Xeon Phi — MIC

- Xeon Phi = first product of Intel’s Many Integrated Core (MIC) architecture
- Co-processor
  - PCI Express card
  - Stripped down Linux operating system
- Dense, simplified processor
  - Many power-hungry operations removed
  - Wider vector unit
  - Wider hardware thread count
- Lots of names
  - Many Integrated Core architecture, aka MIC
  - Knights Corner (code name)
  - Intel Xeon Phi Co-processor SE10P (product name)
Xeon Phi — MIC

- Leverage x86 architecture (CPU with many cores)
  - x86 cores that are simpler, but allow for more compute throughput
- Leverage existing x86 programming models
- Dedicate much of the silicon to floating point ops
- Cache coherent
- Increase floating-point throughput
- Strip expensive features
  - out-of-order execution
  - branch prediction
- Widen SIMD registers for more throughput
- Fast (GDDR5) memory on card
MIC Architecture

- Many cores on the die
- L1 and L2 cache
- Bidirectional ring network for L2
- Memory and PCIe connection
Knights Corner Core

---

X86 specific logic < 2% of core + L2 area

George Chrysos, Intel, Hot Chips 24 (2012):
Vector Processing Unit

- PPF
- PF
- DO
- D1
- D2
- E
- WB
- D2
- E
- VC1
- VC2
- V1-V4
- WB

DEC
VPU
RF
3R, 1W
LD
EMU
ST
Mask
RF
Scatter
Gather

Vector ALUs
- 16 Wide x 32 bit
- 8 Wide x 64 bit
- Fused Multiply Add

George Chrysos, Intel, Hot Chips 24 (2012):
Speeds and Feeds

- **Processor**
  - ~1.1 GHz
  - 61 cores
  - 512-bit wide vector unit
  - 1.074 TF peak DP

- **Data Cache**
  - L1 32KB/core
  - L2 512KB/core, 30.5 MB/chip

- **Memory**
  - 8GB GDDR5 DRAM
  - 5.5 GT/s, 512-bit*

- **PCIe**
  - 5.0 GT/s, 16-bit
Advantages

- Intel’s MIC is based on x86 technology
  - x86 cores w/ caches and cache coherency
  - SIMD instruction set

- Programming for MIC is similar to programming for CPUs
  - Familiar languages: C/C++ and Fortran
  - Familiar parallel programming models: OpenMP & MPI
  - MPI on host and on the coprocessor
  - Any code can run on MIC, not just kernels

- Optimizing for MIC is similar to optimizing for CPUs
  - “Optimize once, run anywhere”
  - Our early MIC porting efforts for codes “in the field” are frequently doubling performance on Sandy Bridge.
What we at TACC like about Phi

- Intel’s MIC is based on x86 technology
  - x86 cores w/ caches and cache coherency
  - SIMD instruction set

- Programming for Phi is similar to programming for CPUs
  - Familiar languages: C/C++ and Fortran
  - Familiar parallel programming models: OpenMP & MPI
  - MPI on host and on the coprocessor
  - Any code can run on MIC, not just kernels

- Optimizing for Phi is similar to optimizing for CPUs
  - “Optimize once, run anywhere”
  - Our early Phi porting efforts for codes “in the field” have doubled performance on Sandy Bridge.
Will My Code Run on Xeon Phi?

• Yes

• ... but that’s the wrong question
  – Will your code run *best* on Phi?, or
  – Will you get great Phi performance without additional work? (The answer is most likely **NO**
Early Phi Programming Experiences at TACC

• Codes port easily
  – Minutes to days depending mostly on library dependencies

• Performance can require real work
  – While the software environment continues to evolve
  – Getting codes to run *at all* is almost too easy; really need to put in the effort to get what you expect

• Scalability is pretty good
  – Multiple threads per core is really important
  – Getting your code to vectorize is really important
LBM Example

- Lattice Boltzmann Method CFD code
  - Carlos Rosales, TACC
  - MPI code with OpenMP
- Finding all the right routines to parallelize is critical
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
For Both Offload and Native Models

Tools
MKL
OpenMP
MPI
TBB
OpenCL
Intel® Cilk™ Plus
C++/Ftn

Host Executable

Parallel Compute

PCle

Coprocessor Executable

Parallel Compute

Heterogeneous Compute

Offload Directives (Data Marshalling)

Offload Keywords (Shared Virtual Memory)

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
  
  ```fortran
  RESULT = FUNC(A,B)  ! but not RESULT = SCALE * 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
Support for Multiple Coprocessors

```
#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
```
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.

```bash
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)
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**

*Other names and brands may be claimed as the property of others.*
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
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
- DISTRIBUTED 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
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
    
    \[
    \text{__attribute__((vector)) float myfun(float a, float x, float y)} \{...\}
    \]
    \[ z[:] = \text{myfunl(a[:], b[:], c[:])} \];

#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

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.

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

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 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];  
}  

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];  
}  

loop was not vectorized:
existence of vector dependence.  
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
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
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.

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: #include <offload.h>

Or define coprocessor – specific versions of env variables using

MIC_ENV_PREFIX=PHI (no underscore)

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

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"
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 (eg 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
Will it vectorize?

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

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.