Intel Xeon PHI Programming
LNCC Training

Dr Cédric Bourrasset

27-10-2016
Outline

- Introduction to Xeon PHI architecture (Knight Corner)

- Offload Compilation

- Optimization on KNC

- Introduction to the new Xeon PHI architecture (Knight Landing)
Introduction to Xeon PHI Architecture (Knight Corner)
Introduction
KNC architecture

- 61 in order cores at 1.2GHz (Ring interconnect)
  - 4 Hardware threads per core
  - Caches:
    - L1 32K I/D-cache
    - L2 512K Unified
  - 1 new vector unit 512bit

- 16GB GDDR5
  - 320GB/s theoretical
  - 180GB/s measurement

Theoretical Gflops = #cores*freq*#inst/cycle = 61*1.2*16 = 1171.2 Gflops
KNC core architecture

- Two pipelines:
  - Intel Pentium processor family-based scalar units
  - Pipelined one-per-clock scalar throughput (4 clock latency, hidden by round-robin scheduling of threads)

- 4 hardware threads per core

- New vector unit
  - 512-bit SIMD Instructions - not Intel SSE, MMX or Intel AVX
  - 32 512-bit wide vector registers

- Fully-coherent L1 and L2 caches
KNC overview

- **L1 Cache**
  - 32K I-cache per core
  - 32K D-cache per core
  - 3 cycles access
  - Up to 8 outstanding requests
  - Fully coherent

- **L2 Cache**
  - 512K Unified per core
  - 11 cycles best access
  - Up to 32 outstanding requests
  - Fully coherent

- **Memory**
  - GDDR5
  - 16 Memory channels (5.5 Gb/s each)
  - 8 GB up to 16 GB
  - 300 ns access
  - BW: 320 GB/s

- **Data Alignment**
  - 64B alignment
## User environment

<table>
<thead>
<tr>
<th>Compilers and libraries</th>
<th>TOOLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Compiler</td>
<td>Intel Debugger</td>
</tr>
<tr>
<td>Intel MPI Library</td>
<td>Intel Profiler (Vtune)</td>
</tr>
<tr>
<td>OpenMP 4.x</td>
<td>Standard runtime libraries and Gnu tools</td>
</tr>
<tr>
<td>Intel CILK Plus (Parallel Building Block)</td>
<td>micsmc</td>
</tr>
<tr>
<td>Intel Libraries (MKL)</td>
<td></td>
</tr>
</tbody>
</table>
Administrative environment

- **micinfo**
  provides information about host and coprocessor system configuration

- **micsmcc**
  monitoring and managing Intel Xeon Phi Coprocessors

- **miccheck**
  check mic configuration

- **micnativeloadex**
  executes native binary to Intel Xeon Phi Coprocessor

- **micctrl**
  system administrator tool
Monitoring the Intel Xeon Phi micsmc GUI
Parallel programming is the same on Xeon Phi and CPU.

**Three programming models:**

- **Offload:**
  - Compilation: No change
  - On Phi: Specific sections

- **Native**
  - Compilation: Add “-mmic”
  - On Phi: All

- **Symetric**
  - MPI Ranks are on Intel Xeon Phi Coprocessor AND Hosts
  - Require two compilations
Software architecture for offload mode
Software architecture for native mode

- **Linux* Host**
  - ssh or telnet connection to /dev/mic*

- **Intel® Xeon Phi™ Coprocessor**
  - Target-side “native” application
    - User code
    - Standard OS libraries plus any 3rd-party or Intel libraries
  - Virtual terminal session

**Intel® Xeon Phi™ Coprocessor Architecture** support libraries, tools, and drivers

**Intel® Xeon Phi™ Coprocessor communication and application-launching support**

PCI-E Bus

Linux* OS
Offload Compilation
Offload Programming Model

You need to

Embed your MIC sections/functions with directives.

You don’t need to

Setup/teardown, data transfer, synchronization, are managed automatically by compiler and runtime (Warning! Embedded pointers or allocatable arrays are not managed)
Programming Model

- You will add pragmas and new keywords to make sections of code run on the Intel Xeon Phi Coprocessor
  - Same way as adding OpenMP directives inside serial code

- Compiler will generate two binary codes for both target architectures
  - Binary will run with or without Intel Xeon Phi Coprocessor
  - You don’t need to add specials commands to transfer your code
Data transfer: 2 ways

► The host and the Phi don’t share physical or virtual memory space

► Explicit Copy
  – You have to designate variables that need to be copied between the host and the targets
  – Using pragmas
  – C/C++  
    #pragma offload target(mic) in(data:length(size))
  – Fortran  
    !dir$ offload target(mic) in(data:length(size))

► Implicit copy
  – You have to mark variables that need to be shared between host and Phi. Same variable can be used for both
  – Use keyword extension
  – Works with C/C++ extension Cilk Plus
Simple way to offload

At first offload, if Intel® MIC device is installed and available, MIC program is loaded

At each offload, if Intel MIC device is present, statement runs on device, else statement runs on CPU

At program termination, Intel MIC program is unloaded

```
/* a=b+g() on phi */
f() {
#pragma offload target(mic)
    a=b+g()
}

__attribute__((target(mic)))
g() {...}

/* g() on CPU */
h() {
    c=d+g()
}
```
Finer way to offload

- Specify data transfer and size

```c
#pragma offload target(mic : target_id) \
  in(all_Vals : length(MAXSZ)) \
  inout(numEs) out(E_vals : length(MAXSZ/2))
#pragma omp parallel for
  for (k=0; k < MAXSZ; k++) {
    if ( all_Vals[k] % 2 == 0 ) {
      E_vals[numEs] = all_Vals[k];
      numEs++;
    }
  }
```

- Specify MIC vars and functions

```c
__attribute__(( target(mic) )) int all_Vals;
```
## Compiler directives

### Pragma

<table>
<thead>
<tr>
<th>Pragma</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offload &lt;clause&gt; &lt;statement&gt;</td>
<td>Offload section of code</td>
</tr>
<tr>
<td><strong>attribute</strong>((target(mic)))</td>
<td>specifies MIC vars and function</td>
</tr>
<tr>
<td>offload_transfer &lt;clause&gt;</td>
<td>transfer data HOST &lt;-&gt; MIC</td>
</tr>
<tr>
<td>offload_wait(sig)</td>
<td>Wait for asynchronous offload</td>
</tr>
</tbody>
</table>

### Clause

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target(mic[:unit])</td>
<td>Select specific coprocessors</td>
</tr>
<tr>
<td>in(var-list modifiers)</td>
<td>Copy from host to coprocessor</td>
</tr>
<tr>
<td>out(var-list modifiers)</td>
<td>Copy from coprocessor to host</td>
</tr>
<tr>
<td>inout(var-list modifiers)</td>
<td>Copy host to coprocessor and back when offload completes</td>
</tr>
<tr>
<td>nocopy(var-list modifiers)</td>
<td>Data is local to target</td>
</tr>
</tbody>
</table>
## Compiler directives

<table>
<thead>
<tr>
<th>Offload modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length(N)</td>
<td>Copy N elements of pointer’s type</td>
</tr>
<tr>
<td>alloc_if(bool)</td>
<td>Allocate coprocessor space on this offload (default: TRUE)</td>
</tr>
<tr>
<td>free_if(bool)</td>
<td>Free coprocessor space at the end of this offload (default: TRUE)</td>
</tr>
<tr>
<td>align(N bytes)</td>
<td>Specify minimum memory alignment on coprocessor</td>
</tr>
<tr>
<td>alloc(array-slice) into (var-expr)</td>
<td>Enables partial array allocation and data copy into other vars &amp; ranges</td>
</tr>
</tbody>
</table>
Asynchronous offload

- Signal: Asynchronous offloading and transfer
  - Default: CPU waits for offload/transfer complete.
  - Signal and wait specifiers allow CPU to continue executing after the offload code block.

```c
#pragma offload target(mic[:#id]) ... signal(tag_list)
#pragma offload target(mic[:#id]) ... wait(tag_list)
#pragma offload_wait ... wait(tag_list)
```
Share work between host and a KNC Card (example)

```c
#define N 10000
__attribute__((target(mic:0))) // work function as targeted for offload
void work(int knt, int M, int N, int *a);

int main(){
    int sig1, i, knt=1, a[N], NS, NE; // declare signal var : sig1
    for(i=0; i< 10);{
        do{
            NSm=0; NEm=N/2;
            #pragma offload target(mic:0) signal(&sig1)
            work(knt,NSm,NEm, N,a);  // PHI and send signal sig1 at the end
            NSc=N/2;
            NEC=N;
            work(knt,NSc,NEc, N,a); // run on CPU at the same time
            #pragma offload_wait target(mic:0) wait(&sig1) // wait signal sig1
            knt=knt+1;
        }while (knt < 10);
    }
}
Share work between host and a KNC Card (example)
Share work between a host and two KNC (example)

```c
#define N 10000
__attribute__((target(mic:0))) //work function as targeted for offload
void work(int knt, int M, int N, int *a);

int main(){
    int sig1, int sig2, i, knt=1, a[N], NS, NE; //declare signal var : sig1,sig2
    for(i=0;i<10){
        do{
            NSm=0; NEm=N/3;
            #pragma offload target(mic:0) signal(&sig1)
            work(knt,NSm,NEm, N,a); //PHI and send signal sig1 at the end
        }while (knt < 10);
    }

    NSm2=(N/3); NEm2=N*(2/3);
    #pragma offload target(mic:1) signal(&sig2)
    work(knt,NSm2,NEm2, N,a); //PHI and send signal sig2 at the end

    NSc=N*(2/3); NEC=N;
    work(knt,NSc,NEc, N,a); //run on CFU at the same time

    #pragma offload_wait target(mic:0) wait(&sig1) //wait signal sig1
    #pragma offload_wait target(mic:1) wait(&sig2) //wait signal sig2
    knt=knt+1;
}
```
Share work between a host and two KNC (example)
### Offload Compilation and MKL

#### Automatic Offload Controls

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_MIC_ENABLE</td>
<td>Enables Automatic Offload</td>
<td>1</td>
</tr>
<tr>
<td>OFFLOAD_DEVICES</td>
<td>List of coprocessors to be used</td>
<td>List with comma</td>
</tr>
<tr>
<td>OFFLOAD_ENABLE_ORSL</td>
<td>Set this variable if your application used both Compiler Assisted and Automatic Offload but does not implement its own synchronisation</td>
<td>1</td>
</tr>
</tbody>
</table>
Offload Compilation and MKL

MKL: Highly Optimized Function
- BLAS Level 3, and much of Level 1 & 2
- Sparse BLAS: ?CSRMV, ?CSRMM
- Some important LAPACK routines (LU, QR, Cholesky)
- Fast Fourier transforms
- Vector Math Library
- Random number generators in the Vector Statistical Library
Offload Compilation and MKL

Automatic Offload Enabled Functions
- A selected set of MKL functions are AO enabled.
- Only functions with sufficient computation to offset data transfer overhead are subject to AO
- **Level-3 BLAS**: ?GEMM, ?TRSM, ?TRMM, ?SYMM
- **LAPACK 3 amigos**: LU, QR, Cholesky, Eigensolver (DSYEV)

Offloading happens only when matrix sizes are right. The following are dimension sizes in numbers of elements.
- ?GEMM: M, N > 2048, K > 256
- ?SYMM: M, N > 2048
- ?TRSM/?TRMM: M, N > 3072
- LAPACK AO: M, N > 8192
Offload Compilation and MKL, Fortran API

- `rc=mkl_mic_enable()` Enables Automatic Offload mode.
- `rc=mkl_mic_get_device_count()` Returns the number of Intel Xeon Phi coprocessors on the system when called on the host CPU.
- `rc = mkl_mic_set_workdivision(target_type, target_number, wd)` For computations in the Automatic Offload mode, sets the fraction of the work for the specified coprocessor or host CPU to do.
- `rc = mkl_mic_get_workdivision(target_type, target_number, wd)` For computations in the Automatic Offload mode, retrieves the fraction of the work for the specified coprocessor or host CPU to do.
- `rc=mkl_mic_set_max_memory(target_type, target_number, mem_size)` Sets the maximum amount of Intel Xeon Phi coprocessor memory reserved for the Automatic Offload computations.
- `rc=mkl_mic_free_memory(target_type, target_number)` Frees the coprocessor memory reserved for the Automatic Offload computations.
- `rc=mkl_mic_set_device_num_threads(target_type, target_number, num_threads)` Sets the maximum number of threads to use on an Intel Xeon Phi coprocessor for the Automatic Offload computations.
- `rc=mkl_mic_set_offload_report(enabled)` Turns on/off reporting of Automatic Offload profiling.
- `rc=mkl_mic_disable()` Disables Automatic Offload mode.
MKL DGEMM
MKL on two Intel Xeon Phi Coprocessor

```fortran
err = mkl_mic_enable()

call mpi_init(ierr)
call mpi_comm_rank(MPI_COMM_WORLD, my_rank, ierr)
call mpi_comm_size(MPI_COMM_WORLD, size, ierr)

if (size.gt.2) then
    call MPI_FINALIZE(ierr)
    stop 666
end if

if (my_rank==0) print *, "MKL Offload", err

if (my_rank == 0) then
    err = mkl_mic_set_workdivision(MKL_TARGET_MIC, 0, wd1)
    print *, "Section 1 ", err, ierr, my_rank
    call DGEMM('No Transpose', 'No Transpose', N, N, N, alpha1, A1(1,1), N, B1(1,1), N, beta1, C1(1,1), N)
    print *, "C1(N,N)="' , C1(N,N)
else

    err = mkl_mic_set_workdivision(MKL_TARGET_MIC, 1, wd2)
    print *, "Section 2 ", err, ierr, my_rank
    call DGEMM('No Transpose', 'No Transpose', N, N, N, alpha2, A2(1,1), N, B2(1,1), N, beta2, C2(1,1), N)
    print *, "C2(N,N)="' , C2(N,N)
end if

call mpi_finalize(ierr)
```
Offload compilation

MKL: Users can use AO for some MKL calls and use CAO for others in the same program

- Only supported by Intel compilers
- **Work division must be set explicitly** for AO
- Otherwise, all MKL AO calls are **executed on the host**

  - Set OFFLOAD_ENABLE_ORSL=1 for better resource synchronization

  - Can be done simultaneously from different threads
Optimization on KNC
Optimization Cycle

Profile

Identify Bottlenecks

Understand origin

Identify Solution

Apply Solution

Test

PROFILING

OPTIMIZING
Profile

Profile: Search which part of app. takes the most of time

- Find **hot spots**: Intel Vtune Amplifier or Gprof
- Use loop profiler or Vtune Amplifier to find the **hot loop**

- **Hot Spots** Function: large function that take a lot of runtime.

- **Hot loops** with large trip counts are responsible for large amounts of runtime.
Profile: Identify hot spot and functions

- Use **Gprof** to add instrumentation to the application (HOST only)

```
ifort -pg ..
```

- Running the application
- Generate a report

```
gprof options [executable-file [profile-data-files...]] [> outfile]
```

- **Outfile** file contains information:
  - Call count of routines
  - Self-time of functions
  - Total-time of functions
  - Call graph
Profile: Identify hot spot and functions

- Use Intel compiler options to add instrumentation to the application
  
  ```bash
  ifort -profile-functions -profile-loops-all -profile-loops-report-2...
  ```

- Running the application generates a report
- Analyze data by looking at the raw text file or use the GUI loopfileviewer.sh (need java)

- Report file contains information:
  - Call count of routines
  - Self-time of functions/loops
  - Total-time of functions/loops
  - Average, minimum, maximum iteration counter of loops
Profile: Identify hot spot and functions

- Use Intel Vtune Amplifier to profile your code

- Self-time of functions
- Total-time of functions
- Call graph
- Open MP analysis
Profile: Identify hot spot and functions

- Use Intel Trace Analyser and Collector (ITAC) to profile your MPI code.
  - Visualize and understand parallel application behavior
  - Evaluate profiling statistics and load balancing
  - Analyze performance of subroutines or code blocks
  - Identify communication hotspots
Profile: Understand the compiler work

- Loop **Vectorization efficiency** is a performance lever on MIC architecture

- Option from compiler: `-qopt-report=[n]`. This option will tell you which loops have been vectorized and why not,
  - Option will inform you how the compiler transforms your code
  - In practice, this can be useful for answering questions like why did the compiler say that?

- Vectorization Guidance: `-guide-vec=[n]`. Compiler can suggest code transformation

- Additional Guidance: `-guide`
Threads affinity and Load Balancing

**How Many Threads per Core?**

- Intel Xeon Phi Coprocessor allow 4 threads per core, there can be good reasons to use fewer:
  - Minimize cache/TLB thrashing when too many threads complete for L1 or L2
  - Lower completion of the core's single vector unit but at least 2 threads/core must be used to exploit ILP (Instruction Level Parallelism)
  - Minimize requests to main memory

- Good reasons to have all 4 threads
  - Take advantage of data locality
  - Keep per-core thread working set small enough

- The best: it's depends
  - 1 thread/core for workloads with high DRAM bandwidth requirements
  - 4 threads/core for many of the GEMM workloads
  - 3 threads/core is the sweet spot for most assembly code we wrote
  - 2 threads/core gives more than 90% of instruction issue bandwidth
Threads affinity and Load Balancing

- Controlling Threads per core using OpenMP

- Use **KMP_AFFINITY** variable (default = none)
  - When OMP_NUM_THREADS is less than the number of available threads
  - KMP_AFFINITY="compact" distributes threads as closed as possible
  - KMP_AFFINITY="scatter" distributes threads as evenly as possible across the node
  - KMP_AFFINITY="balanced"
    - Distributes threads but keeps adjacent thread numbers next to each other
    - Essentially "scatter" when # threads < # available cores
    - Differs when # of threads > # available cores
    - Provides locality between adjacent threads that can be exploited
  - Assigns threads explicitly with proclist option
MPI and Threads pinning

Why pinning? effectively reduces caches problems and IO problems. Default affinity is not necessary the best for you.

- One MPI task:
  - KMP_AFFINITY

- Two MPI tasks on MIC:
  - launch without pinning: MPI task use same CPU
  - launch with pinning: MPI task use different CPU
  - KMP_AFFINITY and KMP_PLACE_THREADS (reserve cpu for each task)
MPI and Threads pinning

- MIC prefix variable: use them on MIC.

```
$ export MIC_ENV_PREFIX=MIC
$ export OMP_NUM_THREADS=16
$ export MIC_OMP_NUM_THREADS=240
$ ./a.out
```

- Threads pinning:

```
$ export MIC_ENV_PREFIX=MIC
$ export MIC_KMP_AFINTY=compact
$ export MIC_KMP_PLACE_THREADS=30c,4t,30o
$ export MIC_OMP_NUM_THREADS=120
```

- `MIC_KMP_PLACE_THREADS` = `#cores c, #threads/cores t, offset o`
Pinning of processes MPI on the PHI

```bash
$ export MIC_ENV_PREFIX=MIC
$ export MIC_KMP_AFFINITY=compact
$ export MIC_OMP_NUM_THREADS=80
$ mpirun -n 1 -env MIC_KMP_PLACE_THREADS=20c,4t,0o ./a.out : -n 1 -env MIC_KMP_PLACE_THREADS=20c,4t,30o ./a.out
```
Threads Affinity - Example of performance gain
Optimization Cycle

- **Profile**
- **Identify Bottlenecks**
- **Understand origin**
- **Identify Solution**
- **Apply Solution**
- **Test**

**PROFILING**

**OPTIMIZING**
Optimization: Three Things for performance

▶ Exploit thread and task parallelism
  – OpenMP on loop (thread)
  – Algorithm parallel (MPI task)
  – Minimize communication and data shared

▶ Optimize memory access
  – Forget arrays of structures, think structures of arrays
  – Use “cache blocking” method to maximize data reuse

▶ Think SIMD (Simple Instruction Multiple Data):
  – No dependencies between loop iterations
  – Minimize branch condition in the inner loop
  – Continuous access to data
  – Vectorize inner loop with pragma SIMD
Optimization

Optimizing your code for CPU is the same as for Xeon PHI coprocessor

- Optimize for processor = optimize for Intel Xeon Phi and next generation
- Just adapt threading and data reuse
Vectorization

Theoretical Gflops = #cores*freq*#inst/cycle

**Scalar computation**

\[
\begin{align*}
X + Y * Z &= A \\
1\text{FMA/cycle} &\Rightarrow 2\text{op/cycle} \\
&\Rightarrow 146\text{GFLOPS}
\end{align*}
\]

**Vectorized computation**

\[
\begin{align*}
X_1 &+ Y_1 * Z_1 &= A_1 \\
X_2 &+ Y_2 * Z_2 &= A_2 \\
X_3 &+ Y_3 * Z_3 &= A_3 \\
X_4 &+ Y_4 * Z_4 &= A_4 \\
X_5 &+ Y_5 * Z_5 &= A_5 \\
X_6 &+ Y_6 * Z_6 &= A_6 \\
X_7 &+ Y_7 * Z_7 &= A_7 \\
X_8 &+ Y_8 * Z_8 &= A_8 \\
8\text{FMA/cycle} &\Rightarrow 16\text{op/cycle} \\
&\Rightarrow 1171.2\text{GFLOPS}
\end{align*}
\]
Branch condition

```c
for (int i=0; i< N; i++){
    if(test)
        a[i]=b[i]*c[i]
    else
        a[i]=1
}
```

**KO**

difficult to vectorize for compiler, generation of mask

```c
if(test)
    for (int i=0; i< N; i++)
        a[i]=b[i]*c[i]
else
    for (int i=0; i< N; i++)
        a[i]=1
```

**OK**
Data access

```
for (int i=0; i< N; i++)
    for(int j=0; j< N ; j++)
        a[j][i] = b[j][i]
```

noncontiguous access

```
Reverse indices of loops

for(int j=0; j< N ; j++)
    for (int i=0; i< N; i++)
        a[j][i] = b[j][i]
```

OK
Data layout (AoS => SoA)

for (int i=0; i< N; i++)
    for (int j=0; j< N ; j++)

for (int i=0; i< N; i++) {
    for (int j=0; j< N ; j++) {
        R[i][j] = sqrt((Ax[i]-Bx[j])*(Ax[i]-Bx[j])
                        + (Ay[i]-By[j])*(Ay[i]-By[j])
                        + (Az[i]-Bz[j])*(Az[i]-Bz[j]));
    }

E5-2680v3  24OMP ICC 15.0.2  N=2000
Make Loops Simpler

Better for data reuse, vectorization, compiler optimizations

```c
for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
        a[i] += c[i+j]+d[i+j]+e[i+j]+w[i+j];
        b[i] += x[i+j]+y[i+j]+z[i+j]+v[i+j];
    }
}
```
Increase data reuse

```java
for (int i = 0; i < N; i++) {
    a[i] = b[i] * c[i]
}
for (int i = 0; i < N; i++) {
    z[i] = b[i] + c[i]
}
```

**KO**

```java
for (int i = 0; i < N; i++) {
    a[i] = b[i] * c[i]
    z[i] = b[i] + c[i]
}
```

**OK**

Cache blocking is also to be sought!
Cache Blocking

```java
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        a[i][j] = b[i][j] * b[i-1][j] + c[i][j] * c[i-1][j]

for (int ib = 0; ib < N; ib += size_bi)
    for (int jb = 0; jb < N; jb += size_bj)
        for (int i = ib; i < min(size_bi, N); i++)
            for (int j = jb; j < min(size_bj, N); j++)
                a[i][j] = b[i][j] * b[i-1][j] + c[i][j] * c[i-1][j]
```

KO

OK
Profile and Optimize your application

▶ Profile your code to search the HotSpot (Vtune, Gprof...)

▶ Is it a parallel and vectorized code ?
  – Profile your code with Vtune,
  – see the optimization report to determine if your code is vectorized

▶ Identify problems
  – Conditional loop, bad memory access ?
  – ...

▶ Optimize it
Example of gain non-opt vs data-layout opt kernel on CPU E52680v3

![Graph showing speed-up comparison between non-opt and opt kernels on 2*E52680v3](#)
Example of gain non-opt vs data-layout opt kernel on PHI7120P

kernel non-opt vs opt on PHI 7120P 240threads

Time(s) vs #elem

- speed-up
- phi1.2
- phi3
Example of gain
2*E52680v3 and PHI7120p

PHI vs CPU opt kernel

<table>
<thead>
<tr>
<th>#elem</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0.5</td>
</tr>
<tr>
<td>8000</td>
<td>1</td>
</tr>
<tr>
<td>16000</td>
<td>2.5</td>
</tr>
</tbody>
</table>

- speed-up
- PHI7120p 240 61c,4t
- CPU 2*E52680v3 48OMP
Introduction to KNL
Knights Landing

A Paradigm Shift from KNC

- Memory Bandwidth
  400+ GB/s STREAM
- Memory Capacity
  Over 25x KNC
- Resiliency
  Systems scalable to >100 PF
- Power Efficiency
  Over 25% better than card
- I/O
  200 Gb/s/dir with int fabric
- Cost
  Less costly than discrete parts
- Flexibility
  Limitless configurations
- Density
  3+ KNL with fabric in 1U
# Knights Landing vs. Knights Corner Feature Comparison

<table>
<thead>
<tr>
<th>FEATURE</th>
<th>INTEL® XEON PHI™ COPROCESSOR 7120P</th>
<th>KNIGHTS LANDING PRODUCT FAMILY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Cores</td>
<td>Up to 61 enhanced P54C Cores</td>
<td>Up to 72 enhanced Silvermont cores</td>
</tr>
<tr>
<td>Key Core Features</td>
<td>In order</td>
<td>Out of order</td>
</tr>
<tr>
<td></td>
<td>4 threads / core (back-to-back scheduling restriction)</td>
<td>4 threads / core</td>
</tr>
<tr>
<td>Peak FLOPS</td>
<td>SP: 2.416 TFLOPs • DP: 1.208 TFLOPs</td>
<td>Up to 3x higher</td>
</tr>
<tr>
<td>Scalar Performance</td>
<td>1X</td>
<td>Up to 3x higher</td>
</tr>
<tr>
<td>Vector ISA</td>
<td>x87, (no Intel® SSE or MMX™), Intel IMIC</td>
<td>x87, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, Intel® AVX, AVX2, AVX-512 (no Intel® TSX)</td>
</tr>
<tr>
<td>Interprocessor Bus</td>
<td>Bidirectional Ring Interconnect</td>
<td>Mesh of Rings Interconnect</td>
</tr>
</tbody>
</table>
**KNL Microarchitecture**

- Intel Xeon Binary compatible (BDX)
- On-package memory
  - MCDRAM (16 GB, ~500Gb/s on STREAM)
- 6ch DDR4 @ 2400 MT/s

**Improvements:**
- 2D Mesh architecture
- Out-of-order cores
- 3x single-threads vs KNC

Tiles are based on Intel Atom cores (Silvermont)
KnL Mesh Interconnect

- Mesh of Rings
  - Every row and column is a ring
  - YX routing: Go in Y → Turn → Go in X
    - 1 cycle to go in Y, 2 cycles to go in X
    - Messages arbitrate at injection and on turn

Mesh at fixed frequency of 1.7 GHz

Distributed Directory Coherence protocol

KnL supports 3 Cluster Modes
1) All-to-all
2) Quadrant
3) Sub-NUMA Clustering

Selection done at boot time.
Cluster mode: all-to-all

- Address uniformly hashed across all distributed directories
- No affinity between tile, directory and memory

Typical Read L2 miss

1. L2 miss encountered
2. Send request to the distributed directory
3. Miss in the directory. Forward to memory
4. Memory sends the data to the requestor

1) L2 Miss 2) Directory access 3) Memory access 4) Data return
Cluster mode: Quadrant

- Chip divided into four Quadrants
- Address hashed to a directory in the same quadrant as the memory
- Affinity between the Directory and Memory
- SW Transparent

1) L2 Miss 2) Directory access 3) Memory access 4) Data return
Cluster mode: Sub-NUMA Clustering (SNC4)

- Each Quadrant (Cluster) exposed as a separate NUMA domain to OS

- Analogous to 4-socket Xeon

- Affinity between tile, directory and Memory

- Local Communication. Lowest latency of all modes

- SW Needs to NUMA optimize to get benefit

1) L2 Miss 2) Directory access 3) Memory access 4) Data return
KNL Memory Modes

- 3 possible modes (Flat, Cache, Hybrid)
- Mode selected at boot
- 16 GB Fast memory
- MCDRAM-Cache covers all DDR
MCDRAM: Cache vs Flat Mode

<table>
<thead>
<tr>
<th></th>
<th>DDR Only</th>
<th>MCDRAM as Cache</th>
<th>MCDRAM Only</th>
<th>Flat DDR + MCDRAM</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Software Effort</strong></td>
<td>No software changes required</td>
<td>Change allocations for bandwidth-critical data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Not peak performance.</td>
<td>Best performance.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Recommended
- Limited memory capacity
- Optimal HW utilization + opportunity for new algorithms
**KNL ISA**

E5-2600(SNB)  
- x87/MMX  
- SSE  
- AVX  

E5-2600v3 (HSW)  
- x87/MMX  
- SSE  
- AVX  
- AVX2  
- BMI  
- TSX  

**KNL (Xeon PHI)**  
- x87/MMX  
- SSE  
- AVX  
- AVX2  
- BMI  

**KNL Implements all legacy instructions**  
- Legacy binary runs w/o recompilation  
- KNC binary require compilation

**KNL introduces AVX-512 Extensions**  
- 512-bit FP/integer Vectors  
- 32 registers & 8 mask registers  
- Gather/scatter

**Conflict Detection**: improves vectorization  
**Prefetch**: Gather and scatter prefetch  
**Exponential and Reciprocal** instructions
Programming Model on KNL

• Programming KNL is simpler than programming KNC

• No data transfer or offloading to manage

• Only compile with `-xMIC_AVX512` and run

• All optimizations made on KNC should be benefic on KNL
  • Vectorization
  • Threading
  • Cache blocking
Lessons from Previous Architectures

Vectorization
• Avoid cache-line splits; align data structures to 64 bytes.
• Avoid gathers/scatters; replace with shuffles/permutes for known sequences.
• Avoid mixing SSE, AVX and AVX512 instructions.

Threading
• Ensure that thread affinities are set.
• Understand affinity and how it affects your application (i.e. which threads share data?).
• Understand how threads share core resources.
  – An individual thread has the highest performance when running alone in a core.
  – Running 2 or 4 threads in a core may result in higher per core performance but lower per thread performance.
Thanks

For more information please contact:

cedric.bourrasset@atos.net
Backup slides

27-10-2016
KNL Core and VPU

- Out-of-order core w/ 4 SMT threads
- VPU tightly integrated with core pipeline
- 2-wide decode/rename/retire
- 2x64B load & 1 64 B store port
- L1 and L2 prefetcher
- Fast unaligned and cache-line split support
- Fast gather/scatter support
KNL Hardware Threading

- 4 threads per core SMT

- Resources dynamically partitioned
  - Re-order buffer
  - rename buffers
  - reservation station

- Resources shared
  - Caches
  - TLB
KNL Core organization
KNL Core organization

- **Front-End Unit (FEU)**
  - Decode, allocate 2 instructions/cycle
  - 32-entry instruction queue
  - Gskew-style branch predictor
FEU nuances

- Branch predictor assumes branch destination is within the same 4GB as the source. i.e. Far branches to dynamic libraries will be predicted poorly.
  - Limit OS calls or use glibc >= 2.23 with LD_PREFER_MAP_32BIT_EXEC
  - Patch needs to be applied to OS manually ATM
KNL Core organization

- Allocation Unit
  - 72-entry ROB buffer
  - 72-entry rename buffers
  - 16 store data buffers
  - 4 gather scatter data tables
KNL Core organization

Integer Execution Unit (IEU)

2 IEUs per core
- 2 uops dispatched / cycle
- 12-entries each

Out-of-order

Most operations take 1 cycle
- Some operations take 3-5 and are supported on only one IEU (e.g. muls)
IEU nuances

Integer division is slow. If integers are known, use fast SW sequences or SHR/AND operations.

Avoid scalar code whenever possible.
Memory Execution Unit (MEU)

Dispatches 2 uops (either LD/ST)
- in-order
- but can complete in any order

2 64B load & 1 64B store port for Dcache

L2 supports 1 Line Read and ½ Line Write per cycle

L1 - L2 prefetcher
- Track up to 48 access patterns

Fast unaligned and cache-line split support

Fast gather/scatter support
MEU nuances

Performance penalty for split cache line accesses

Performance penalty when a load/store crosses the 4K boundary.
  • If cannot avoid, try adding SW prefetches to L2 several iterations ahead
Vector Processing Unit (VPU)

- 2 VPUs tightly integrated with core pipeline
- 20-entry FP RS
- executed out-of-order
- 2 512-bit FMA / cycle
- Most FP operations take 6 cycles
- 1 VPU provides legacy x87, MMX support a subset of SSE instructions
KNL Core organization

VPU nuances

Minimize usage of x87, MMX and SSE instructions and favor AVX equivalents

Code relying on intrinsic code for KNC is unlikely to generate optimal KNL code
  • Review the code or re-compiling from high-level language will probably generate faster code

Moves within the 2 VPUs require a delay to allow the values to arrive
  • Blocks implementing VPU shuffles, x87 and byte/word multiplies are distant to other blocks

Vectorize FP divisions/SQRTs

AVX-512 achieves best performance for instructions that do not use masking
KNL Core organization

- Retire

- 2 instructions / cycle