Multi-core Programming: Introduction

Timo Lilja

January 22, 2009
Outline

1. Practical Arrangements

2. Multi-core processors
   - CPUs
   - GPUs
   - Open Problems

3. Topics
Meetings: A232 on Thursdays between 14-16 o’clock
Presentation: extended slide-sets are to be handed-out before the presentation
Programming topics and meeting times are decided after we review the questionnaire forms
We might not have meetings every week!
Course web-page: http://www.cs.hut.fi/u/tlilja/multicore/
Multi-core CPUs

- Combines two or more independent cores into a single chip.
- Cores do not have to be identical.
- Moore’s law still holds but increasing frequency started to be problematic ca. 2004 for the x86 architectures.
- Main problems
  - memory wall
  - ILP wall
  - power wall
History

- Originally used in DSPs. E.g., mobile phones have general purpose processor for UI and DSP for RT processing.
- IBM POWER4 was the first non-embedded dual-core processor in 2001
- HP PA-8800 in 2003
- Intel’s and AMD’s first dual-cores in 2005
- Intel and AMD are came relatively late to the multi-core market, Intel had, however, hyper-threading/SMT in 2002
- Sun UltraSparc T1 in 2005
- Lots of others: ARM MPCore, STI Cell (PlayStation 3), GPUs, Network Processing, DSPs, ...
Multi-core Advantages and Disadvantages

**Advantages**
- Cache coherency is more efficient since the signals have less distance to travel than in separate chip CPUs.
- Power consumption may be less when compared to independent chips.
- Some circuitry is shared. E.g., L2-cache.
- Improved response time for multiple CPU intensive workloads.

**Disadvantages**
- Applications perform better only if they are multi-threaded.
- Multi-core design may not use silicon area as optimally as single core CPUs.
- System bus and memory access may become bottlenecks.
Programming multi-core CPUs

- Basically nothing new here
  - Lessons learnt in independent chip SMP programming still valid
- Shared memory access - mutexes
- Conventional synchronization problems
  - Shared memory vs. message passing
- Threads
- Operating system scheduling
- Programming language support vs. library support
What to gain from multi-cores

Amdahl’s law

The speedup of a program is limited by the time needed for the sequential fraction of the program.

- For example: if a program needs 20 hours in a single core and 1 hour of computation cannot be parallelized, then the minimal execution time is 1 hour regardless of number of cores.
- Not all computation can be parallelized.
- Care must be taken when an application is “parallelized”
  - If the SW architecture was not written with concurrent execution in mind then good luck with the parallelization!
Software technologies

- Posix threads
- Separate processes
- CILK
- OpenMP
- Intel Threading Building Blocks
- Various Java/C/C++ libraries/language support
- FP languages: Erlang, Concurrent ML/Haskell
Stream processing

- Based on SIMD/MIMD paradigms
- Given a set data stream and a function kernel which is to be applied to each element in the stream
- Stream processing is not standard CPU + SIMD/MIMD
  - stream processors are massively parallel (e.g. 100s of GPU cores instead of CPUs 1-10 cores today)
  - imposes limits on kernel and stream size
- Kernel must be independent and data locally used to get performance gains from stream processing
An example: traditional for-loop

```c
for (i = 0; i < 100 * 4; i++)
    r[i] = a[i] + b[i];
```

in SIMD paradigm

```c
for (i = 0; i < 100; i++)
    vector_sum(r[i], a[i], [i]);
```

in parallel stream paradigm

```c
streamElements 100
streamElementFormat 4 numbers
elementKernel "@arg0+@arg1"
result = kernel(source0, source1)
```
GPUs

- General-purpose computing on GPUs (GPGPU)
- Origins in programmable vertex and fragment shaders
- GPUs are suitable for problems that can be solved using stream processing
- Thus, data parallelism must be high and computation independent
- arithmetic intensity $= \text{operations} / \text{words transferred}$
- Computation that benefit from GPUs have high arithmetic intensity
High arithmetic intensity requires that communication between stream elements is minimised.

**Gather**
- Kernel requests information from other parts of memory
- Corresponds to random-access *load* capability

**Scatter**
- Kernel distributes information to other elements
- Corresponds to random-access *store* capability
GPU Resources

Programmable processors
- Vertex Processors
- Fragment Processors

Memory management
- Rasterizer
- Texture Unit
- Render-to-Texture
Data types

- Basic types: integers, floats, booleans
- Floating point support somewhat limited
  - Some NVidia Tesla models support full double precision floats
- Care must be taken when using GPU floats
### CPU vs. GPU

- **mapping between CPU and GPU concepts:**

<table>
<thead>
<tr>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>textures (streams)</td>
<td>arrays</td>
</tr>
<tr>
<td>fragment programs (kernel)</td>
<td>inner loops</td>
</tr>
<tr>
<td>render-to-texture</td>
<td>feedback</td>
</tr>
<tr>
<td>geometry rasterization</td>
<td>computation invocation</td>
</tr>
<tr>
<td>texture coordinates</td>
<td>computational domain</td>
</tr>
<tr>
<td>vertex coordinates</td>
<td>computational range</td>
</tr>
</tbody>
</table>
Software technologies

- ATI/AMD Stream SDK
- NVidia Cuda
- OpenCL
- BrookGPU
- GPU kernels and Haskell? Other FPs?
- Intel Larrabee and corresponding software?
First beta in 2007

C compiler with language extensions specific to GPU stream processing

Low-level ISAs are closed, proprietary driver compiles the code to the GPU (AMD/ATI have opened their ISAs)

OS Support: Windows XP/Vista, Linux, Mac OS X

In Linux Redhat/Suses/Fedora/Ubuntu supported, though no .debs but a shell-script installer available

http://www.nvidia.com/object/cuda_get.html

PyCuda: Python interface for cuda:
http://mathema.tician.de/software/pycuda
NVidia Cuda (2/2)

- An Example:
  ```c
  // Kernel definition
  __global__ void vecAdd(float* A, float* B, float* C)
  {
  }
  int main()
  {
    // Kernel invocation
    vecAdd<<<1, N>>>(A, B, C);
  }
  ```

- Compiler is `nvcc` and file-extension `.cu`

Open Problems

- How ready are current environments for multi-core/GPU?
  - E.g., Java/JVM

- What tools are needed for developing concurrent software?
  - In multi-core CPUs and GPUs
  - E.g., debuggers for GPUs?

- Operating system support?
  - Schedulers
  - Device drivers?

- Totally proprietary, licensing issues?

- Lack of standards? Is OpenCL a solution?
Possible topics (1/2)

- Multi-core CPUs
  - Threads, OpenMP, UPC, Intel Threading Building Blocks
  - Intel’s Tera-scale Computing Research Program
- GPU
  - NVidia Cuda, AMD FireStream, Intel Larrabee, OpenCL
  - Stream processing
- Programming languages
  - FP languages: Haskell and GPUs, Concurrent ML, Erlang
  - Main stream languages: Java/JVM/C#/C/C++
  - GPU/Multi-core support in script languages (Python, Ruby, Perl)
  - Message passing vs. shared memory
Possible topics (2/2)

- Hardware overview
  - multi-core CPUs, GPUs: What is available? How many cores?
  - embedded CPUs, network hardware
  - other?

- Applications
  - What applications are (un)suitable for multi-core CPUs/GPUs?
  - Gaining performance in legacy applications: (Is it possible? How to do it? Problems? Personal experiences?)