The Era of Heterogeneous Computing

EU-US Summer School on High Performance Computing
New York, NY, USA

June 28, 2013

Lars Koesterke: Research Staff @ TACC
## Nomenclature

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>nVIDIA GPU</td>
<td>Fermi, Tesla</td>
</tr>
<tr>
<td>Intel MIC</td>
<td>Xeon Phi</td>
</tr>
</tbody>
</table>

MIC = Many Integrated Cores
Heterogeneous Computing
Why MICs? Why GPUs?

• Success of GPUs has proven that there is room for two different core sizes in the HPC space

• Big core / Little core concept
  – Big cores (host): Serial/irregular workloads
  – Little cores (MIC/GPU): Parallel/regular workloads

• MIC provides different features than GPUs
  – MIC’s little cores are very similar to Xeon’s big cores
  – Familiar programming models
  – Native Compilation — Symmetric MPI execution

• Little cores promise more Flops per Watt and per Dollar
Competition everywhere

• Competition between MICs and GPUs
  – Performance and Programmability
  – Intel v. NVIDIA (+ ARM, AMD, IBM?, etc.)
  – Titan: 27 PF, K20X
  – Blue Waters: 12 PF, K20X
  – Stampede: 10 PF, Xeon Phi
  – Tianhe-2: 55 PF, Xeon Phi
  – K-computer: 10PF, not accelerated

• Competition between countries/regions
  – USA — Japan — Europe — China

• Competition in the hardware and software area
  – China will continue to develop their own chips and software
MIC Architecture

• Intel’s MIC is based on x86 technology
  – x86 cores w/ caches and cache coherency
  – SIMD instruction set: Vectorization

• **Programming for MIC is similar** to programming for CPUs
  – Familiar languages: C/C++ and Fortran
  – Familiar parallel programming models: OpenMP & MPI
  – **Symmetric execution**: 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
  – Make use of existing knowledge!
MIC Value Proposition

• Competitive Performance: Peak and Sustained

• Familiar Programming Model
  – HPC: C/C++, Fortran, and Intel’s TBB
  – Parallel Programming: OpenMP, MPI, Pthreads, Cilk Plus, OpenCL
  – Intel’s MKL library (later: third party libraries)
  – Serial and Scripting, etc. (anything a CPU core can do)

• “Easy” transition for OpenMP code
  – Pragmas/directives added to “offload” OMP parallel regions onto MIC

• Support for MPI
  ✓ MPI task on host, communication through offloading
  ✓ One or more MPI tasks per MIC, communication through MPI
Coprocessor vs. Accelerator

• **Differences**
  – **Architecture:** x86 vs. streaming processors
    coherent caches vs. shared memory and caches
  – HPC Programming model:
    extension to C++/C/Fortran vs. CUDA/OpenCL
    OpenCL support
  – Threading/MPI:
    OpenMP and Multithreading vs. threads in hardware
    MPI on host and/or MIC vs. MPI on host only
  – Programming details:
    offloaded regions vs. kernels
  – Support for any code: serial, scripting, etc.
    Yes  No

• **Native mode:** Any code may be “offloaded” as a whole to the coprocessor
Xeon Phi v. K-20X

- Peak performance: $\geq 1.0 \geq 1.3$ TFLOPS
- Cores/SMs: $\approx 60$ 15
- SIMD lanes/CUDA cores: 4/8 192 (per core/SM)
- FMA: yes yes
- Pipelined: yes no
  - MIC: 1 op per cycle
  - GPU: 1 op per 4 cycles
- K20X has more L2 cache per core (I think)
- MIC has more total L2 cache
- MIC relies on caches & prefetching to hide latency (opaque)
- GPU uses thousands of threads (transparent)
(Some) Lessons Learned
GPUs and MICs in the Future
Little cores are everywhere now

- Little cores are very successful in GPUs
- Intel’s little cores (aka MIC) will be a success, too
- All major companies have little cores on their roadmaps
- Little cores seem to be part of the “Road to Exascale”
- Little cores will move closer to the host CPU
  - AMD Fusion
  - Project Denver
  - Intel has similar roadmaps
  - DSP, FPGA, etc.

Lesson 1b: Heterogeneous architectures are here to stay
MICs and GPUs: All the same?

Some parts are actually the same

- Separate device
  - PCIe connection, separate memory spaces
  - Limited memory on card
  - Data transfer, double buffering (if applicable)
  - Challenges to use the host and “device” at the same time (if applicable)

Many parts are not the same
MICs and GPUs: All the same?

MICs are closer to a CPU than to an Accelerator

MICs and CPUs are programmed in the same way

• Based on the only 4 proven standards in HPC
  – C/C++ & Fortran
  – MPI & OpenMP (or threads)

• No new languages, no CUDA
  – Although OpenCL will work
  – OpenMP will become very important

• One code base for MIC and CPU (w/ minor tweaks)
  – Optimizations for MIC will also improve performance on SNB

Lesson 2: MICs are not GPUs
GPUs and MICs are the same! Both won’t do any wonders

- Neither MICs nor GPUs can perform miracles
- Reasonable expectation: 2x, 3x, 4x, … (see discussion later)

Lesson 4: Do not expect miracles
GPUs and MICs cannot pull away easily
The Xeon CPUs are just too good

- Optimizations crucial for GPU and MIC also increase performance on Sandy-Bridge
  - Highly data-parallel implementation
  - Smart memory access
  - Data alignment, huge pages, streaming stores, etc.

Optimizing for MIC will enable users to make better use of their allocation on Sandy-Bridge
(or any other cluster architecture for that matter)

This is also true for GPUs, if users are willing to port back (see Erik’s talk)

Lesson 6: Sandy/Ivy-Bridge is just so good
Adapting Scientific Code to MICs

• Today: Most scientific code for clusters
  – Languages: C/C++ and/or Fortran,
  – Communication: MPI
  – may be thread-based (Hybrid code: MPI & OpenMP),
  – may use external libraries (MKL, FFTW, etc.).

• With MIC on Stampede:
  – Languages: C/C++ and/or Fortran,
  – Communication: MPI
  – may run an MPI task on the MIC
    or may offload sections of the code to the MIC,
  – will be thread-based (Hybrid code: MPI & OpenMP),
  – may use external libraries (MKL),
    that automatically use MIC

Lesson 7: Threads are key on MICs and GPUs
Programming MIC with Threads

Key aspects of the MIC co-processor

- A lot of local memory, but even more cores
- 100+ threads or tasks on a MIC

One MPI task per core? — Probably not

- Severe limitation of the available memory per task
- Some GB of Memory & 100 tasks
  → some ten’s of MB per MPI task

➤ Threads (OpenMP, etc.) are a must on MIC
MPI Task Placement and Communication

Symmetric setup: Easier
  • MPI tasks on host and co-processor
    ➢ Equal (symmetric) members of the MPI communicator
  • Same code on host processor and MIC processor
  • Communication between any MPI tasks through regular MPI calls

Offloaded setup: More involved
  • MPI tasks on host only
  • Offload directives added to OpenMP directives
  • Communication between host and MIC through offload semantics

Lesson 8: “Offloading” will play a key role
Symmetric Execution comes with severe Challenges

- The whole code does only scale well, if
  - the serial sections are small
  - all parallel sections scale well

- Speed-up for a hypothetical MIC with 50 cores

<table>
<thead>
<tr>
<th>Scaling of parallel sections</th>
<th>Serial: 0.1%</th>
<th>Serial 1%</th>
<th>Serial 10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 Virtually no barriers, no resource contention</td>
<td>50</td>
<td>33</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>20</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

- Reasonable speed-up requires
  - serial sections ≤ 1%
  - scaling of all parallel sections ≥ 25x

Lesson 8: “Offloading” will play a key role
Taxonomy of Code Optimization

• **Level 4**: Performance is achieved by any and all means
  – Incl. assembly and architecture specific & replicated code

• **Level 3**: High-level code only (C/C++ & Fortran)
  – Incl. architecture specific/replicated code. Example: loop unrolling

• **Level 2**: High-level code.
  – Optimizations that are beneficial on all architectures are applied
  – 6 simple techniques are applied, no loop unrolling, etc.

• **Level 1**: High-level code. No/little regard for optimization

• **Level 0**: Non-HPC languages (who would do such a thing?)

  • In many cases **Level 2** is sufficient
    – Good performance on SNB and MIC
One or two code-bases for SNB & Phi?

I may be getting this one wrong!

• You may stick to “Level 2” optimizations
  ➢ “Symmetric” MPI
    – Reasonable chance that one code-base is enough
    – Different compiler options are used for the 2 compiles
  ➢ “Offloaded” MPI
    – Two kernels: one on SNB — one on MIC
    – Kernels may be the same, except for minor tweak to the thread setup and vector width with OpenMP
What I like about …

• GPUs
  – CUDA: Fine control over all moving parts
  – Can write “assembly” code in C or Fortran
  – Can hide latency by increasing number of threads
  – User-managed shared (local) memory
  – No implied barriers

• MICs
  – Can measure initial performance right away (native execution)
  – Can use C/C++ or Fortran will all features
  – OpenMP threads; proven language; synchronization on MIC
  – Caches managed by compiler and run-time
  – Can use the Intel compiler
No Revolution, but fast Evolution

• I do not buy into the “Revolution” hype (yet!)
  – The world will not be turned upside-down
  – But heterogeneous computing is here and will not go away soon

• Nothing is ever easy in parallel programming
  – Don’t know if this helps: “You can easily compile very slow code”

• Compiling code is different from getting good performance

• Things take longer than expected

• It will take time to learn and to get acquainted

• Find the right balance for you between
  – Waiting for the ultimate and mature solution
  – Jumping right in — now may be the time to jump, though!
I seem to have a different view on …

• From Erik Lindahl’s and John Urbanic’s talks
  – Speed of GPUs and MICs compared to the speed of CPUs
  – How will programming look like in the future?
  – What is a core on a GPU? Two-tiered parallelism
  – The number of cores of a GPU and a MIC
  – Speed of future generations of CPUs and MICs
  – Speed of past and future CPU cores
    (are they really getting slower?)
  – How GPUs and MICS (and CPUs!) save power
  – Limitations due to Moore’s law
  – Is graphics the future of GPU and MICs?
Speed of CPUs, GPUs, Xeon Phi

- High-end hardware in Stampede
  - Xeon E5 (Sandy-Bridge), Kepler K20, Xeon Phi
- Leveled playing field at 300W
  - K20 and Phi: 300 W cards
  - Host: dual socket (120W each), add power for the memory, etc.
- Peak performance: Peak Flops
  - Phi: 3x of host; K20X: 4x of host
- Peak performance: Peak memory bandwidth
  - Phi and Kepler alike: 2x of host (GDDR5)
- For low-end hardware the situation looks a bit better
  - Not available with Xeon Phi
  - Not really relevant for HPC
Speed of CPUs, GPUs, Xeon Phi

What speed-up should I expect

• Flop limited code: 3-4x
• Bandwidth limited code: 2x (most codes “live” here!)
• Exceptions
  – Code for accelerator more (or less) optimized
  – Percentage of peak lower (or higher) on accelerator

Anything above 4x demands an explanation

• Erik had a graph that showed
  – 1 host CPU + 1 GPU = 3 times faster than one host CPU alone

This is great but accelerators are not “insanely” fast
What about Moore’s law?

Warning: EXTREME Speculation!!!

• Yes, the shrinking of features may end somewhere, but
  – The devices can become larger
  – Xeon Phis and Kepler have already increased die sizes
  – Devices may grow in X and Y, or gain height

I fully understand, this comes with huge problems, but just because the features can not shrink in size does not mean that the number of transistors on a chip is limited with certainty.
How fast are CPUs?
Are they getting slower?

The *increase(!)* in speed has slowed down somewhat, but

- The clock rate is again slowly rising, after a drop a few years ago
  - Shrinking of transistors helps; manufactures have learned how to conserve energy
- CPUs provide more throughput through vector registers
  - Pre Sandy-Bridge: SSE (128 bits)
  - Sandy/Ivy-Bridge: AVX (256 bits)
- More powerful functional units
  - Sandy Bridge: 1 add & 1 multiply
  - Haswell: 2 FMAs (fused multiply-add)
How do all devices save power?

Two-tiered parallelism

- **Lower level**: *in lock-step* (SIMD, SIMT)
  - CPU and MIC: Vector registers (4/8 doubles wide)
  - GPU: CUDA cores/warps (Kepler: 192 wide)
- **Higher level**: *independent* (Cores act independently)
  - CPU and MIC: cores (CPU: 8, Phi: 60)
  - GPU: Streaming Multiprocessors (Kepler: 15)
- Decoding instructions consumes a lot of power
- Lock-step execution reduces number of decodings
- FMA helps here, too
- Save power: More lock-step, less independent cores
How fast will future generations be? Will the gap between CPU and accelerator widen?

The gap between will widen a bit, but:
Do not expect too much. CPUs will get faster, too.

- Haswell will be 2x faster than Ivy-Bridge
  - Ivy-Bridge: 1 add & 1 multiple
  - Haswell: 2 FMAs (Fused Multiply add)
- If the next generation of MICs and GPU provide 2x more Flops, then the current gap (3x-4x) does not increase
- I dream of an increase of 4x for MICs and GPUs
- 16 Gflops in Maxwell (up from 1.3 Gflops in Kepler) will I believe when I see it
Number of cores
Will we have to deal with “Billions” of cores in the near future?

- An SM on a GPU is comparable to a CPU/MIC core
  - Cores and SMs act independently
- CUDA cores are comparable to Vector lanes
  - Both act in lock-step
- Kepler has only 15 SMs (much lower than previous gen.) but more CUDA cores per SM
  - More lock-step, less cores → more power savings
- If you count SM’s as cores, then you never get to a “billion”
What will programming look like in the future?

These assumptions may offer some guidance

• Exascale computing by 2020
  – May be difficult in the 20 MW power envelope
  – However, the Chinese government may allow for larger envelope
  – HPC arms race may get us to Exascale with single-purpose (Top500) machines

• Accelerators are said to be key
  – We already have a feeling how these are being programmed
  – No “billion” cores for Exascale

• Radically new technologies are needed for memory and network
  – My imagination is too underdeveloped
What will programming look like?

These assumptions may offer some guidance

• We know how to code for the K computer (10 PF)
  – 80,000 nodes and we still use MPI and OpenMP
• We know how to code for accelerators
  – Threads (OpenMP or CUDA)

Imagine 160,000 nodes in 2018
• 2x nodes, each 5x more powerful: 100 PF
• Accelerators: 4x more powerful than host: 400 PF

Anything can happen but it could just be

MPI + Threads?
Future trends

• CPUs will get faster
  – Increased of clock-speed (low)
  – Increased numbers of cores (moderate)
  – More lock-step parallelism (high)
    • Sandy-Bridge: 2x from wider vectors (SSE → AVX)
    • Haswell (next year): 2x from FMA

• Accelerators will get faster
  – Increased clock-speed (low)
  – Number of cores: up, down, constant
  – Expect more lockstep parallelism

• Gap will widen, but not that quickly
Future trends

• CPUs will get faster
  – Increased of clock-speed (low)
  – Increased numbers of cores (moderate)
  – More lock-step parallelism (high)
    • Sandy-Bridge: 2x from wider vectors (SSE → AVX)
    • Haswell (next year): 2x from FMA

• Accelerators will get faster
  – Increased clock-speed (low)
  – Number of cores: up, down, constant
  – Expect more lockstep parallelism

• Gap will widen, but not that quickly
Summary

• Accelerators are fast enough to be relevant
  – Flops: 3x-4x
  – Bandwidth: 2x
• Accelerators are not fast enough to solve all your problems
• You will have to use CPUs and accelerators
  – CPUs contribute significant compute power
  – CPUs are needed for irregular/serial code
Thank You!

Questions?
How Fast is Fast?
(A word of caution)

• Floating-point bound
  – Bandwidth sufficient to support floating-point operations w/o stalls
  – Latency hidden by caches & prefetching
  – Data re-use in fast caches (not always applicable)

• Memory bound
  – Frequent stalls of floating-point
  – Bandwidth insufficient and/or latency not hidden
  – Memory access non-optimal: high-stride or random
  – Data re-use pattern exceeds cache size(s)

• Most codes and code kernels are “Memory Bound”
  – Measure both numbers: Flop rate and Bandwidth
  – Account for “wasted” bandwidth (non stride-1, random access)
How fast are CPUs and accelerators?

- Let's level the playing field first by comparing against power consumption (power is the main motivator to begin with).

  - High-end GPU K20X: 300 W
  - High-end MIC Xeon Phi: 300 W
  - High-end node: 300 W
    - Dual-socket, 8 cores
    - Each socket: 135W
    - With memory, hard drive, etc: total power 300W
Taxonomy of Code Optimization (I)

6 simple techniques for architecture independent optimization

1. Use structures of arrays (SOA) and not arrays of structures (AOS)
   – Mainly C/C++ programmers
2. Allocate multi-D arrays with one malloc to ensure contiguous memory usage
   – C/C++ programmers only
   – If it is not contiguous, it is not an array!
3. Know that stride-1 access is better than high-stride or random access. At least try to design the data for low-stride access
Taxonomy of Code Optimization (II)
6 simple techniques for architecture independent optimization

1. Know that there is vectorization and pipelining
   – Avoid loops in which operation depend on each other
   – Helps also when code is later parallelized with OpenMP

2. Do not rely on the compiler for certain optimizations
   – Compiler may not detect, may not be allowed to
     • Hoist operations
     • Eliminate redundant operations
     • Algebraic code transformations

3. Advanced techniques, like loop unrolling, cache blocking, assembly coding requires a deeper understanding of the hardware and coding techniques
   – These techniques lead to:
     • Architecture-dependent code versions
     • Code bloat