PERFORMANCE ANALYSIS AND TUNING FOR GPUs
MICRO ’10 TUTORIAL
Hyesoon Kim, Richard Vuduc
Outline

- Part #1.0 Basic CUDA/GPU (Kim)
- Part #1.5 Performance Principles (Vuduc)
- Break
- Part #2.0 Analysis Tuning (Vuduc)
  - SC’10 Gordon Bell Prize winner’s example
- Part #2.5 Metrics and Analysis Tuning (Kim)
- Discussions (Vuduc & Kim)
Part #1.0 Basic CUDA/GPU
Outline of Part #1.0

- CUDA Programming 101
- GPGPU Architecture basic
- GPGPU Performance
- GPGPU Performance Analysis Tools
GPU Architecture Trend

- Fixed pipelines → programmable cores → unified programmable cores → more cores → GPGPU support

[the slide is from Hong&Kim ISCA’10]
GPGPU Programming

- Become popular with CUDA (Compute Unified Device Architecture)
- CUDA (Based on Nvidia architectures)
  - Portland Group Compiler supports CUDA → x86
- OpenCL
Quick Summary of CUDA Programming Model

- SIMD or SIMT
  - Single instruction multiple data or single instruction multiple thread
- Unified Memory space (global memory space)
- Program hierarchy
  - Thread, block, kernel
Thread & Block

Sing instruction multiple thread
Block = a group of thread which share "the shared memory space"
Warp

[the slide is from Hong&Kim ISCA'09]
Thread, block, and kernel have different memory spaces

<table>
<thead>
<tr>
<th>Space</th>
<th>~= CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Memory</td>
<td>Within Threads Stack</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Within Blocks Distributed memory space</td>
</tr>
<tr>
<td>Global Memory</td>
<td>All Centralized storage</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>All Centralized read-only storage (very small)</td>
</tr>
<tr>
<td>Texture Memory</td>
<td>All Centralized read-only storage (medium size, 2D-cache)</td>
</tr>
</tbody>
</table>
Memory Data Indexing

- SIMT-execution model
- Use thread id and block id to index data

Let’s assume $N=16$, $blockDim=4 \rightarrow 4$ blocks

- $blockIdx.x = 0$
  - $blockDim.x = 4$
  - $threadIdx.x = 0,1,2,3$
  - $ldx = 0,1,2,3$

- $blockIdx.x = 1$
  - $blockDim.x = 4$
  - $threadIdx.x = 0,1,2,3$
  - $ldx = 4,5,6,7$

- $blockIdx.x = 2$
  - $blockDim.x = 4$
  - $threadIdx.x = 0,1,2,3$
  - $ldx = 8,9,10,11$

- $blockIdx.x = 3$
  - $blockDim.x = 4$
  - $threadIdx.x = 0,1,2,3$
  - $ldx = 12,13,14,15$
1D, 2D, 3D data structures

- A kernel is executed as a grid of thread blocks
- Threads and blocks have IDs:
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Loop index in sequential loop:
  - Use thread ids, block ids
  - 1D array index = \( c_1 \cdot \text{threadId}.x + c_2 \cdot \text{blockId}.x \)
  - 2D array index = \( c_1 \cdot \text{threadId}.x + c_2 \cdot \text{blockId}.x + c_3 \cdot \text{threadId}.y + c_4 \cdot \text{blockId}.y \)

CPU code

```c
for (ii = 0; ii < 100; ++ii) {
}
```

CUDA code

```c
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```
Synchronization Model

- Bulk-Synchronous Parallel (BSP) program (Valiant [90])
- Synchronization within blocks using explicit barrier
- Implicit barrier across kernels
  - Kernel 1 → Kernel 2
  - C.f.) Cuda 3.x
MIMD with CUDA

- Use thread id to write serial programs or reduce the number of running threads

If (threadId.x%==2)
If (threadld.x%==4)
If (threadld.x%==8)

(reduction example)

- Use block id to generate MIMD effects
  - If (blockld.x == 1) do work 1
  - If (blockid.x == 2) do work 2
Global Communications

- Use multiple kernels
- Write to same memory addresses
  - Behavior is not guaranteed
  - Data race
- Atomic operation
  - No other threads can write to the same location
  - Memory Write order is still arbitrary
  - Keep being updated: atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}
- Performance degradation
  - Fermi increases atomic performance by 5x to 20x (M. Shebanow)
New Programming Features in Fermi

- Supporting pointers
  - Limited stacks
- Recursive programming
- Concurrent Kernel executions from the same application
  - Efficient pipelining parallel program paradigm
- More...
## OpenCL vs. CUDA

<table>
<thead>
<tr>
<th></th>
<th>OpenCL</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Model</strong></td>
<td>Work-groups/work-items</td>
<td>Block/Thread</td>
</tr>
<tr>
<td><strong>Memory model</strong></td>
<td>Global/constant/local/private</td>
<td>Global/constant/shared/local + Texture</td>
</tr>
<tr>
<td><strong>Memory consistency</strong></td>
<td>Weak consistency</td>
<td>Weak consistency</td>
</tr>
<tr>
<td><strong>Synchronization</strong></td>
<td>Synchronization using a work-group barrier (between work-items)</td>
<td>Using synch_threads Between threads</td>
</tr>
</tbody>
</table>
Understanding GPGPU Architectures

Caution!

- The material is not an NVIDIA official documentation
- Much information comes from literatures such as Kirk&Hwu lecture notes
Overview of GPU (Tesla) Architecture

- Streaming Multiprocessor
- Streaming Multiprocessor
- Streaming Multiprocessor

- Interconnection Network

- Global Memory (Device memory)

- PC
- I-Cache
- Decoder
- Shared Memory
- Stream Processor
- Stream Processor
- Stream Processor
- Stream Processor

- Caches
Execution Unit: Warp

- **Warp is the basic unit of execution**
  - A group of threads (e.g. 32 threads for the Tesla GPU architecture)

**Warp Execution**

```
<table>
<thead>
<tr>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Inst 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Sources ready

```
T T T T
T T T T
T T T T
```

One warp

```
Inst 1
Inst 2
Inst 3
```

Finite number of streaming processors

```
S S S S
P P P P
```

SIMD Execution Unit

```
T T T T
```

One warp

```
T T T T
```

One warp

```
T T T T
```

Finite number of streaming processors
Pipeline

- **Fetch**
  - One instruction for each warp (could be further optimizations)
  - Round Robin, Greedy-fetch (switch when stall events such as branch, I-cache misses, buffer full)

- **Thread scheduling polices**
  - Execute when all sources are ready
  - In-order execution within warps
  - Scheduling polices: Greedy-execution, round-robin
Handling Branch Instructions

- Recall the reduction example
  
  If (threadId.x%==2) 
  If (threadId.x%==4) 
  If (threadId.x%==8) 

- What about other threads?

- What about different paths?

From Fung et al. MICRO ‘07
Divergent Branches

- All branch conditions are serialized and will be executed
  - Parallel code $\rightarrow$ sequential code
- Divergence occurs within a warp granularity.
- It’s a performance issue
  - Degree of nested branches
- Depending on memory instructions, (cache hits or misses), divergent warps can occur
  - Dynamic warp subdivision [Meng’10]
- Hardware solutions to reduce divergent branches
  - Dynamic warp formation [Fung’07]
**Divergent Branches Execution Time**

- Divergent branches serialize execution of warps

![Diagram showing a graph with nodes and branches, and a code snippet. The graph illustrates the execution flow and the code snippet includes a conditional statement for modulo operations.]
Many levels of queues
- Large size of queues
- High number of DRAM banks
- Sensitive to memory scheduling algorithms
  - FRFCFS >> FCFS
- Interconnection network algorithm to get FRFCFS Effects
  - Yan’09,
Even coalesced memory accesses generate multiple transactions. $4 \times 32 \times 32 = 128$ B req size

- More processing cycles for the uncoalesced case
Multiple In-flight Memory Requests

- In-order execution but
- Warp cannot execute an instruction when sources are dependent on memory instructions, not when it generates memory requests
- High MLP
Same Data from Multiple Threads (SDMT)

- **High Merging Effects**
  - Inter-core merging
  - Intra-core merging

- **Techniques to take advantages of this SDMT**
  - Compiler optimization [Yang’10]: increase memory reuse
  - Cache coherence [Tarjan’10]
  - Cache increase reuses
Fermi Architecture
Outline of Part #1.0

- CUDA Programming 101
- GPGPU Architecture basic
- GPGPU Performance
- GPGPU Performance Analysis Tools
Performance Optimization Opportunities

- Communication cost between CPU and GPU
- Occupancy
- Uncoalesced global memory access
- Divergent branches
- Bank conflicts (shared memory/global memory)
- Integer operations (not so critical in Fermi)
- Memory allocation overhead
  - Allocate once, lazy copy etc.
CPU-GPU Communications

- High overhead between CPUs and GPUs

Matrix 1000 * 1000

Matrix 8000 * 8000

From Luk[09]
Occupy

- Shows how many blocks are assigned to the SM
- Programmer specifies the number of threads per block

![Diagram](image)

- Register requirements per block
- Shared memory requirements per block

100% Occupancy

Only one block is allocated
Higher Occupancy

- Better processor utilization
- Hide the memory latency

![Diagram showing processor occupancy and memory latency]

Warp 1: Processor is not utilized
Warp 2: Processor is not utilized
Warp 3: Processor is not utilized
Warp 4: Better utilization!
Warp 5: Better utilization!
Round-robin fetch/execution model reduces synchronization overhead

Uniform thread reduce synchronization overhead

Ideally, one synchronization overhead is less than one memory latency
Memory Bank Conflicts

- **Shared memory bank conflicts**
  - Shared memory accesses within a warp are serialized
  - 4 cycles $\rightarrow$ $2 \times 32 = 64$ cycles

- **DRAM memory bank conflicts**
  - DRAM memory accesses are serialized
  - 100 cycles $\rightarrow$ 100 cycles $\times$ # of serialization
  - Higher number of banks, statically, there will be some amount of bank conflicts
Other Performance Impacts

- Warp Serialization
  - Shared memory bank conflicts
  - Constant memory bank conflicts

- TLB Miss effects
  - Based on “Micro-benchmarking the GT200 GPU”, Papadopoulou et al.
Outline of Part #1.0

- CUDA Programming 101
- GPGPU Architecture basic
- GPGPU Performance
- GPGPU Performance Analysis Tools
Tools

- CUDA Profiler [NVIDIA]
  - coalesced/uncoalesced, global memory throughputs, warp serializations, throughput oriented information

- Ocelot: Dynamic execution environment [GT, Yalamanchili]
Hardware Performance Counters

- Divergent Branches
- Warp Serialize
  - Bank conflicts
- Memory transfer size
- Global memory requests
- TLB Misses
- CTA launched
Existing Tools

- Ocelot: Dynamic execution environment [GT]

- Ocelot
  - Ocelot: PTX Emulator
    - Parallel Thread Execution (PTX)
    - NVIDIA's virtual ISA
    - PTX 1.4 compliant Emulation
    - PTX 2.1 support in progress
    - Validated on full CUDA SDK
    - Open Source version released
    - Status: Available as open source at http://code.google.com/p/gpuocelot/

- Dynamic Translation (LLVM)
  - Bridge to UIUC's LLVM
  - Dynamic Compilation Infrastructure
    - Support for multiple ISAs
    - Dynamic compilation to x86 commodity CPUs
    - Status: Available
    - Available as open-source at http://code.google.com/p/gpuocelot/

- Online selection of Native Hardware Execution
  - Status: Available

- Workload Characterization Tools
  - Status: Internal use
  - Distribution (TBD)

Prof. Yalamanchili (GT, ECE)
CUDA Compilation Process

- The CUDA runtime library (`cudart`)
- The CUDA core library (`cuda`)

**Kernel** → **Libraries**
- PTX
  - No register allocations
  - Virtual ISA

**Device driver**
- Run-time compiler

**Dynamic binaries**

- Ocelot replaces libraries with emulator
- Run PTX directly on CPUs
Ocelot: Dynamic Execution Infrastructure

**PTX Kernel**

```
L_BB_1:
  add.s64 %rd2, %rd1, 1
  mul.s64 %rd3, %rd2, 4
  mov.s64 %rd4, 256
  setp.lt.s64 %p1, %rd3, %rd4
  @%p1 bra L_BB_3

L_BB_2:
  abs.f64 %fd1, %fd1
  mov.s64 %rd5, 64
  setp.lt.s64 %p2, %rd3, %rd5
  @%p2 bra L_BB_4

L_BB_3:
  sin.f64 %rd2, %fd1
  sli.f64 %fd2, (%rd9 + 4)

L_BB_4:
  reconverge
  reconverge
  exit
```

**Productivity Tools**

- PTX Emulator
- Dynamic Translation (LLVM)

**Multiplatform Support**

- Emulator Device
- NVIDIA GPU Device
- Multicore CPU Device

**Multi-GPU Support**

- NVIDIA GPUs

**Performance Analysis and Modeling**

- PTX Kernel IR (CFG)
- PTX Metadata (DFG, Dom)

**PTX Transformer/Code Generator**

- PTX Transformations

**Prof. Yalamanchili**

(GT, ECE)
Data Analytics for Workload Characterization

**Gathering Statistics**
- Identify uncorrelated variables
- Discover correlated behaviors

**Principal Components Analysis**
- Find applications with common characteristics

**Clustering**
- Predict execution time, energy, or power with static or dynamic measurements

**Regression Modeling**

- Software Emulation (trace analysis)
- Instrumented Native implementation (run-time)
- Dynamically translated PTX-to-multicore x86 implementation (via LLVM bridge)

Prof. Yalamanchili (GT, ECE)