Thanks for Coming!

- In this tutorial, we will teach you what you need to know to start running your code on the Intel Xeon Phi Co-Processor.
- For the hands-on portion, we will user the Stampede system at TACC in Austin, Texas, USA, which has 6,880 Xeon Phi cards.
Team Introductions

- Lars Koesterke
- Kent Milfeld
- Dan Stanzione
- Jerome Vienne
TACC introduction

• The Texas Advanced Computing Center is a leading US provider of cyberinfrastructure.
  – Stampede, the Xeon/Xeon Phi system you will hear about today, is our 10PF flagship system
  – Come to the BOF tomorrow to hear about Wrangler, our next Big Data System

• TACC has about 15,000 users around the world (primarily US), for whom we provide:
  – ~1 billion CPU hours per year.
  – Manage 5 billion files and ~40PB of data
  – 100
Stampede - High Level Overview

- **Base Cluster (Dell/Intel/Mellanox):**
  - Intel Sandy Bridge processors
  - Dell dual-socket nodes w/32GB RAM (2GB/core)
  - 6,400 nodes
  - 56 Gb/s Mellanox FDR InfiniBand interconnect
  - More than 100,000 cores, 2.2 PF peak performance

- **Co-Processors:**
  - Intel Xeon Phi “MIC” Many Integrated Core processors
  - Special release of “Knight’s Corner” (61 cores)
  - All nodes have at least 1 card
    - 12 racks (480 nodes) have
  - 7+ PF peak performance

- **Max Total Concurrency:**
  - exceeds 500,000 cores
  - 1.8M threads

- **Entered production operations on January 7, 2013**
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)
Key aspects of acceleration

• We have lots of transistors… Moore’s law is holding; this isn’t necessarily the problem.
• We don’t really need lower power per transistor, we need lower power per *operation*.
• How to do this?
  – nVidia GPU
  – FPGA
  – AMD Fusion
  – ARM Cores
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
Intel Xeon Phi Chip

- 22 nm process
- Based on what Intel learned from
  - Larrabee
  - SCC
  - TeraFlops Research Chip
MIC Architecture

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

George Chrysos, Intel, Hot Chips 24 (2012):
George Chrysos, Intel, Hot Chips 24 (2012):
Speeds and Feeds (Intel Xeon Phi SE10P)

• 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.
(Current) Xeon Phi Programming Models

- **Traditional Cluster**
  - Pure MPI and MPI+X
    - X may be OpenMP, TBB, Cilk+, OpenCL, ...

- **Native Phi**
  - Use one Phi and run OpenMP or MPI programs directly

- **MPI tasks on Host and Phi**
  - Treat the Phi (mostly) like another host
    - Pure MPI and MPI+X (limited memory: using ‘X’ is almost mandatory)

- **MPI on Host, Offload to Xeon Phi**
  - Targeted offload through OpenMP extensions
  - Automatically offload some library routines with MKL
Traditional Cluster

- Stampede is 2+ PF of FDR-connected Xeon E5
  - High bandwidth: 56 Gb/s (sustaining >52 Gb/s)
  - Low-latency
    - ~1 µs on leaf switch
    - ~2.5 µs across the system
- Highly scalable for existing MPI codes
- IB multicast and collective offloads for improved collective performance
Native Execution

• Build for Phi with –mmic
• Execute on host (runtime will automatically detect an executable built for Phi)
• … or ssh to mic0 and run on the Phi
• Can safely use all 61 cores
  – But: we recommend use of 60 cores, i.e. 60, 120, 180, or 240 threads
  – Offload programs should certainly stay away from the 61\textsuperscript{st} core since the offload daemon runs here
Symmetric MPI

• Host and Phi can operate symmetrically as MPI targets
  – High code reuse
  – MPI and hybrid MPI+X (X = OpenMP, Cilk+, TBB, pthreads)
• Careful to balance workload between big cores and little cores
• Careful to create locality between local host, local Phi, remote hosts, and remote Phis
• Take advantage of topology-aware MPI interface under development in MVAPICH2
  – NSF STCI project with OSU, TACC, and SDSC
Symmetric MPI

• Typical 1-2 GB per task on the host
• Targeting 1-10 MPI tasks per Phi on Stampede
  – With 6+ threads per MPI task
MPI with Offload to Phi

• Existing codes using accelerators have already identified regions where offload works well
• Porting these to OpenMP offload should be straightforward
• Automatic offload where MKL kernel routines can be used
  – xGEMM, etc.
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)
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
PETSc/MUMPS with AO

- Hydrostatic ice sheet modeling
- MUMPS solver (called through PETSC)
- BLAS calls automatically offloaded behind the scenes
Phi evolution

• A few details of the next generation of Xeon Phi are announced.
  – “Knight’s Landing”
  – 14nm process
  – Available as a stand alone processor
    • Think about that as you consider programming model
  – On-package high bandwidth memory
  – AVX-512 instruction set (now publicly available).
Summary

• MIC experiences
  – Early scaling looks good; application porting is fairly straight forward since it can run native C/C++, and Fortran code
  – Some optimization work is still required to get at all the available raw performance for a wide variety of applications; but working well for some apps
  – **vectorization** on these large many-core devices is key
  – **affinitization** can have a strong impact (positive/negative) on performance
  – **algorithmic threading performance** is also key; if the kernel of interest does not have high scaling efficiency on a standard x86_64 processor (8-16 cores), it will not scale on many-core
  – **MIC optimization efforts also yield fruit on normal Xeon** (in fact, you may want to optimize there first.

• Questions?

• Then let’s dive in deeper
Acknowledgements

• Thanks/kudos to:
  – National Science Foundation
    • NSF Grant #OCI-1134872 Stampede Award, “Enabling, Enhancing, and Extending Petascale Computing for Science and Engineering”
    • NSF Grant #OCI-0926574 - “Topology-Aware MPI Collectives and Scheduling”
Lab 1

Stampede Orientation
Lab I

• What you will learn
  – The lab introduces you to Stampede and to the Xeon Phi processors built into Stampede

• What you will do:
  – Compile for Xeon (host) and Xeon Phi (MIC)
  – Submit a job
  – Inspect the queue
  – Submit an interactive job
  – Execute on the host and on the Phi
Part 0 – Grab the Lab Files

• Login to Stampede
  $ ssh <username>@stampede.tacc.utexas.edu

• Change to your $WORK directory:
  $ cdw
  $ pwd
  $ module list

• Untar the file lab1.tar file (in ~train00) into your directory:
  $ tar xvf ~train00/lab1.tar

• Move into the newly created lab1 directory:
  $ cd lab1 # first char is lower case "L"; last is a one
  $ pwd
  $ ls
Part 1 – Run an MPI Batch Job (sbatch)

- Compile the mpiipi program:
  
  ```bash
  $ mpicc mpiipi.c -o mpiipi
  ```

- Open the batch script in an editor to see if you need to change it:

  ```bash
  $ nano lab1batch  # or vi, or emacs, or just cat lab1batch
  << you shouldn't need any changes >>
  ```

- Launch the batch job

  ```bash
  $ sbatch lab1batch
  ```

- Monitor the job’s status (when done, command will return nothing):

  ```bash
  $ squeue -u <username>
  $ showq | more    # hit space bar to advance
  $ squeue | more    # hit space bar to advance
  ```

- When job completes, take a look at results:

  ```bash
  $ ls    # Note presence/names of output files
  $ more mpiipi.xxxxx.out  # "xxxxx" is your job's jobid
  $ more mpiipi.xxxxx.err  # "xxxxx" is your job's jobid
  ```
Part 2 – An Interactive Session (idev)

• Launch a one-node interactive session using a special reservation
  
  $ idev

• When session begins, compile hello.F90* from compute node:
  
  $ ifort -openmp hello.F90 -o hello

• Run the code:
  
  $ ./hello     # you're on a compute node, not a login node

• Set OpenMP threads and try again
  
  $ export OMP_NUM_THREADS=4
  
  $ ./hello     *Note: the capital "F" in the suffix instructs the compiler to interpret the macros in the source code. If the suffix were "f90" the compilation would require a "-cpp" flag.
Part 3 – Run MIC App from the Host

• While on the compute node, recompile to produce "native MIC" code (compilers are not visible from the MIC):
  
  \[
  \text{
  $\text{ifort -mmic -openmp hello.F90 -o helloMIC}$
  \]

• Launch the MIC code from the host:
  
  \[
  \text{
  $\text{./helloMIC}$
  \]

\text{Note: the program reports 244 “processors” because each MIC core has four hardware threads. It may not be efficient to run this many threads.}

• From the host, modify the MIC thread count and try again:
  
  \[
  \text{
  $\text{export MIC_OMP_NUM_THREADS=60}$
  \]
  \[
  \text{
  $\text{export MIC_ENV_PREFIX=MIC}$
  \]
  \[
  \text{
  $\text{./helloMIC}$
  \]
Part 4 – Visit the MIC

• First note the full path to your working directory:
  $ echo $WORK  # you'll need this info when you get to the MIC

• Go the MIC using ssh:
  $ ssh mic0    # the "zero" identifies the MIC card

• Move into the lab1 directory with explicit cd (alias and env variable not avail):
  $ cd /work/01875/djames    # replace with your own path
  $ cd lab1

• Run your MIC code:
  $ ./helloMIC

• Change the MIC's thread count and run code again (don't use "MIC" prefix):
  $ export OMP_NUM_THREADS=25
  $ ./helloMIC

• Return to host, then end idev session as desired:
  $ exit    # to return to host
  $ exit    # to end idev session