Accelerators

Jakub Kurzak
kurzak@eecs.utk.edu
http://web.eecs.utk.edu/~kurzak/

Innovative Computing Laboratory
Electrical Engineering and Computer Science
University of Tennessee
In the twilight of Moore’s Law, the transitions to multicore processors, GPU computing, and HaaS cloud computing are not separate trends, but aspects of a single trend – mainstream computers from desktops to ‘smartphones’ are being permanently transformed into heterogeneous supercomputer clusters. Henceforth, a single compute-intensive application will need to harness different kinds of cores, in immense numbers, to get its job done.

The free lunch is over. Now welcome to the hardware jungle.
In June 2008 The Roadrunner supercomputer at Los Alamos National Laboratory crossed the performance of **1 PetaFLOPS** ($10^{15}$ floating point operations per second) using 6,480 AMD Opteron dual-core processors and 12,960 IBM PowerXCell 8i processors.
In November 2010 the Tianhe supercomputer at National Supercomputer Center in Tianjin crossed the performance of 2.5 PetaFLOPS ($10^{15}$ floating point operations per second) using 14,336 Intel Xeon 6-core processors and 7,168 Nvidia Fermi GPU accelerators.
In November 2011 the K computer at RIKEN Advanced Institute for Computational Science in Kobe crossed the performance of **10 PetaFLOPS** \((10^{15} \text{ floating point operations per second})\) using **88,128 SPARC64 VIIIfx** 8-core processors.
In June 2012 the Sequoia supercomputer at Lawrence Livermore National Laboratory crossed the performance of 16 PetaFLOPS ($10^{15}$ floating point operations per second) using 98,304 IBM BQC 16-core* processors.
In November 2012 the Titan supercomputer at Oak Ridge National Laboratory crossed the performance of 17 **PetaFLOPS** ($10^{15}$ floating point operations per second) using **18,688 AMD Opteron** 16-core processors and **18,688 Nvidia Kepler** GPU accelerators.
In June 2013 the Tianhe-2 supercomputer at National Supercomputing Center in Guangzhou crossed the performance of 33 PetaFLOPS ($10^{15}$ floating point operations per second) using 32,000 Intel Ivy Bridge 12-core processors and 48,000 Intel Xeon Phi accelerators.
What is wrong with CPUs?

performance walls

- **Power Wall – Power Dissipation**
  - The chip will melt if running any faster (higher clock rate)

- **Frequency Wall – Pipeline Depth**
  - To crank up the clock shorter pipeline stages are required
  - To have shorter pipeline stages, more stages are required
  - When code branches, pipelines are flushed
    (there is not enough Instruction Level Parallelism in serial code)

- **Memory Wall – DRAM Latency**
  - DRAM can provide plenty of bandwidth, but very high latency
  - If data does not reside in cache, it can cost 1000 cycles to access it
  - Prefetching reached the point of diminishing returns
What is an accelerator?
British for gas pedal
What is an accelerator?

what is the tradeoff

- A device that runs very fast
  - What if it is very expensive?
  - What if it is very power hungry?

- A device that is very cost/power efficient?
  - What if it is very slow (the ARM in your cellphone)?
  - To have shorter pipeline stages, more stages are required

- A device that is fast and cost/power efficient?
  - Can it run the OS?
  - Can it run serial code?
  - Can it run legacy software (numerical libraries, e.g. LAPACK)?
Cell Broadbane Engine
Sony / Toshiba / IBM

Mercury PCI accelerator cards

IBM blade servers

Sony Playstation 3

Embedded systems

UTK PS3 Cluster
Cell Broadbaine Engine
architecture overview

- **PPE** – Power Processing Element
- **SPE** – Synergistic Processing Element
  - **SPU** – Synergistic Processing Unit
  - **LS** – Local Store
  - **MFC** – Memory Flow Controller
- **EIB** – Element Interconnect Bus
Cell Broadbane Engine
architecture overview

- 4-way SIMD vector architecture
- 128 vector registers (128B)
- 256 KB of scratchpad memory (local store)
- dedicated DMA engine for data transfers to main memory and other local stores
Cell Broadbane Engine
programming challenges

- SIMD vectorization – exploiting data parallelism
  - completely SIMD architecture
  - no scalar registers
  - no scalar instructions
  - no SIMD vectorization = 1% performance

- Parallelization – exploiting thread-level parallelism
  - octa-core architecture
  - no parallelization = 12.5% performance

- Explicit communication
  - no coherent caches
  - core-private scratchpad memories
  - out-of-core / out-of-memory programming
Cell Broadbane Engine
demise

Cell Technology Roadmap:
Compatible code and security base across entire line

Performance Enhancements/Scaling

Cost Reduction

Low Power

2006 2007 2008 2009 2010 2011

Cell/B.E. (1+8) 90nm SOI 3.2 GHz
Cell/B.E. (1+8) 65nm SOI 3.2 GHz
Cell/B.E. (1+8) 45nm SOI 3.2 GHz
IBM PowerXCell™ 8i (1+8 eDP SPE) 65nm SOI, 3.2 GHz
IBM PowerXCell 32v (4PPE<32 eSPE) ~3.8 GHz 1 TFlop (est.)
Toshiba SPURiEngine (0=4 SPE) 65nm 1.5 GHz

All future dates and specifications are estimations only and subject to change without notice. Dashed outlines indicate concept designs.

© 2009 IBM Corporation
Intel Xeon Phi
a.k.a. Intel MIC

**PCI accelerator card**

**multi-accelerator server blade**

**multi-accelerator server blade**
Intel Xeon Phi architecture

- ca. 60 cores
- x86 architecture
- 4 hardware threads each
- shot in-order pipelines
- **512-bit SIMD**
- 16 single precision instruction/cycle
- 8 double precision instructions/cycle
- support for FMA
- **coherent caches**
- ring interconnect
GPU systems

- High-end gaming
- Computing
- Low-end gaming
- Embedded
Main Sources
CUDA documentation

CUDA Programming Guide

CUDA C PROGRAMMING GUIDE
PG-0803-001, v5.5 | July 2011
Design Guide

docs.nvidia.com/cuda/

CUDA Documents

Release Notes
The Release Notes for the CUDA Toolkit from v4.0 to today.

CUDA
The End User License Agreements for the NVIDIA CUDA Toolkit, the NVIDIA CUDA Samples, the NVIDIA Display Driver, and NVIDIA FlyKit (Visual Studio Edition).

Getting Started Guides

Getting Started Linux
This guide discusses how to install and check for correct operation of the CUDA Development Tools on GNU/Linux systems.

Getting Started Mac OS X
This guide discusses how to install and check for correct operation of the CUDA Development Tools on Mac OS X systems.

Getting Started Windows
This guide discusses how to install and check for correct operation of the CUDA Development Tools on Microsoft Windows systems.

Programming Guides

Programming Guide
This guide provides a detailed discussion of the CUDA programming model and programming interface. It then describes the hardware implementation, and provides guidance on how to achieve maximum performance. The appendix include a list of all CUDA-enabled devices, detailed description of all extensions to the C-language, listings of supported math kernel functions, C++ features supported in host and device code - details on memory fencing, technical specifications of various kernels, and concludes by introducing the low-level driver API.

Best Practices Guide
This guide presents established parallelization and optimization techniques and explains coding mechanisms and idioms that can greatly simplify programming for CUDA capable GPU architectures. The intent is to provide guidelines for obtaining the best performance from NVIDIA GPUs using the CUDA Toolkit.

Kepler Compatibility Guide
This application note is intended to help developers ensure that their NVIDIA CUDA applications will run effectively on GPUs based on the NVIDIA Kepler Architecture. This document provides guidance to ensure that your software applications are compatible with Kepler.

Kepler Tuning Guide
Kepler is NVIDIA's next-generation architecture for CUDA compute applications. Applications that follow the best practices for the Fermi architecture should typically see speedups on the Kepler architecture without any code changes. This guide summarizes the ways that an application can be fine-tuned to gain additional speedups by leveraging Kepler architectural features.
GPUs vs CPUs

computing power
GPUs vs CPUs
memory bandwidth
GPUs vs CPUs

architecture

CPU

GPU
Nvidia Fermi architecture

- 16 multiprocessors
- 32 cores each
- 512 total cores
Nvidia Kepler architecture

- 15 multiprocessors
- 192 cores each
- 2,880 total cores
Nvidia Kepler

gaming Kepler vs HPC Kepler

Adding double precision units
SM / SMX / SMM
Fermi / Kepler / Maxwell

32 cores

192 cores

128 cores
## GPU Computing

**Nvidia software stack**

### GPU Computing Applications

<table>
<thead>
<tr>
<th>Libraries and Middleware</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUFFT</td>
<td>CUDA MAGMA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming Languages</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C++</td>
</tr>
</tbody>
</table>

### CUDA-Enabled NVIDIA GPUs

<table>
<thead>
<tr>
<th>Architecture</th>
<th>GeForce 600 Series</th>
<th>Quadro Kepler Series</th>
<th>Tesla K20 Tesla K10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kepler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(compute capabilities 3.x)</td>
<td>GeForce 500 Series</td>
<td>Quadro Fermi Series</td>
<td>Tesla 20 Series</td>
</tr>
<tr>
<td>Fermi</td>
<td>GeForce 400 Series</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(compute capabilities 2.x)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tesla</td>
<td>GeForce 200 Series</td>
<td>Quadro FX Series</td>
<td>Tesla 10 Series</td>
</tr>
<tr>
<td>Architecture</td>
<td>GeForce 9 Series</td>
<td>Quadro Plex Series</td>
<td></td>
</tr>
<tr>
<td>(compute capabilities 1.x)</td>
<td>GeForce 8 Series</td>
<td>Quadro NVS Series</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Entertainment</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Professional Graphics</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>High Performance Computing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Think of image processing.
Every block is a tile.
Every thread is a pixel.

Say you want to dim the image.
Every thread computes new brightness for one pixel.
CUDA Example

add vectors

Add two vectors A and B of size N, and store the result in C.

```c
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Every thread gets a unique thread ID accessible within the kernel through the built-in variable `threadIdx`.
Add two matrices A and B of size NxN, and store the result in C.

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                        float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Every thread gets unique coordinates `threadIdx.x` and `threadIdx.y`
CUDA Example

add matrices

the same with multiple blocks

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{

    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

block coordinates (blockIdx.x, blockIdx.y) and thread coordinates (threadIdx.x, threadIdx.y) available within the kernel
CUDA
heterogeneous programming

C Program
Sequential
Execution

Serial code

Parallel kernel
Kernel0<<<>>>() (1)

Device

Grid 0
Block (0, 0) Block (1, 0) Block (2, 0)
Block (0, 1) Block (1, 1) Block (2, 1)

Serial code

Host

send data to the GPU

receive results from the GPU

Device

Grid 0
Block (0, 0) Block (1, 0)
Block (0, 1) Block (1, 1)
Block (0, 2) Block (1, 2)

GPU memory (GDDR5)

PCl bus

CPU memory (DDG3)

Parallel kernel
Kernel1<<<>>>() (1)

Serial code

Host

send data to the GPU

receive results from the GPU
Optimization

communication overlapping

Hide communication between the host and the device.
Overlap DMA data transfers with kernel launches.
Optimization
communication overlapping

Basic Loop

for $i = 0$ to $n$

- Load $i$
- Compute $i$
- Store $i$

Pipeline 0 (even) Pipeline 1 (odd)

Software-Pipelined Loop

- Load 0
- Compute 0
- Load 1
- Compute 0
- Load $i + 1$
- Store $i - 1$
- Compute $n$
- Store $n - 1$
- Store $n$

Pipeline 0 (even) Pipeline 1 (odd)

- pipelining
- double buffering
- multi-buffering
Unless...

system on a chip

ADM Fusion
Accelerated Processing Unit (APU)
GPU + x86

Nvidia Tegra
GPU + ARM
CUDA
memory hierarchy
Barrier is the main mechanism for synchronizing threads
  • registers are thread-private
  • local memory is thread-private
  • shared memory exchanges require barriers

Barrier is the main mechanism for synchronizing blocks
  • shared memory is private to thread block

There are other ways of synchronization, such as atomic memory operations.
Performance Optimization

Minimize Thread Divergence
- all threads to the same

Avoid Warp Serialization
- all threads access different shared memory banks

Optimize Global Memory Access
- access is sequential and aligned

Maximize Occupancy
- there is a massive number of threads

Hide Host to Device Communication
- overlap communication with kernel execution
Optimization

thread divergence

All threads in a warp have to follow the same execution path.

I.e., all thread in a warp have to branch in tandem.

If threads take different execution paths the execution is serialized, i.e., different path are executed in sequence.

This is like the Japanese 31-legged race. If one kid is out of step, everyone falls on the face.
The best access to DRAM is sequential and aligned.

Sequential means that consecutive threads read consecutive memory locations.

Aligned means that the first address is 128B-aligned (divisible by 128).

Caches were introduced with compute capability 2.x and 3.x.

Caches relieve the penalty for no-sequential access.

There is still penalty for mis-aligned access.
DRAM Access
aligned and sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compute capability:

<table>
<thead>
<tr>
<th>1.0 and 1.1</th>
<th>1.2 and 1.3</th>
<th>2.x and 3.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Uncached</td>
</tr>
<tr>
<td>1x 64B at 128</td>
<td>1x 64B at 128</td>
<td>1x 32B at 128</td>
</tr>
<tr>
<td>1x 64B at 192</td>
<td>1x 64B at 192</td>
<td>1x 32B at 160</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1x 32B at 192</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1x 32B at 224</td>
</tr>
</tbody>
</table>
DRAM Access
aligned and non-sequential

### Aligned and non-sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
</tr>
<tr>
<td>128</td>
</tr>
<tr>
<td>160</td>
</tr>
<tr>
<td>192</td>
</tr>
<tr>
<td>224</td>
</tr>
<tr>
<td>256</td>
</tr>
<tr>
<td>288</td>
</tr>
</tbody>
</table>

### Threads:

- 0
- ... 31

### Compute capability:

- **1.0 and 1.1**
- **1.2 and 1.3**
- **2.x and 3.x**

### Memory transactions:

<table>
<thead>
<tr>
<th>Uncached</th>
<th>Uncached</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 x 32B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
</tr>
<tr>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td>1 x 32B at 160</td>
</tr>
<tr>
<td>8 x 32B at 192</td>
<td>1 x 64B at 192</td>
<td>1 x 32B at 192</td>
</tr>
<tr>
<td>8 x 32B at 224</td>
<td>1 x 32B at 224</td>
<td>1 x 32B at 224</td>
</tr>
</tbody>
</table>
DRAM Access
mis-aligned and sequential

<table>
<thead>
<tr>
<th>Compute capability:</th>
<th>1.0 and 1.1</th>
<th>1.2 and 1.3</th>
<th>2.x and 3.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Uncached</td>
<td>Cached</td>
</tr>
<tr>
<td>7x 32B at 128</td>
<td>1x 128B at 128</td>
<td>1x 32B at 128</td>
<td>1x 128B at 128</td>
</tr>
<tr>
<td>8x 32B at 160</td>
<td>1x 64B at 192</td>
<td>1x 32B at 160</td>
<td>1x 32B at 160</td>
</tr>
<tr>
<td>8x 32B at 192</td>
<td>1x 32B at 256</td>
<td>1x 32B at 192</td>
<td>1x 32B at 256</td>
</tr>
<tr>
<td>8x 32B at 224</td>
<td>1x 32B at 224</td>
<td>1x 32B at 224</td>
<td>1x 32B at 256</td>
</tr>
<tr>
<td>1x 32B at 256</td>
<td>1x 32B at 256</td>
<td>1x 32B at 256</td>
<td>1x 128B at 256</td>
</tr>
</tbody>
</table>
Optimization
shared memory access

Explaining the simpler model of compute capability 2.x.
Shared memory is organized in banks.
There are 32 banks mapped to consecutive memory locations.
The best case is when each thread in a warp reads from a different bank.
If different threads read from the same bank, bank conflicts happen.
(One bank can only serve one request at a time.)
In that case the read instruction has to be replayed.
This is called “warp serialization”.
The exception is when many threads read the same address.
In this case broadcast happens.
Explaining bank conflicts for compute capability 2.x (easier to explain than 3.x)

- stride one – no conflicts
- stride two – two-way conflicts
- stride three – no conflicts

For compute capability 3.x there are no conflicts for stride two.
Explaining bank conflicts for compute capability 2.x (easier to explain than 3.x)

- random permutations – no conflicts
- broadcast & permutation – no conflicts
- broadcast – no conflicts
The more registers and shared memory each block needs the less blocks there will be. less blocks = less threads = low occupancy = low performance
Optimization

CUDA C Best Practices Guide

- Assess
- Parallelize
- Optimize
- Trace
- Profile
- Disassemble

further reading