Introduction to KNL and Parallel Computing

Steve Lantz
Senior Research Associate
Cornell University Center for Advanced Computing (CAC)
steve.lantz@cornell.edu

High Performance Computing on Stampede 2, with KNL, Jan. 23, 2017
www.cac.cornell.edu
Big Plans for Intel’s New Xeon Phi Processor, KNL

<table>
<thead>
<tr>
<th>HPC System</th>
<th>Cori</th>
<th>Trinity</th>
<th>Theta*</th>
<th>Stampede 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sponsor</td>
<td>DOE</td>
<td>DOE</td>
<td>DOE</td>
<td>NSF</td>
</tr>
<tr>
<td>Location</td>
<td>NERSC</td>
<td>Los Alamos</td>
<td>Argonne</td>
<td>TACC</td>
</tr>
<tr>
<td>KNL Nodes</td>
<td>9,300</td>
<td>9,500</td>
<td>3,240</td>
<td>5,940</td>
</tr>
<tr>
<td>Other Nodes</td>
<td>2,000</td>
<td>9,500</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total Nodes</td>
<td>9,500</td>
<td>19,000</td>
<td>3,240</td>
<td>5,940</td>
</tr>
<tr>
<td>KNL DP FLOP/s</td>
<td>27.9 PF</td>
<td>30.7 PF</td>
<td>8.5 PF</td>
<td>18.1 PF</td>
</tr>
<tr>
<td>Other DP FLOP/s</td>
<td>1.9 PF</td>
<td>11.5 PF</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Peak DP FLOP/s</td>
<td>29.8 PF</td>
<td>42.2 PF</td>
<td>8.5 PF</td>
<td>18.1 PF</td>
</tr>
</tbody>
</table>

*Forerunner to Aurora: next-gen Xeon Phi, 50,000 nodes, 180 PF
## Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>One of the individual computers linked together by a network to form a parallel system. User access to a node is mediated by an operating system which runs on that node. One node may host several operating systems as virtual machines.</td>
</tr>
<tr>
<td>cluster</td>
<td>An architecture consisting of a networked set of nodes functioning as a single resource.</td>
</tr>
<tr>
<td>processor</td>
<td>The part of the computer that actually executes instructions. Commonly it refers to a single physical chip in a node. That chip may contain multiple cores or CPUs, each of which can execute operations independently.</td>
</tr>
<tr>
<td>flop/s</td>
<td>FLoating-point OPerations per Second. Used to measure a computer’s performance. It can be combined with common prefixes such as M=mega, G=giga, T=tera, and P=peta.</td>
</tr>
</tbody>
</table>
Xeon Phi: What Is It?

- An x86-derived processor featuring a large number of cores
  - Many Integrated Core (MIC) architecture
- An HPC platform geared for high floating-point throughput
  - Optimized for floating-point operations per second (flop/s)
- Intel’s answer to general purpose GPU (GPGPU) computing
  - Similar flop/s/watt to GPU-based products like NVIDIA Tesla
- Just another target for the compiler; no need for a special API
  - Compiled code includes instructions for 512-bit vector operations
- Initially, a full system on a PCIe card (separate Linux OS, RAM)...  
- **KNL**: with “Knights Landing”, *Xeon Phi can be the main CPU*
### More Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
<td>A processing unit on a computer chip capable of supporting a thread of execution. Usually “core” refers to a physical CPU in hardware. However, Intel processors can appear to have 2x or 4x as many cores via “hyperthreading” or “hardware threads”.</td>
</tr>
<tr>
<td>thread</td>
<td>A portion of a process (running program) that is executing a sequence of instructions. It shares a virtual address space with other threads in the same process.</td>
</tr>
<tr>
<td>vectorization</td>
<td>A type of parallelism in which specialized vector hardware units perform numerical operations concurrently on fixed-size arrays, rather than on single elements. See SIMD.</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data. It describes the instructions and/or hardware functional units that enable one operation to be performed on multiple data items simultaneously.</td>
</tr>
</tbody>
</table>
And... Here It Is! But... How Did We Get Here?

**Intel Xeon Phi “Knights Landing” (KNL)**
- 72 cores maximum
- Cores grouped in pairs (tiles)
- 2 vector units per core
Committee on Sustaining Growth in Computing Performance, National Research Council.
“What Is Computer Performance?”
In *The Future of Computing Performance: Game Over or Next Level?*
Moore’s Law in Another Guise

• *Moore’s Law is the observation that the number of transistors in an integrated circuit doubles approximately every two years*
  – First published by Intel co-founder Gordon Moore in 1965
  – Not really a law, but the trend has continued for decades
• **So has Moore’s Law finally come to an end? Not yet!**
  – Moore’s Law *does not* say CPU clock rates will double every two years
  – Clock rates have stalled at < 4 GHz due to power consumption
  – Only way to increase performance is through *greater on-die parallelism*
• Microprocessors have adapted to power constraints in two ways
  – From a single CPU per chip to multi-core to *many-core* processors
  – From scalar processing to *vectorized* or *SIMD* processing
  – Not just an HPC phenomenon: such chips are in your laptop too!
Evolution of Vector Registers and Instructions

- Core has 16 (SSE, AVX) or 32 (AVX-512) separate vector registers
- In 1 cycle, both ADD and MUL units can do operations with registers
### Processor Types in TACC’s Stampede, 2017

<table>
<thead>
<tr>
<th></th>
<th>Xeon E5</th>
<th>KNC</th>
<th>KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>8</td>
<td>61</td>
<td>68</td>
</tr>
<tr>
<td>SIMD width (doubles)</td>
<td>4</td>
<td>8</td>
<td>8 x 2</td>
</tr>
<tr>
<td>Multiply/add in 1 cycle</td>
<td>x 2</td>
<td>x 2</td>
<td>x 2</td>
</tr>
<tr>
<td>Clock speed (Gcycle/s)</td>
<td>2.7</td>
<td>1.01</td>
<td>1.4</td>
</tr>
<tr>
<td>DP Gflop/s/core</td>
<td>21.6</td>
<td>16.2</td>
<td>44.8</td>
</tr>
<tr>
<td>DP Gflop/s/processor</td>
<td>173</td>
<td>988</td>
<td>3046</td>
</tr>
</tbody>
</table>

- Xeon is designed for all workloads; high single-thread performance
- Xeon Phi is general purpose, too; optimized for number crunching
  - High aggregate throughput via lots of weaker threads, more SIMD
  - Possible to get 2–8x performance compared to dual E5 CPUs
Two Types of Xeon Phi (and Xeon) Parallelism

- **Threading** *(task parallelism)*
  - OpenMP, Intel Threading Building Blocks, Intel Cilk Plus, Pthreads, etc.
  - It’s all about sharing work and scheduling

- **Vectorization** *(data parallelism)*
  - “Lock step”, Instruction Level Parallelism (SIMD) using vector operands
  - Compiler generates instructions for synchronized execution
  - It’s all about finding simultaneous operations

- To utilize Xeon Phi fully, both types of parallelism must be exposed!
  - With 2–4 threads per core, can get 60x single-threaded performance
  - Vectorized loops gain 8x or 16x performance on Xeon Phi!
  - Important for CPUs as well: vectorized loops gain 4x or 8x on Xeon E5
  - *Question for later*: can vector units be fed with data fast enough?
Task (Functional) Parallelism

- Each worker performs a separate task by working on a completely different “function”, or just by executing code sections that are independent.

- Analogy: 2 brothers do yard work
  - 1 edges, 1 mows

- Analogy: 8 farmers build a barn
  - 1 saws the wood
  - 2 hammer in the nails, etc.

- Commonly programmed with:
  - Message-passing libraries like MPI
  - Threading libraries like OpenMP
Data Parallelism

• Each worker does the exact same task on unique and independent groups of data

• Analogies:
  – 2 brothers mow the lawn
  – 8 farmers paint a barn

• Vectorizes well!
  – Naturally expressed as SIMD
  – Load balancing is precise

• MPI, OpenMP also work well
  – Assign different datasets to workers
  – Workers are MPI processes or OpenMP threads
What About Those Non-Parallel Parts?

- All parallel programs contain:
  - Parallel sections (we hope!)
  - Serial sections (unfortunately)
  - In our analogy: the farmers must meet to decide who is painting where!
- Serial sections limit the parallel effectiveness

<table>
<thead>
<tr>
<th>serial part</th>
<th>parallel part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 task</td>
<td></td>
</tr>
<tr>
<td>2 tasks</td>
<td></td>
</tr>
<tr>
<td>4 tasks</td>
<td></td>
</tr>
</tbody>
</table>

- *Amdahl’s Law* quantifies this limit
Amdahl’s Law

- For large $N$, the parallel speedup doesn’t asymptote to $N$, but to a constant $1/a$, where $a$ is the serial fraction of the work.
- The graph below compares perfect speedup (green) with maximum speedup of code that is 99.9%, 99% and 90% parallelizable.

\[
T(N) = \text{total time} = \frac{p}{N} + s
\]

\[p = \text{parallel workload}\]
\[s = \text{serial time}\]

\[
S(N) = \text{speedup} = \frac{T(1)}{T(N)} = \frac{p + s}{\left(\frac{p}{N} + s\right)}
\]

If $a = s/(p + s)$, then

\[
S(N) = \frac{1}{\left[\frac{(1-a)}{N} + a\right]}
\]

\[\rightarrow 1/a \text{ for large } N\]
“Embarrassingly Parallel” – the Ideal?

- Workers are so independent that they have no need to coordinate
  - Also called “pleasingly parallel” (Why be embarrassed? You win!)
  - Special case of data parallelism
  - “Master” may assign the tasks

- Examples:
  - Monte Carlo simulations
  - Parameter sweeps
  - ATM transactions

- Programming is fairly easy
  - MPI, OpenMP, or even just a top-level script
  - Stampede provides a special framework for running this type of job without any parallel programming; see “module spider pylauncher”
Hands-on Session

Goals

1. Start an interactive session on a KNL compute node
   – Assumption: you are already logged in to the KNL login node

2. Compile and run a simple code parallelized with OpenMP
   – For fun: play with the OMP_NUM_THREADS environment variable
   – We will take some time for discussion after this step

3. Change the OpenMP directive so the code behaves unpredictably!
   – Don’t terminate your interactive session until this step is completed

To start:

```
tar xvf ~tg459572/LABS/cornellcac_labs.tar
cd knl_intro
```
The Parallel Section of omp_hello.c

```c
#pragma omp parallel private(tid)
{
    tid = omp_get_thread_num();
    printf("  => hello from thread id %3.3d\n", tid);
}
```

- The line preceding {...} is an OpenMP directive.
- This tells the compiler to insert special instructions that will cause identical copies of {...} to run in parallel on every thread.
- We specify one variable “tid” to be private to each thread.
  - By default, variables are shared by all threads and are not copied.
- Its value is set by an OpenMP function that returns the thread id.
1. Start an Interactive KNL Session

Only compute nodes have KNL processors – the login node does not. To get a 30-minute interactive session on a development node, type:

```
idev -r -A TG-TRA140011
```

You will see SLURM-related output scroll by, followed by a prompt on a compute node. Your node should be part of the reservation (-r) for this workshop. If no nodes are left (unlikely), try a different queue:

```
idev -p development -A TG-TRA140011
```

Check queue status as necessary with:

```
sinfo -o "%20P %5a %.10l %16F"
```
2. Compile and Run the Simple OpenMP Code

Compile the code with `-xMIC-AVX512`, which is the option to use for any code that you ultimately want to run on the KNL compute nodes:

```
icc -qopenmp -xMIC-AVX512 omp_hello.c -o omp_hello
```

Set the number of threads to the number of cores and run the code:

```
export OMP_NUM_THREADS=68
./omp_hello | sort
```

Repeat with the variable set to the number of hardware threads, 4 x 68:

```
export OMP_NUM_THREADS=272
./omp_hello | sort
```
Discussion Questions

• Is the omp_hello program task parallel, data parallel, or “embarrassingly parallel”?
  – Are all the workers using the same set of instructions?
  – Do the workers have to coordinate among each other?
  – Careful! How do all the workers write to a single output stream?

• Is the code affected by Amdahl’s Law?
  – Are there any serial sections?
  – Careful! Is the parallel section really being divided among the workers?

• Do you think it’s possible to beat Amdahl’s Law?
  – What if the workload grows along with $N$, the number of workers?
  – Which is better: $N$ times the work in fixed time $T$, or fixed work in $T/N$?
Challenges of Parallel Programming

• What happens if we fail to make “tid” a private variable?
  – All the threads compete to write their id into the shared location!
  – This is known as a “race condition” – the results are not predictable

• Writing a *correct* parallel application can be tricky!
  – Order of completion of tasks must not be allowed to affect results
  – Workers need private memory, occasional synchronization

• Writing a *correct and effective* parallel application can be difficult!
  – Synchronization and private memory add to the overhead costs
  – Workers must wait at synchronization points if the load is unbalanced
  – Serial sections limit the parallel speedup due to Amdahl’s Law
  – Such sources of parallel overhead and inefficiency must be minimized
3. Change the Code to Make It Unpredictable

- Use an editor (nano, vi, emacs) to remove the private(tid) clause

```c
#pragma omp parallel private(tid)
{
    tid = omp_get_thread_num();
    printf(" => hello from thread id  %3.3d\n", tid);
}
printf(" x> goodbye from thread id  %3.3d\n", tid);
```

- Recompile, export OMP_NUM_THREADS=68, run multiple times
- See if a race condition results on the last line!
- When you are done, terminate your session by typing Ctrl-d
Hardware Designers Face the Same Issues

- Think of modern processors as a collection of parallel workers
  - Multiple cores all operate in parallel; so do their vector units
- It is convenient to let workers share data, but there are problems too
  - Finding the desired items among many gigabytes of data is slow
  - What if two workers need the same item at the same time?
- Solution: cache memory
  - Workers keep private copies of the main memory they need to use
  - Access to these smaller, local caches is much faster than RAM
- Cache coherence prevents race conditions
  - Two workers cannot both alter local copies of the same main memory
  - Built-in hardware mechanisms keep all the caches in agreement
- It works best to have multiple levels of cache: a memory hierarchy
Memory Hierarchy in Stampede’s KNLs

- **96 GB DRAM (max is 384)**
  - 6 channels of DDR4
  - Bandwidth up to 90 GB/s
- **16 GB high-speed MCDRAM**
  - 8 embedded DRAM controllers
  - Bandwidth up to 475 GB/s
- **34 MB shared L2 cache**
  - 1 MB per tile, 34 tiles (max is 36)
  - 2D mesh interconnection
- **32 KB L1 data cache per core**
  - Local access only
- Data travel in 512-bit cache lines
The New Level: On-Package Memory

- KNL includes 16 GB of high-speed multi-channel dynamic RAM (MCDRAM) on the same package with the processor.
- Up to 384 GB of standard DRAM is accessible through 3,647 pins at the bottom of the package (in the new LGA 3647 socket).

How Do You Use MCDRAM? Memory Modes

- **Cache**
  - MCDRAM acts as L3 cache
  - Direct-mapped associativity
  - Transparent to the user

- **Flat**
  - MCDRAM, DDR4, are all just RAM; different NUMA nodes
  - Use `numactl` or `memkind` library to manage allocations

- **Hybrid**
  - Choice of 25% / 50% / 75% of MCDRAM set up as cache
  - Not supported on Stampede
Where Do You Look for an L2 Miss? Cluster Modes

• **All-to-all**: request may have to traverse the entire mesh to reach the tag directory, then read the required cache line from memory

• **Quadrant**: data are found in the same quadrant as the tag directory

• **Sub-NUMA-4**: like having 4 separate sockets with attached memory
This Is How the Batch Queues Got Their Names!

- Stampede’s batch system is SLURM
  - Start interactive job with `idev`, OR...
  - Define batch job with a shell script
  - Submit script to a queue with `sbatch`
- Jobs are submitted to specific queues
  - Option `-p` stands for “partition”
  - Partitions are named for modes: `Memory-Cluster`
  - Development and normal partitions = `Cache-Quadrant`
- View job and queue status like this:
  
  ```
  squeue -u <my_username>
  sinfo | cut -c1-44
  ```

### Queues (Partitions) #

<table>
<thead>
<tr>
<th>Queue</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>development*</td>
<td>16</td>
</tr>
<tr>
<td>normal</td>
<td>376</td>
</tr>
<tr>
<td>Flat-Quadrant</td>
<td>96</td>
</tr>
<tr>
<td>Flat-SNC-4</td>
<td>8</td>
</tr>
<tr>
<td>Flat-All2All</td>
<td>8</td>
</tr>
<tr>
<td><strong>- Total -</strong></td>
<td><strong>504</strong></td>
</tr>
<tr>
<td>systest (restricted)</td>
<td><strong>508</strong></td>
</tr>
</tbody>
</table>
Conclusions: HPC in the Many-Core Era

- HPC has moved beyond giant clusters that rely on coarse-grained parallelism and MPI (Message Passing Interface) communication
  - *Coarse-grained*: big tasks are parceled out to a cluster
  - *MPI*: tasks pass messages to each other over a local network
- HPC now also involves many-core engines that rely on fine-grained parallelism and SIMD within shared memory
  - *Fine-grained*: threads run numerous subtasks on low-power cores
  - *SIMD*: subtasks act upon multiple sets of operands simultaneously
- Many-core is quickly becoming the norm in laptops, other devices
- *Programmers who want their code to run fast must consider how each big task breaks down into smaller parallel chunks*
  - Multithreading must be enabled explicitly through OpenMP or an API
  - Compilers can vectorize loops automatically, if data are arranged well
References

- **Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor**, slides by Avinash Sodani, KNL Chief Architect, Intel Corporation
- **Parallel Programming Concepts and High-Performance Computing**, a module in the Cornell Virtual Workshop
- **Glossary** of HPC terms from the Cornell Virtual Workshop
- **Applications of Parallel Computers**, a set of lectures from a course taught by Jim Demmel at U.C. Berkeley in Spring 2012. This online rendition is sponsored by XSEDE and is only available through the [XSEDE User Portal](http://www.xseed.org).
- **Designing and Building Parallel Programs**, a 1995 book by Ian Foster. It serves as a fine introduction to parallel programming. Some of the languages covered in later chapters are outmoded, but the concepts have not changed much.