];

for (int j=0; j<SIZE; j++)  {
    for (int i=0; i<SIZE; i++)   b[i] += a[i][j] * x[j];
}

for (int i=0; i<SIZE; i++)  b[i] += a[i] * x[index[i]];

for (j=1; j<MAX; j++)  sum = sum + a[j]*b[j]

for (int i=0; i<length; i++)   {
    float s = b[i]*b[i] – 4.f*a[i]*c[i];
    if ( s >= 0 ) x2[i] = (-b[i]+sqrt(s))/(2.*a[i]);
}
```
Will it vectorize? Answers

1) Vectorizes if \( n \leq 0 \); doesn’t vectorize if \( n > 0 \) and small; may vectorize if \( n \geq \) number of elements in a vector register

2) Unlikely to vectorize because of non-unit stride (inefficient)

3) Doesn’t vectorize because of non-unit stride, unless compiler can first interchange the order of the loops. (Here, it can)

4) Doesn’t vectorize because of indirect addressing (non-unit stride), would be inefficient. If \( x[index[i]] \) appeared on the LHS, this would also introduce potential dependency (\( index[i] \) might have the same value for different values of \( i \))

5) Reductions such as this will vectorize. The compiler accumulates a number of partial sums (equal to the number of elements in a vector register), and adds them together at the end of the loop. gcc needs –ffast-math in addition.

6) This will vectorize. Neither “if” masks nor most simple math intrinsic functions prevent vectorization. But with SSE, the sqrt is evaluated speculatively. If FP exceptions are unmasked, this may trap if \( s<0 \), despite the if clause. With AVX, there is a real hardware mask, so the sqrt will never be evaluated if \( s<0 \), and no exception will be trapped.