Hybrid Computing, Past, Present and Future

Eric Petit – University of Versailles

Eric.petit@uvsq.fr

Optimize HPC Applications on Heterogeneous Architectures
Who are we?

• LRC-ITACA
  Université de Versailles St-Quentin-en-Yvelines UVSQ
  French Alternative Energies and Atomic Energy Commission CEA DAM

• Dedicated to HPC
  – Application, compiler and memory hierarchy optimisation, Performance analysis tools
  – Involved in several projects: H4H, ViHPS, Teratec, Exascale
  – We have open positions in the lab

• A master degree specialized in HPC : MIHPS
  – http://mihps.prism.uvsq.fr/
Who are we?

Exascale Computing Research:

Joint lab between Intel, UVSQ, CEA DAM, and GENCI

CTO: Professor William Jalby
Mail: jalby@uvsq.fr

Thematics: runtime, application characterisation, performance analysis, exascale application co-design
What is this presentation about?

- **Past**
  - Computer science short history
  - Historical fact about HPC

- **Present**
  - Some basics about processor architecture
  - What makes a GPU so different?

- **What next?**
  - Current trend and upcoming architectures
    - Xeon Phi, Maxwell and beyond...
  - Impact for application developers?
• Once upon a time...
  – Mechanical computer: **personal computer** until ~60-70
  – Analogic: electronic, mechanic and/or optic computers.
    A measurable physic output following the same equation as the
    target computation (~ASIC of these days)
    => **embedded (bomb dropping...), simulation**
  – Bits-based computer => **HPC**
    • Electro mechanic in the 40's
      e.g. Z3 (first IBM market!)
    • Lamp based:
      – 1945 the famous ENIAC designed by von Neuman and
        considered as the first modern computer
      – Memory was already a huge issue (mercury tunnel, ferrite...)
      – 1955 IBM 704 and FORTRAN designed by Gene Amdhal
        5 kFLOPS => **Physicists are stuck here ;)**
    – 1956: transistors
Computer science is born before current microprocessor technology

- von Neuman 40's “The” model (with Turing in 50's)
- Amdhal 50's “The” law
- Cray 60's the father of modern HPC
  - “Anyone can build a fast CPU. The trick is to build a fast system”
    - Bandwidth oriented design
    - Cray-1 Vector processor
- And many others....
HPC History

• Meanwhile in HPC...
  – HPC is the formula one of computer design
    • Testing extreme solution
    • (almost) No trade off
    • (used to be) First place to experiment architecture improvements
    • Some technologies will never go to mass market (?)
  – HPC solution are not only a matter of processing units
    • Network
    • Infrastructure
      – Cooling
      – Power
      – Set-up and recycling
    • Politics and finance
HPC History

- HPC top 500:

Source: Top500 report June 2012
HPC History

#1 top500 Pmax(flop)/Power

#1 top500 Power
<table>
<thead>
<tr>
<th>Year</th>
<th>Power</th>
<th>TFLOPS</th>
<th>TFLOP/MW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray-1</td>
<td>0,115</td>
<td>0,8</td>
<td>6,95</td>
</tr>
<tr>
<td>ASCI Red</td>
<td>0,85</td>
<td>1,06</td>
<td>1,24</td>
</tr>
<tr>
<td>ASCI White</td>
<td>3</td>
<td>12,3</td>
<td>4,1</td>
</tr>
<tr>
<td>EarthSimulator</td>
<td>3,2</td>
<td>122,4</td>
<td>38,25</td>
</tr>
<tr>
<td>Bluegene/L</td>
<td>1,4</td>
<td>280</td>
<td>200</td>
</tr>
<tr>
<td>RoadRunner</td>
<td>2,35</td>
<td>1040</td>
<td>442,55</td>
</tr>
<tr>
<td>Jaguar</td>
<td>6,96</td>
<td>1760</td>
<td>252,87</td>
</tr>
<tr>
<td>K</td>
<td>9,8</td>
<td>8700</td>
<td>887,75</td>
</tr>
<tr>
<td>K</td>
<td>12,6</td>
<td>10500</td>
<td>833,33</td>
</tr>
<tr>
<td>Sequoia</td>
<td>7,9</td>
<td>16320</td>
<td>2065,82</td>
</tr>
</tbody>
</table>
• Why power is such an issue:
  – Sequoia power consumption assuming 10%-off a year => 63 GW/h
  – Dongarra approximation for the first year is 8 M$/Y
  – Average public price in US ~15cts/kw => 9,45 M$/Y
  – If we keep the same (very approximate!) trend for exascale
    => 24 MW, 28 M$/Y if the electricity price remain the same...
      • ~ 60 wind-power generator, not so much if you have wind...
      • 120 hectares of average solar panels in France
      • 1,8% of an average french nuclear power plant
• Why power is such an issue:
  – Sequoia power consumption assuming 10%-off a year
    => 63 GW/h
  – Dongarra approximation for the first year is 8 M$/Y
  – Average public price in US ~15cts/kw => 9,45 M$/Y
  – If we keep the same (very approximate!) trend for exascale
    => 24 MW, 28 M$/Y if the electricity price remain the same...
      • ~ 60 wind-power generator, not so much if you have wind...
      • 120 hectares of average solar panels in France
      • 1,8% of an average french nuclear power plant

=> HPC will probably not go green...
• Why power is such an issue:
  – Power density
    • More than a rocket nozzle
    • Impossibility to power up the full-chip! => “black silicon”
    • 3D-stacking impact?
  => Huge challenge for founders

– Single core performance
  • Forget about higher frequency
    – $O(F \cdot V^2)$, and $V$ nearly stop decreasing
    – Higher frequency needs higher voltage
    – Thinner transistors “should” require lower voltage
  • Keeping the same frequency already eats all the effort

=> There is no other way than going parallel... (for now)
Current Challenges

• Main challenges for Exascale
  – Power
  – Parallelism
    • Algorithms
    • Numerical precision!
  – Reliability/Resiliency
    • Fault tolerant HW
    • Fault tolerant Software
  – Programmability
  – But also OS, runtime, file system etc...
Some theoretical background

- System design is based on very few basic rules including:
  - Moore's law
  - Amdhal's law
  - Gustafson's law
  - Little's law
Moore's law

• “The number of transistor in a microprocessor / surface unit double every two years”
  - Gordon Moore, Intel founder
• Law → laws: people are citing extrapolated conclusion: memory size, performance...
• The original law is still valid with multicore
  • Moore's law should stop with transistor shrinking limit => yes and no...
  • Uniprocessor performance double every 18 month => dead

Source: wikipedia
Moore's law

• Nvidia GPU:
  • Kepler 7.1 billion (2012/3)
  • Fermi GF100 3 billion (2010)
  • GeForce 9 G92 1.4 (2008)
  • GeForce 8 : G80 0.745 (2007)
• The trend is getting closer to Moore's law

• Kepler GPU die size 550 mm² => 12.1 Mt/mm²
• Sandybridge E CPU die size 435mm², 2.27 Bt=> 5 Mt/mm²
• Despite higher transistor size, the density is ~2.5 fold better than CPU.

• Why and how?
  • Lower frequency => thinner wiring
  • Much less memory => less wire / network (crossbar)
  • Simpler design => easier to organise
  • Power is naturally (i.e. needed from the user to get performance...) spread on all the GPU => no (less) hotspot
• Moving to many core helps keeping close to Moore's Law
  • + a little help for power

• Same in intel's GPU design
Moores law

- DAC 2012 Keynote: Designing a 22 nm Intel® Architecture Multi-CPU and GPU Brad Heaney

Process technology needs for CPU Core vs GFX

- **Core is architected to be a narrow & fast:**
  - Higher frequencies
  - Faster and bigger devices
  - Taller std-cell library
  - Dense power grid & Wider metals

- **Gfx is architected be wide & slow:**
  - Area & Leakage are more critical
  - Smaller and lower leakage devices
  - Shorter std-cell library,
  - Dense layout & Narrower metals

- **Future Trend:**
  - Wider Engines (Frequency less critical)
  - Emphasis is on lower power through lower voltage
  - Shorter libraries, Denser layout & Narrower Metals
  - Higher variation especially for smaller devices

Copyright © 2012 Intel Corporation
Amdhal's law

- Limited speed-up: **sequential wall**
- Diminishing Return On Investment (ROI)

\[
Ta = Ti \times s + \frac{(1-s) \times Ti}{P}
\]

\[
SU = \frac{1}{s + \frac{1-s}{P}}
\]

Amdhal's law

- How many cores in a chip? Why not putting 16, 32 in current CPU?
  - CPU are primarily designed for desktop rather than HPC:
    => the right amount today is 4, trust Intel's guys ;)
      - Not so much parallelism in current desktop app
      - Shared memory bandwidth => sequentialization of memory accesses
      - Need to scale other resources accordingly (memory)...
    - or, like GPU, constraints the code that fit (FLOP/Byte for bandwidth, limited working set)
  - Double vector size for compute-intensive parts
  - MIC, aka Xeon Phi is for HPC => 50+ cores
Amdhal's law

- **Amdhal and power**
  - If the code is parallel enough, using one more processor is more power efficient than using higher frequency.
  - Until 4 cores, if 60% of the code is //, increasing the number of cores is better ((2,0.5),(3,0.45),(4,0.42)...) 

<table>
<thead>
<tr>
<th>Seq</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>1,81</td>
<td>2,5</td>
<td>3,07</td>
<td>3,57</td>
<td>4</td>
</tr>
<tr>
<td>0,25</td>
<td>1,6</td>
<td>2</td>
<td>2,28</td>
<td>2,5</td>
<td>2,66</td>
</tr>
<tr>
<td>0,5</td>
<td>1,33</td>
<td>1,5</td>
<td>1,6</td>
<td>1,66</td>
<td>1,71</td>
</tr>
<tr>
<td>0,75</td>
<td>1,14</td>
<td>1,2</td>
<td>1,23</td>
<td>1,25</td>
<td>1,26</td>
</tr>
</tbody>
</table>

Power at higher frequency for eq SU:

- \( P = F \times (SV + DV)^2 \)
- \( SV = \frac{1}{3} \times V \)
- Dynamic voltage
  - \( DV = \frac{2}{3} \times V \)
- \( D = O(f) \)
- \( VP = S \times (\frac{1}{3} + \frac{2}{3} \times S)^2 \)
Amdhal's law

• Amdhal and power
  – What if the code is not parallel enough:
    • Shut off unused core and use others => power saving
  – What if the code is not always parallel:
    • Use more cores when it is parallel
    • Run high frequency sequential core on the rest
    => run HYBRID

• See also: Extending Amdahl’s Law for Energy-Efficient Computing in the Many-Core Era, Dong Hyuk Woo and, Hsien-Hsin S. Lee
Gustafson's law

• Assumption: the size of work to do varies linearly with the number of processors
• Idea: solve larger problem within the same amount of time
• Optimistic point of view but what about a problem which doesn't fit the assumption?
  • Algorithm wall

\[
SU = P - seq \times (P - 1)
\]

• Gustafson => weak scaling
• Amdhal => strong scaling
Little's law

• Concurrency = latency * bandwidth
  • New interpretation with massively parallel architecture
  • How many compute unit do I have to run in // to hide latency?
  • Co-design

---

e.g. Vasily Volkov work on GPU:
• On G80, latency of SIMD float instruction: 24 cycles
• Throughput 1 every four cycles: ¼
• 6 SIMD instruction to schedule per SM per cycle
• 1 SM = 8 cores so 6 warp will do it
  32 threads per warp => 192 threads / SM

For more details check:
http://www.eecs.berkeley.edu/~volkov/volkov10-PMAA.pdf
Or other related work
• So where are we?
  – Current low level design
  – Current node design
  – Current system design
Computer Architecture in General

- System design consist on optimizing the whole system
  - Bottom up evolution: current trend, architecture was forced to move
  - Top down evolution: used to be, but now less frequent e.g. co-design, ASIC, FPGA, MIC~
Low level basics

- Common concept to CPU/GPU/MIC
  - ALU/FPU/register
  - Memory Hierarchy
  - Pipeline
  - In order / out of order
  - SIMD, SIMT
  - SMT
Low level basics

- Very basic architecture design:
  - ALU/FPU/register/memory
Memory design

- Main Memory has a high latency => Memory hierarchy
• Hard Drives are slow => use fast RAM memory
• But RAM is much slower than core L2 cache
  – Concurrent accesses between 2 cores?
Memory design

• Add L3 cache shared between cores on die
  – Faster core to core communication (≠ AMD)
  – Lower latency
Memory design

- Monolithic shared caches don't scale...

- NUCA cache!
  - Faster (latency)
  - Higher bandwidth
  - Simpler and scalable design
Cache design

• How does a cache memory work?
Cache design

- How does a cache memory work?
Cache design

- How does a cache memory work? => locality
  - If the cache is full you replace an existing line => capacity
  - Replacement policy

Main Memory

@1
@2
@3

Cache
@1
@2
@3

CPU

Loop i:
F(@1)
Loop i:
G(@i)

Temporal
Spatial
Cache design

- Lower @ bits index + Offset

» Direct mapped cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>@1(0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>@1(1)</td>
</tr>
</tbody>
</table>

Main Memory

@1 = 01 0011 0
01 0011 1

@2 = 11 0011 xx

Loop i:
F(@1)
G(@2)
• What if the two addresses share the same lower bits?
  » Same index byte = same line

Main Memory

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
</table>
|        | 11  | \$@1()/@2() \$

CPU

Loop i:
  F(\$@1())
  G(\$@2())
• **Aliasing, associativity miss**

  » Similar to **4k-aliasing** (WAR in WB buffer)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>@1()</td>
</tr>
</tbody>
</table>

Main Memory

```plaintext
@1=01 0011xx
@2=11 0011xx
```

Loop i:
F(@1)
G(@2)

CPU
Memory design

- **Aliasing => need associativity**
  - » N-way associative cache
    - Typically 8-way L1, 4-way L2, direct map L3

<table>
<thead>
<tr>
<th>Way 00</th>
<th>Way 01</th>
<th>Way 11</th>
<th>Tag 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@1=01 0011xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Way 00</td>
<td>Way 01</td>
<td>Way 11</td>
<td>Tag 10</td>
</tr>
<tr>
<td>01</td>
<td>@1</td>
<td>11</td>
<td>@2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- Different replacement policies such as Last Recently Use (LRU) or Round Robin
• **Shared memory vs private cache** => True and false sharing

<table>
<thead>
<tr>
<th>Index</th>
<th>Way 00</th>
<th></th>
<th>Way 01</th>
<th></th>
<th></th>
<th>Index</th>
<th>Way 00</th>
<th></th>
<th>Way 01</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tag</td>
<td></td>
<td>Value</td>
<td></td>
<td></td>
<td>tag</td>
<td></td>
<td>Value</td>
<td></td>
<td></td>
<td>tag</td>
</tr>
<tr>
<td></td>
<td>01@1</td>
<td></td>
<td>1@1</td>
<td></td>
<td></td>
<td>11@2</td>
<td></td>
<td>11@2</td>
<td></td>
<td></td>
<td>01@1</td>
</tr>
</tbody>
</table>

- **P1:** write @1
- **P2**
Memory design

- **Shared memory vs private cache => True and false sharing**

<table>
<thead>
<tr>
<th>Index</th>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>@</td>
<td>@</td>
</tr>
<tr>
<td></td>
<td>1'</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>@</td>
</tr>
</tbody>
</table>

P1: write @1

<table>
<thead>
<tr>
<th>Index</th>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>@</td>
<td>@</td>
</tr>
<tr>
<td></td>
<td>1'</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>@</td>
</tr>
</tbody>
</table>

P2

<table>
<thead>
<tr>
<th>Index</th>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>@</td>
<td>@</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>@</td>
<td>@</td>
</tr>
</tbody>
</table>

Invalid
Memory design

• Shared cache => True and false sharing

<table>
<thead>
<tr>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td>01</td>
<td>@1</td>
</tr>
<tr>
<td></td>
<td>1'</td>
</tr>
<tr>
<td>11</td>
<td>@2</td>
</tr>
</tbody>
</table>

Index

P1: write @1

<table>
<thead>
<tr>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td>01</td>
<td>@1</td>
</tr>
<tr>
<td></td>
<td>1'</td>
</tr>
<tr>
<td>11</td>
<td>@2</td>
</tr>
</tbody>
</table>

Index

P2:...

Read @1
Memory design

- Shared cache => True and false sharing

<table>
<thead>
<tr>
<th>Index</th>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
<td>tag</td>
</tr>
<tr>
<td>01</td>
<td>@1</td>
<td>@1'</td>
</tr>
</tbody>
</table>

P1: write @1

<table>
<thead>
<tr>
<th>Index</th>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
<td>tag</td>
</tr>
<tr>
<td>01</td>
<td>@1</td>
<td>@1'</td>
</tr>
</tbody>
</table>

Comm

<table>
<thead>
<tr>
<th>Index</th>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
<td>tag</td>
</tr>
<tr>
<td>01</td>
<td>@1</td>
<td>@1'</td>
</tr>
</tbody>
</table>

P2:...

Read @1 => true sharing
Memory design

- **Shared memory vs private cache => True and false sharing**

<table>
<thead>
<tr>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>@1</td>
</tr>
<tr>
<td></td>
<td>@1'</td>
</tr>
<tr>
<td>11</td>
<td>@2</td>
</tr>
</tbody>
</table>

P1: write @1

<table>
<thead>
<tr>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>@1'</td>
</tr>
<tr>
<td></td>
<td>@1</td>
</tr>
<tr>
<td>11</td>
<td>@2</td>
</tr>
</tbody>
</table>

P2:...
- **Shared memory vs private cache** => True and false sharing

<table>
<thead>
<tr>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td>01</td>
<td>@1</td>
</tr>
</tbody>
</table>

P1: write @1

<table>
<thead>
<tr>
<th>Way 00</th>
<th>Way 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Value</td>
</tr>
<tr>
<td>01</td>
<td>@1</td>
</tr>
</tbody>
</table>

Invalid

P2:...

Read @1'
• Shared memory vs private cache => True and false sharing

P1: write @1

P2:...
Read @1' => false sharing
Memory design

• Also some physical issues
  » Read/write ports
  » Banking (not only for caches)

<table>
<thead>
<tr>
<th>Index</th>
<th>Way 00</th>
<th>Way 01</th>
<th>Way 11</th>
<th>Tag 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>@1</td>
<td>11</td>
<td>@2</td>
<td>@3</td>
</tr>
<tr>
<td>01</td>
<td>@4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Most of the time
consecutive line in
distinct bank (locality)

2 read port / bank
1 write port / bank

If we access @1,2,3
At the same time
=> Bank conflict
=> also on GPU

Read: @ modulo=>bank
Read index
Tags compare
If hit return value
Else miss&&ask higher Memory

F(@1)|G(@2)|H(@3)=> 3 cycles
F(@1)|J(@4) => 1 cycle
Memory design - CPU

- Memory is IMPLICIT on CPU
  - Data hierarchy fetches the data from memory for you
    - (almost) No control on the cache content
    - Memory mapping need tricks (e.g. first touch policy)
  - HW mechanism to accelerate the process
    - Adjacent pages
    - Adjacent cache line
    - Prefetcher
    - Victim cache
    - Multi-way concept
    - ....
  - Interact with software => how to optimize?
Memory design – Efficient usage

• Be aware of the memory hierarchy
  – Temporal locality (Blocking, tiling...)
  – Spatial locality (Data structure...)
  – Vectorized load
  – Software prefetch (Intrinsic, directive...)
  – Avoid data sharing (tree reduction ...)
  – First touch policy (// initialization)

• Hide L/S latency by computation, do use too many array (memory stream) at a time
  – Unrolling (and jam)
  – Loop fusion
  – Loop splitting
  – Software pipelining...
• Memory is EXPLICIT on GPU
  – You explicitly pull the data down to the computing unit and push it back to memory
  • Throughput oriented data-flow model (2-3x /CPU)
  – Caches are very recent on GPU and different from CPU: RO cache, possibility to switch part of the L1 into shared memory
  – Less logic more efficiency, rely on programmer/compiler
Memory design

- Get the data close to you (temporal locality)
  - Use registers (and don't waste them)
    - Minimize occupancy to match the needed concurrency (Little's law)
  - Use shared (copy global to shared before computing)
- Use spatial locality
  - Neighbour threads access neighbour data
    - Irregular access application with no data locality?
  - Be careful about banking (old GPU, new?)
  - Don't share data between SMX
- **COMPUTE INTENSITY**
  - Algorithmic changes
  - Loop fusion/grouping (even with synchronization)

=> Many tutorials on this topic including NVIDIA ones
More about registers:

• Few of them because of design and power consumption issue
  – CPU 144+160=\textbf{304} per core
  – GPU \textbf{344} (65536/192c) per core? No you can access max 255 per core, why?
    • Registers are attach to threads, not cores!
    • Register are shared between concurrent thread
    • Only 32 per thread if you use max concurrency
    • More register if you use less thread
    • Needed since the closest memory for spill-fill in GPU is ~40cycles far!
More about registers:

• Problem: graph coloring is NP complete
  – Spill-fill (saving to stack) can cost a lot
  – Use heuristic algorithm
    • And some are really bad...

• Register pressure is a variable to optimize
  • Coloring at ~BB unit => be careful with optimization which enlarge them (unrolling)
  • On GPU register file is shared between threads => minimize occupancy = maximize registers per thread
    – e.g exotic (and funny) register design: rotating registers on Itanium
Die design

• We are now aware of how code and data get to the compute unit, what happen next?

• Transistor count:
  – 26 M per core on 10c Westmere CPU (32 nm)
  – 28 M per core on 8c Sandy-bridge EP (28 nm)
  – 2.4 M per core on 2880c Kepler GPU (28 nm)

• Total memory on chip (including register):
  – 8c Sandy bridge EP ~24 Mo
  – 2880c Kepler ~10 Mo

=> design are slightly different, usage should be!
A closer look:

- What makes different CPU core and GPU core (SM):
  - **CPU**
    - Capture a maximum ILP: MIMD, OoO, SIMD
    - Optimize Latencies:
      - **Pipelined** design at all level
      - Memory **prefetcher**
      - **Branch predictor**
      - **Loop buffer, victim cache**...
  - **GPU**
    - Use “existing” massive parallelism (gridification)
      - **SM(X), WARP, Thread**
      - **SIMT (+SIMD)**
    - Throughput oriented
      - Use thread concurrency to mask latency
      - Execute all path and **predicate**
      - Explicit data management
    - Rely on user code and compiler quality ( : ( )

Die Design
Exploit parallelism - Flynn Taxonomy

- SIMD, Single Instruction Multiple Data
- MIMD, Multiple Instruction Multiple Data
- SISD, Single Instruction Single Data
- MISD, Multiple Instruction Single Data
Exploit parallelism - Flynn Taxonomy

Scalar processor
von Neuman model

SuperScalar processor
VLIW

SISD

MISD

~Systolic array
Redundant system

Vector Instruction
~GPU

MIMD

SIMD
Exploit parallelism - Flynn Taxonomy

- Some “refinements” are proposed from time to time like SIMT, Single Instruction Multiple Thread => GPU
- Architecture can use many paradigms at different granularities, ex SSE on superscalar or SIMD in SIMT

- Exploiting parallelism:
  - ILP: Instruction Level Parallelism
  - TLP: Task Level Parallelism
    - Fine grain: e.g. thread
    - Coarse grain: e.g. MPI process
Exploit parallelism - ILP

- ALU and FPU and Memory operation can work in parallel, how to do it?
  - Introduce Pipeline to benefit from ILP
  - Idea, compute an instruction can be split into steps

Hierarchy

Memory, code+data

Control Logic

ALU

FPU

Registers

Add Logic for Pipelining here
Exploit parallelism - ILP

- Fetch Instruction
- Decode the instruction
- Fetch the operand
- Execute
- Write back the result
Exploit parallelism - ILP

- Imagine a tube:
  - Latency: time to go from entry to exit
  - Throughput: output element per time step
- The more stages the thinner the time step, limit? (P4...)
  - Reality is much more complex
Exploit parallelism - OoO

- A short example explaining why this not as simple...

<table>
<thead>
<tr>
<th>Fetch Inst</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Fetch Op</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop:
- R2 = LOAD R1+0 // 1
- R3 = R3 ADD R2 // 2
- R1 = R1 ADD 4 // 3
- R4 = R4 SUB 1 // 4
- BNZ R4, boucle // 5

IPC 5/(5+2) = 0.71
Exploit parallelism - OoO

- Need to change execution order:
  - Statically => SW
  - Dynamically => JIT, HW

<table>
<thead>
<tr>
<th>Fetch Inst</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Fetch Op</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Mem</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Mem</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>WB</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Loop:
- R2 = LOAD R1+0 // 1
- R1 = R1 ADD 4    // 3
- R4 = R4 SUB 1    // 4
- R3 = R3 ADD R2   // 2
- BNZ R4, boucle   // 5

IPC 5/5 = 1
Exploit parallelism - OoO

- In order execution => the compiler or the user tune the instruction scheduling
- How to do it in HW => Out of Order execution
  - Fetch and decode many instruction at a time
  - Store it in buffer at the stage that fetch operand
  - Dispatch those that are ready to execute
  - Enhancement
    - More physical register than logical register
    - Use renaming to break false dependency (Tomasulo 1967...)
- More ILP in the pipeline (fixed design)
  => more execute units to dispatch
  => issues more than one execution / cycle

- Now you should understand “4-issue superscalar processor”
Enhanced pipeline: Remove “bubbles”
- Branch prediction
- Out of order execution
- Hide Memory latency with concurrency (Little's law)
- Many path pipeline (Int, float etc...)
- Loop buffer ...

Enhanced ILP:
- Longer pipeline
- More than one unit of each kind
- Out of order execution
- Compiler optimization like unroll or software pipelining

Alternative way for ILP: Vector, SMT...
Exploit parallelism - SIMD

- SIMD commonly implemented as vector operations:
  - Adjacent memory location input
  - Adjacent memory location output
  - 4 classes: vector arithmetic and logic + scatter + gather + shuffle
  - AVX, SSE, MMX, Altivec...

\[
\begin{align*}
\text{A}[1] & \quad \text{B}[1] & \quad \text{A}[1]+\text{B}[1] \\
\text{A}[2] & \quad \text{B}[2] & \quad \text{A}[2]+\text{B}[2] \\
\text{A}[3] & \quad \text{B}[3] & \quad \text{A}[3]+\text{B}[3] \\
\text{A}[4] & \quad \text{B}[4] & \quad \text{A}[4]+\text{B}[4]
\end{align*}
\]

Vector AL

\[
\begin{align*}
\text{A}[1] & \quad \text{A}[2] & \quad \text{A}[1] \\
\text{A}[2] & \quad \text{A}[4] & \quad \text{A}[1] \\
\text{A}[3] & \quad \text{A}[1] & \quad \text{A}[1] \\
\text{A}[4] & \quad \text{A}[3] & \quad \text{A}[1]
\end{align*}
\]

Shuffle

\[
\begin{align*}
\text{A}[1] & \quad \text{A}[1] & \quad \text{F}(\text{A}[1], \text{A}[2], \text{A}[3], \text{A}[4]) \\
\text{A}[2] & \quad \text{A}[2] & \quad \text{F}(\text{A}[1], \text{A}[2], \text{A}[3], \text{A}[4]) \\
\text{A}[3] & \quad \text{A}[3] & \quad \text{F}(\text{A}[1], \text{A}[2], \text{A}[3], \text{A}[4]) \\
\text{A}[4] & \quad \text{A}[4] & \quad \text{F}(\text{A}[1], \text{A}[2], \text{A}[3], \text{A}[4])
\end{align*}
\]

Gather

\[
\begin{align*}
\text{A}[1] & \quad \text{A}[1] \\
\text{A}[2] & \quad \text{A}[1] \\
\text{A}[3] & \quad \text{A}[1] \\
\text{A}[4] & \quad \text{A}[1]
\end{align*}
\]

Scatter
Exploit parallelism - SIMD

- Problem with vector operation:
  - Computations have to be independent
    - Inter-iteration dependancy?
  - Data structures are heavily constrained
    - Adjacent
    - Aligned (with ½ line of cache)

=> compiler are not so good at vectorizing
- Check the output! (-opt-report in ICC)
- ICC has the best vectorization result
- Use compiler flags
- Use directive to improve your code (#pragma ivdep)
- Rewrite your code to ease vectorization: requires experience
- If there is no other way and you really need it, use intrinsics
Exploit parallelism - SIMD

• Vectorization direction:
  – Same array independent iteration
  – Different arrays dependent iteration

On CPU:
If it pay off reshape the data
- Local shuffle operation
- or at global level (but be careful
  RB struct can be very bad in other
cses)
Similar to coloring for MPI
On GPU:
• Don't even need to reshape

for i
  a[i]=b[i]+c[i]

\[
\begin{align*}
\end{align*}
\]

for i
  a[i]+=a[i-1]
  b[i]+=b[i-1]

\[
\begin{align*}
\end{align*}
\]
Exploit parallelism

• Until now, we only deal with single core parallelism
  – Still an active problem!
• What about exploiting parallelism at coarser grain
  – Pushing farther single core => SMT
  – Multicore => MT
  – Manycore => ???
    • On Nvidia MIMD WARP and SIMT threads
Exploit parallelism - SMT
Exploit parallelism - GPU

- Work is shared in blocks
  - Multiple blocks are assigned to one SMX
  - A block is divided into WARP of 32 threads
    - Warp of different block can be scheduled at the same time
      => MWMD
      => the pool of thread is shared between blocks
  - Threads of a same WARP are executed in SIMT
- This is very complex, and you have to take care of it
Exploit parallelism - MT

- Node design: context for threads based parallelism
- What to do in extra threads:
  - Share the work in tasks
    - Symmetric
    - Steps (pipeline like) => stream
    - Asymmetric collaborative task
    - Slave/helper task
- ! I use task as a generic term. Not the “task” as openMP one
  - “Task”-based runtime such as openMP 3.0
    - Worker thread
    - Task queue
    - Work stealing
- Task scheduling
  - Many solutions implemented in runtimes
  - Not the purpose of the lecture
  - But consider following pointers: OpenMP, OmpSS, StarPU, MPC, HMPP...
Exploit parallelism – Multi Level

But also MIMD+SIMD on CPU
Exploit parallelism – CPU SPMD

- At a software level, using openMP, TBB, Cilk...
  - Expose symmetric task to execute simultaneously
e.g. parallel for, iteration are independent and can be done in parallel
  - Express synchronisation if needed

- At runtime level
  - The runtime produce one thread with its own instruction stream, stack...
  - Work is shared between threads
  - Threads are spread onto the HW (scatter, compact, fixed...)

- At system level:
  - Scheduling (can interact with runtime)
  - Memory allocation (first touch...
Exploit parallelism – CPU SPMD

- OpenMP parallel for:
  - Implicit barrier at the end
  - Load balancing?
Exploit parallelism – GPU SIMT

• From SIMD to SIMT: not the same granularity
  – Feed all threads with the same instruction
  – Each thread take care of its own data
  – Doesn't need to be adjacent
  – Doesn't need to be aligned (Bank issue and coalescing almost disappear)
  – But need locality so that you can explicitly copy the data close to your set of threads and they can easily share between them if needed

• SPMD vs SIMT?
  – SPMD is a software based model, HW is not aware of it
  – SIMT, HW is executing one instruction stream in many thread
• A SIMT specific issue: divergent path

Code
```c
x = 0;
// Uniform condition
if(tid > 17) {
    x = 1;
}
// Divergent conditions
if(tid < 2) {
    push if(tid == 0) {
        push x = 2;
    } pop
    else {
        push x = 3;
    } pop
} pop
```

Mask Stack
1 activity bit / thread
```
1111
```
tid=0
```
1111
```
tid=2
```
tid=1
```
tid=3
```
```
Node design example – Cray XE6

Data to check

NUMA=
Non Uniform Memory Access

Non uniform bandwidth
Non uniform latency

Distributed shared memory: HW coherency protocol
Same as XE6 but with 2 NV GPUs

The link to the GPU is the weak point!

Will it get better?

Interesting example because you can compare same node technologe with and without GPU

See also:
acss workshop 2012
ORNL Titan
HLRS Hermi
CPU and GPU does not share the same address space!
• User needs to explicitly copy/copy back the data
• Runtime can do it in software
• HW guys are thinking about it
• Parallelism vs ...

• Other important topics
  – Be fast is cool, be right is better:
    • FPU and FP standardization
    • Numerical validation of algorithm e.g.:
      – Parallelisation of reduction
      – Parallel evaluation of polynomial
  – Fault tolerance
    • HW, SW error
    • HW, SW recovery mechanism
  – Resource sharing
    • Queueing effect
    • Concurrent access: race condition
    • Saturation
In a not so far future

• Heterogeneous many-core
  – Consensus: Big and fast core performing sequential part and many small cores for // parts
  • “Extending Amdahl’s Law for Energy-Efficient Computing in the Many-Core Era” Woo and Lee
  • Cell BE example
  • André Seznec ERC grant proposal
  • Intel public road map
  • Nvidia + ARM? (rumor and project...)
  – Shared Memory? Yes, NUCA, but for how long?
    • Transactional memory?
  – Freq? Lower but boost on sequential short period of time
  – NUMA RAM? Still need intermediate level between nfs/hdd, and on-die memory
  – Out of core accelerators? GPU?
• IBM Cell Broadband Engine architecture lesson:
  – Hybrid design: 1 “large” superscalar PPE and 6 small vector processors
    SPE => very high performance for a single chip
  – A complex design
    • Explicit DMA based transfer
    • Complex vector instruction set
    • Unpredictable 7 bidirectional ring network
    • Too few memory per core
  – Programming: a funny puzzle to solve
    • But what about production development
    • No good compiler
      – Very innovative architecture as Itanium was...
  – Supportive market:
    • HPC, but very few
    • PS3 game console, but for how long?
    • No longterm output
=> Sony stop the production,
Programming future many-core

• HPF “failure”
  • K Kennedy “The rise and fall of High Performance Fortran: an historical object lesson”
  • Sakagami and Murai “The rise and fall of High Performance Fortran in Japan - Lessons learned from HPF”
    – Overselling:
      • Don't need to rewrite
      • If the compiler doesn't use the directive your code will run as sequential version
      • Compiler will generate “good” code
        => reality doesn't match the expectation
    – Based on immature language
      • Fortran90 was not ready, bad performance, bug
      • No tool F77 → F90
      • People have to rewrite their application if they want performance in return
    – Rely on compiler for performance
      • No entry point for Hand-tuning
Programming future many-core

• What about:
  – CUDA?
  – OpenCL?
  – OpenHMPP?
  – OpenACC?
• The (immediate) future:
  – Jurassic Park: MPI/OpenMP
  – Improved runtime
  – Unified runtime such as MPC
• Challenging and very promising solution:
  – Cilk, TBB, ompSS => task programming
    • Cilk+MPI?
Programming future many-core

The world outside HPC: commercial desktop software, games, embedded system

### DEVELOPERS MOVING TO OPEN STANDARDS... ACROSS THE WORLD!
EVANS DATA JUNE 2011 DEVELOPER SURVEY

<table>
<thead>
<tr>
<th>NORTH AMERICA</th>
<th>EMEA</th>
<th>APAC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#2</td>
<td>#3</td>
</tr>
<tr>
<td>Count</td>
<td>Count</td>
<td>Count</td>
</tr>
<tr>
<td>Intel Threading Building Blocks</td>
<td>57</td>
<td>91</td>
</tr>
<tr>
<td>OpenMP</td>
<td>41</td>
<td>91</td>
</tr>
<tr>
<td>Intel Parallel Building Blocks</td>
<td>39</td>
<td>72</td>
</tr>
<tr>
<td>CUDA®</td>
<td>24</td>
<td>59</td>
</tr>
<tr>
<td>Intel Cilk Plus</td>
<td>24</td>
<td>56</td>
</tr>
<tr>
<td>MPI</td>
<td>19</td>
<td>50</td>
</tr>
<tr>
<td>Co Array Fortran</td>
<td>8</td>
<td>34</td>
</tr>
<tr>
<td>Other</td>
<td>140</td>
<td>145</td>
</tr>
<tr>
<td>Total Responses</td>
<td>402</td>
<td>653</td>
</tr>
</tbody>
</table>

Note that this multiple response question allowed the developers to select as many responses as they wished, and thus the total number of cases will not come to 100%. The response column shows the percent of total responses, while the case column shows the percent of actual developers (cases) who responded.

• Key challenges:
  => Adaptability
  => Virtualisation
  => Resiliency?
Programming future many-core

• More powerful runtime
  – Todays new trend:
    • StarPU: resource load balancing)
    • HMPP: Distributed memory management, data mirroring, resource abstraction)
    • MPC: unified runtime unlock multi-level optimisation e.g. HLS, microthread-MPI

• More powerful hardware
  – Scheduler? e.g. improved block scheduler in NV
  – Transactional memory support (Haswell?)
Programming future many-core

- Future runtime:
  -> Scheduling, balancing, placement are moving to hardware
  -> Dealing with distributed memory
Programming today, thinking about future

- Do I need low level programming and/or intrinsic?
  - NO
  - … except if you are paid for it
  - What should I do then?
    - Wait compiler improvement and/or the next architecture
    - Ask an expert
    - Use libraries...

- Do I need to take care of the underlying architecture?
  - Taking full advantage of a micro-architecture will become a nightmare
  - But application/algorithm will have to be architecture aware...
  - Parallelism pattern that match the architecture topology has to be exposed. For the rest relies on expert, compiler and libraries.
A few words about libraries

- Why you should use library first:
  - Sorry, you are not as good as these guys...
  - Tested for you and it comes with specifications!
  - Updated transparently
  - Better than compiler output on a naïve code
  - Prefer constructor library: e.g. mkl

- When do you NEED to do your own code:
  - When you are doing something very very specific
    - CORNER CASES
  - If you are in a hurry and libraries are not mature
  - When you are the library developer...
A few words about compilers:

- Compilers are always late...
  - Compiler becomes good when the architecture is over
  - Usually good enough during the CPU lifetime
  - Sometimes not...: Itanium, Cell, first CUDA
  - Very complex software that need to be tuned on the architecture
  - Parallelization have always been an issue: it is getting worst...

- Compiler needs to be driven by the user
  - Check the output quality! => MAQAO
  - Use flags and directives!

- Compiler are not equal => try them all!
Compilers trend:
- Retargetable
- Modular (like LLVM, not GCC)
- Profile guided
- Machine learning, auto tuning
- JIT
  => Dynamic and retargetable
  => Compiler are moving into runtime and even architecture
A few words about language

• FORTRAN is difficult for compilers
  – It implies also poor semantic for compiler error and warning...

• FORTRAN is difficult to maintain

• FORTRAN is not well standardized: different compiler, different results (when it compiles)

• FORTRAN is objectively difficult to read

• FORTRAN is difficult to interface with other languages
  – Types translation
  – Always the last to benefit from the improvement (ask HMPP guys, MPC guys or CUDA guys)

• FORTRAN is not the future!

As you have maybe guessed, may advice is for new software, choose another language
A few words about language

• What should I choose:
  – C++ it is faster to develop, better for software architecture, but you need to be careful about performance
  – C, it just works and perform well... But maintainability...
  – C + C++ it rocks!
  – Scripting language (Python, lua, …) can work, be careful about maintenance and exiting library for parallelisation
  – DSL language, but be careful about performance (scilab, matlab...)
  – If you are working with an existing software, or with “old-school” people, or if you have no choice, choose FORTRAN

• => 90% of the software in our community are in FORTRAN, you have to deal with it
Conclusion

• The pessimist
  – A lot of HPC application just don't fit onto GPU => rewrite
  – Things are moving fast, take your time to do safer choice
    • Hardware design
    • Programming model
    • Programming language
    • Compiler
  – Your algorithmic transformation for current many-core will need to evolve again in the coming years
  – I will need a computer scientist in my lab :( 
  – Free lunch is over, the long and regular evolution of CPU has end, we enter a new era
Conclusion

• The optimist
  – HPC application and parallelism have a long history
  – Things are moving forward and converge:
    • Hardware design
    • Programming model
    • Programming language
    • Compiler
  – Your algorithmic transformation for current many-core will be useful for future many core
  – I will have a computer scientist in my lab :)
  – Science will benefit from huge computing power
  – As long as it is difficult to do your job, no body will take it from you ;)
• Thank you all!