Future Programming Models

Sriram Swaminarayan
September 8, 2011
Group Leader, Applied Computer Science
Los Alamos National Laboratory
Contributing Subject Matter Experts

- Benjamin Bergen
- Marcus Daniels
- Nathan DeBardeleben
- Timothy Kelley
- Michael Lang
- Patrick McCormick
- Jamaludin Mohd-Yusof
- Craig Rasmussen
- John Thorp
Outline Of This Talk

- Anticipated changes in computer architecture
  - Hardware trends
  - Hardware attributes that will change
  - Implications of these changes for programming these architectures

- Implications for Programming Models
  - Differences between current and evolving programming models
  - Changes needed to fully utilize hardware
  - Selected models and languages
  - Resilience, or lack thereof

- Conclusions
Computer Architectures Have Not Changed For The Past Two Decades (From a Programmer’s Perspective)

- **Speed has come from added complexity:**
  - Higher clock speeds
  - More hardware caches
  - More pipelines

- **Supercomputers have reaped the benefits of ‘bigger, faster, more’:**
  - Bigger memory (both total and cache)
  - Faster CPUs
  - More nodes (for clusters)

- **From a programmer point of view**
  - Topology of computers / clusters did not change
  - All the added parallelism has been *external* to the node
  - Typically recompiling for new platform was enough to reap the benefits of the ‘new’ architectures
Computational Capacity Has Been Increased By Increasing Complexity: This Trend Is Unsustainable

Why Do architectures have to change?

- Memory bandwidth and memory per core have not kept pace with clock speed increases
- Transferring data across the chip is expensive (50% of power budget goes towards the memory in a supercomputer)
- Added clock speed requires higher power
- Added complexity within the CPU comes at cost of higher power
- This has been true for some time

Power, and more generally, Energy To Solution Will Drive Future Supercomputer Designs
Future Increases In Computational Power Will Come From Within A Node

- **Increased parallelism within a node**
  - Wider vector widths
  - More threads per core
  - More (simpler) cores per CPU

- **Heterogeneous / hybrid system designs**
  - Different types of cores within a single CPU
  - Accelerators (e.g. GPUs) added to a system
  - Accelerators *are* the system
  - Increased levels of memory hierarchy

- **All of the above have negative impacts on fault rates**

This effectively kills the *recompile and run* strategy for code survival: You *will* have to rewrite your codes.
Who Is Driving This Market?

- **Not Supercomputing!**
  - 2008 video game market: $39.0 Billion
  - 2009 video game market: $60.0 Billion
  - 2009 industry estimate for netbooks: $9.0 Billion
  - 2010 embedded computing market: $6.0 Billion
  - NNSA budget for supercomputers: $0.2 Billion

- Leads to some interesting trade-offs in power-performance-memory space
One Year Ago There Were Two Distinct Paths
Or ‘Swim Lanes’

- **Many-core**
  - Espoused by Intel, IBM, Sun… ‘Traditional’ CPU manufacturers
  - Characterized by many identical cores on a chip
  - May or may not be cache coherent
  - $O(100)$ cores per socket
  - $O(4)$ threads per core
  - $O(10^3)$ threads per socket

- **Accelerated / Hybrid**
  - Compute sits apart (e.g. a GPU or Cell or Intel MIC on PCI bus)
  - Or is on the same chip (e.g. AMD Fusion, NVidia Denver)
  - May or may not have user-controlled cache
  - $O(1000)$ cores per socket / accelerator
  - $O(10-100)$ threads per core
  - $O(10^5)$ threads per socket
Today The Lines Are Blurred And The Two Paths Appear To Be Converging

- NVidia Denver platform
  - GPU streaming cores + ARM chips
  - Could run OS natively (e.g. Windows 8 on ARM expected by 2012)

- AMD Fusion platform
  - Heterogeneous chip
  - x86 cores + GPUs on socket: No data transfers over PCI bus
  - Runs OS natively

- Intel’s Many Integrated Core (MIC) platform
  - 32 cores-128 threads on a PCI accelerator
  - Attached to host processor over PCI (like GPU)
  - Accelerated streaming mathematical computations

Common theme is greatly increased intra-node task parallelism (multi-threaded) and data parallelism (SIMD vectorized)
**NVidia: Denver Overview**

- **Maxwell will likely be the first released product**
  - Large number of streaming cores + few latency cores
    - Compare to Echelon: 128 SM + 8 Latency Processors, 20.5 TF/s

- **Programmed with CUDA**
AMD: Fusion Overview

- CPU and GPU on one chip
- x86 cores for running serial tasks
- SIMD cores for running vector tasks
- Both types of cores share same global memory, but cache coherency is not guaranteed
- Heterogeneity within a single chip, similar to the CELL processor from IBM
- Programmed with OpenCL
Intel: Many Integrated Core (MIC) Architecture Overview

- Many ‘simpler’ x86 cores
- Connected by high speed interconnect
- Programmed with Intel Parallel Building Blocks
  - Includes ArBB and TBB
  - Single source
  - Targets both Xeon and MIC

Implications For Programming Models / Applications

Programming models need to

- Expose hardware parallelism to application programmer
  - SIMD / data parallelism (vectorization)
  - SIMT / task parallelism (multiple SIMT kernels / multi-threaded)

- Expose memory hierarchy and inform about latency / bandwidth issues
  - Enable transparent data motion for application programmers

- Expose heterogeneity of underlying hardware

- Enable ‘accelerator/host’ paradigm for some classes of machines

- Enable performance portability

- Enable programmers to exploit fine-grained parallelism in applications

- Enable transparent fault tolerant behavior for applications

- Expose different compiler metrics

Most changes are related to power consumption and increased concurrency at the node level
A Simple Element-wise Array Addition In Different Ways

- **Simple Array Addition**
  - Code compiles on traditional CPUs
  - Only parallelism is provided by whatever the compiler can generate
  - Usually compiler does not unroll unless specifically told to do so

- We will look at this in different candidate programming models and look at advantages and disadvantages

```c
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i;
    float a[N], b[N], c[N];
    /*...initialize a and b...*/
    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i];
    /*...do other stuff...*/
    return 0;
}
```
OpenMP: Multi-threaded Shared Memory Parallelism

Advantages

- Portable, but only targets homogeneous multi-core
- CAPS/HMPP will handle accelerated hierarchies
- Simple syntax and allows for incremental parallelization
- Non-invasive
  - Naïve user can get by using only `#pragma` directives
  - Advanced users can embed `omp_..()` calls in their code to probe the underlying architecture, but that can lead to non-portable code

Disadvantages

- Not accelerator friendly (may not be true for long)
- No simple way to enforce NUMA-ness

```c
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i;
    float a[N], b[N], c[N];
    <...initialize a and b...>

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i];
    <...do other stuff...>
    return 0;
}
```
OpenCL: Open Standard For Writing Portable Code Across Multiple Platforms

**Advantages**
- Support from multiple vendors spanning the range from embedded to desktops: likely to survive
- Extension to C/C++
- Portable, targets both multi-core and accelerated / heterogeneous platforms
- Allows for task-based and data-based parallelism
- Allows for per-platform accelerated kernels
- JIT compiling makes codes future-proof

**Disadvantages**
- Support from multiple vendors spanning the range from embedded to desktops: design by committee
- Not very user friendly
- Requires re-architecting entire code base in a host / accelerator paradigm
- Requires cumbersome setup / shutdown of host / accelerator elements
CUDA: Accelerating Codes On NVidia GPUs

## Advantages
- **Single vendor control**: nimble and adaptable
- Extension to C/C++
  (Thrust STL library just released)
- Easier to implement than OpenCL
  (for run-time API)
- Allows for task-based and data-based parallelism

## Disadvantages
- **Single vendor control**: may disappear tomorrow
- Not very user friendly
- Requires re-architecting entire code base in a host / accelerator paradigm
- Requires cumbersome data transfers to and from GPU
- Cumbersome for driver API (which allows for more flexibility and is essentially equivalent to OpenCL)
- Works **only** with NVidia GPUs
Intel Thread Building Blocks (TBB) and Array Building Blocks (ArBB)

- **Advantages**
  - C++ Template based
  - Intended to work across entire line of Intel processors including MIC
  - Easier threading / data parallelism
  - Single source, multiple targets

- **Disadvantages**
  - GPUs, what GPUs?
  - Requires re-architecting entire code
Intel Thread Building Blocks (TBB) and Array Building Blocks (ArBB)

- **Advantages**
  - C++ Template based
  - Intended to work across entire line of Intel processors including MIC
  - Easier threading / data parallelism
  - Single source, multiple targets

```cpp
#include <iostream>
#include <arbb.hpp>

using namespace arbb;

typedef arbb::u8 T;

doctor myAdd(dense<T> a, dense<T> b, dense<T>& out)
{
  out = a - b;
}

int main()
{
  try {
    int size = 100000;
    unsigned char* frame = aligned_malloc(sizeof(unsigned char)*size);
    unsigned char* background = aligned_malloc(sizeof(unsigned char)*size);
    unsigned char* resultArbb = aligned_malloc(size * sizeof(unsigned char));
    dense<T, 1> vec_frame; bind(vec_frame, frame, WIDTH*HEIGHT);
    dense<T, 1> vec_background; bind(vec_background, background, WIDTH*HEIGHT);
    dense<T, 1> vec_result; bind(vec_result, resultArbb, WIDTH*HEIGHT);
    call(myAdd(vec_frame, vec_background, vec_result));
    const_range<T> r = vec_result.read_only_range();
  }
  catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
    return 1;
  }
  return 0;
}
```
SCOUT: Data Parallel Hardware Agnostic Computation

- **Advantages**
  - Research toolchain and language for exploring hybrid computing languages
  - Allows for mixed sequential and explicit parallel regions
  - Compiler determines best match between code and hardware and schedules accordingly
  - Code is handed off to underlying architecture with no intervention from user
  - Write once, run everywhere

- **Disadvantages**
  - Research toolchain – using latest features requires using developer branch rather than release branch
  - Have to rewrite entire code using SCOUT
  - Need to be comfortable with data parallel programming

```c
uniform mesh myMesh[100000] {
cells: float a,b,c;
}
myMesh.a = myMesh.b = 100.0;  // initialize A and B arrays
forall cells c of myMesh[0:100000] {
c = a + b
}
```
Haskell: Functional Programming

**Advantages**
- Easier verification
- Automatic parallelization
- Greater modularity of code
- Separation between programmer and platform
- Functional code with no side effects

**Disadvantages**
- Only compiler available is the Glasgow Haskell Compiler (GHC)
- Single node solution (parallel version using MPI backend coming soon…)
- No mainstream acceptance

```haskell
{-# LANGUAGE TypeOperators, Rank2Types, FlexibleContexts #-}

-- Packages imported to do GPU accelerated computing
import Data.Array.Accelerate as AC
import Data.Array.Accelerate.CUDA as CUDA

-- Create a standard 2D array in Haskell, using unboxed types (like a malloc’d array of integers)
-- The positions are named with tuples, and the values after them, e.g. (0,0).1
-- means put 1 in the upper left cell.
ary :: UArray (Int,Int) Int
ary = IA.array ((0,0),(1,1)) [((0,0),1),((0,1),2),((1,0),3),((1,1),4)]

-- Convert the standard array to an Accelerate host-side array type (using `fromIArray`), and then prepare it for us on the GPU (with `use`)
-- Objects with Acc (GPU side) type live in GPU memory.
accAry :: (Acc (AC.Array (Z:.Int:.Int) Int))
accAry = use $ fromIArray ary

-- Do a row wise reduction and then a column wise reduction (the folds).
-- The ‘run’ can be thought of as the kernel invocation, and the right hand side as a fused set of operations.
retAry = CUDA.run (AC.fold (+) 0 (AC.fold (+) 0 accAry))

-- Print the result
main = do putStrLn (show retAry)
```
Resilience: Fault Tolerant Applications

- **What is computing resilience?**
  - The ability of an application to ride-through faults/errors/failures and continue to compute towards a *correct* solution in a timely manner.
  - Slightly different from fault-tolerance which is focused more on recovering after failure.

- **Fault? Error? Failure?**
  - *Faults* can turn into *errors* which can turn into *failures*. Any of the 3 can be handled and prevented from propagating . . . in theory.
  - Generally speaking, there are two types of faults – *hard* and *soft*
  - Vendors expend a lot of hardware to prevent faults at the hardware level from propagating into errors.

- **Why worry?**
  - **Faults are on the rise, and predicted to continue.**
    - With lower voltages come more faults, particularly soft errors.
    - Smaller feature size cause more faults, particularly due to guard band violations.
    - It’s not just data corruption in memory! We can’t just get around this with ECC!
    - At 20nm we are seeing errors in *logic* units that are equal to memory error rates at 100nm. If we used ECC at 100nm (we did), we have equal reasons to be concerned at 20nm.
    - GPUs are notoriously bad for data corruption – the consumer market doesn’t demand the level of data integrity needed by supercomputing field.
Tunable Fidelity Can Reduce Resilience Overhead

What is Tunable Fidelity?

- Knowing when we need resilience and when we don’t permits us to construct systems that trade resilience for power and/or performance.

- But how do we know?
  - Theoretical algorithmic development.
  - Experiment with tools to simulate and emulate fault-rich environments.
Conclusions - 1

- **The computational node is changing**
  - Power will be a key design factor
  - Increased concurrency: both data and task parallelism
  - Heterogeneous chips will dominate computing over the next decade
  - Flops are free, memory / data movement and power are expensive
  - ‘recompile and run’ no longer guaranteed path to newer architectures

... Continued
Conclusions - 2

- Anticipated change in hardware will create unique challenges for programming models
  - Need to expose hardware parallelism
  - Need to expose memory hierarchy
  - Need to expose heterogeneity
  - Need to expose different compiler metrics
  - Need to be fault tolerant(!)
  - Key is abstraction

- No clear leader in programming models has emerged yet
  - Front runners are OpenMP, OpenCL, CUDA and ArBB/TBB
  - DSLs (like SCOUT) may be one way to ‘future-proof’ code
  - Functional languages like HASKELL are another option

... Continued
Conclusions - 3

- Resilience / fault tolerance will be a key factor in code development
  - Increased concurrency at node level implies that hard (crashes) and soft (silent data corruption) error rates will continue to rise
  - Needs to be dealt with at both application and hardware levels
  - Tunable Fidelity may be the key tradeoff in optimizing resiliency / performance
    - Requires algorithm development
    - Requires experimentation with tools to simulate and emulate fault rich environments
The Major Phases Of Supercomputing As Witnessed On The Top500 List
NVidia Software Solution: CUDA Everywhere

- More on CUDA later

CUDA C: C with a Few Keywords

```c
void saxpy_serial(int n, float a, float *x, float *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
} // Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

__global__ void saxpy_parallel(int n, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
} // Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```
OpenCL

- Targets both ‘CPU’ and ‘GPU’ parts of Fusion processor
- Open standard
- More on OpenCL later
Intel: Many Integrated Core (MIC) Software Solution

Intel® MIC Architecture Programming

Single Source

Compilers and Runtimes

Intel® Xeon® processor family

Intel® Xeon® processor

Intel® MIC architecture co-processor

Eliminates Need for Dual Programming Architecture

Common with Intel® Xeon®

- Languages
- C, C++, Fortran compilers
- Intel developer tools and libraries
- Coding and optimization techniques
- Ecosystem support

Implications For Programming Models / Applications

The issues:

- Greatly increased concurrency / parallelism within a node (‘fat’ nodes)
- Increased levels of memory that are exposed to the user
- Many cores on each socket
- Not all cores may be the same
- Not all sockets may be the same
- Non-uniformity in resource affinity / latency
- Fault tolerance becomes critical
Survey Of Programming Models And Their Readiness For The Architectural Upheaval

List of Programming Models
- OpenMP
- OpenCL
- CUDA
- DirectCompute
- Dacs/ALF
- Intel TBB & ArBB
- DSLs
- SCOUT
- HMPP

Some Languages To Track
- Fortran
- Haskell
Hard Errors

- Much of the computer science research in resilience has gone into this subfield.

- What can we expect from next-generation programming paradigms?
  - Language constructs to trap errors before they turn into application failures

- What can we do to prepare?
  - Design for failures – applications will crash, hardware will break
  - Example: Netflix does this on Amazon’s EC2 Cloud – they have tasks that kill random processes at random intervals. Their programmers are taught to design around this.
Soft Errors

- Much nastier! Have the potential to corrupt data, causing wrong output and, worst case, causing decision makers to take the wrong action.

- This field is *hard!* Not much research has been done in it for ground-based systems.
  - Space-based community has been dealing with this for decades. But their solutions are often:
    - Radiation hardened (radhard) hardware – super expensive, several generations old. These factors fly in the face of conventional supercomputing needs / interests.
    - Redundant (such as triple modular redundancy – TMR). But can we afford to compute the same thing so many times and check answers?

- ... continued
Soft Errors (continued)

- **What can we expect from next generation programming paradigms?**
  - Probably not much. This is going to be a real problem.

- **What can we do to prepare?**
  - Best solution is algorithmic resiliency to soft errors, where possible (theoretically not possible in many fields).
    - Design your application so you can tell when it gets the wrong answer, or at least know the approximate bounds of an error.
    - Check calculations . . . often! Within the code, not just final results.
  - Experiment today with tools that can simulate soft errors in applications and operating systems to start designing for these problems.
    - LANL’s Soft Error Fault Injection Framework (SEFI) can inject logic and memory errors
Who Is Driving This Market?

- **Not Supercomputing!**
  - 2008 video game market: $39.0 Billion
  - 2009 video game market: $60.0 Billion
  - 2009 industry estimate for netbooks: $9.0 Billion
  - 2010 embedded computing market: $6.0 Billion
  - NNSA budget for supercomputers: $0.2 Billion

- **Industry Focus will be on low-power / high performance**

- **Leads to some interesting trade-offs**
<table>
<thead>
<tr>
<th></th>
<th>Opterons today (RR + 5 years)</th>
<th>Opterons in Roadrunner</th>
<th>Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>12</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Threads</td>
<td>48</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Memory per core</td>
<td>$10^7$</td>
<td>$10^7$</td>
<td>$10^5$</td>
</tr>
<tr>
<td>Performance (GF)</td>
<td>100</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>GF/Watt</td>
<td>$\sim 0.5$</td>
<td>0.12</td>
<td>0.72</td>
</tr>
</tbody>
</table>