EE382N: Principles in Computer Architecture
Parallelism and Locality
Fall 2009

Lecture 16 - GPU Architecture of NVIDIA GeForce 8&9 + CUDA

Mattan Erez

UT ECE

The University of Texas at Austin
Make the Compute Core The Focus of the Architecture

- Processors execute computing threads
- Alternative operating mode specifically for computing

Manages thread blocks
Only one kernel at a time
GeForce-8 Series HW Overview

Streaming Processor Array

Texture Processor Cluster

Streaming Multiprocessor

Instruction Fetch/Dispatch

Shared Memory

Instruction L1

Data L1

TEX

SM

SP

SFU
CUDA Processor Terminology

- **SPA** – Streaming Processor Array
  - Array of TPCs
    - 8 TPCs in GeForce8800

- **TPC** – Texture Processor Cluster
  - Cluster of 2 SMs + 1 TEX
    - TEX is a texture processing unit

- **SM** – Streaming Multiprocessor
  - Array of 8 SPs
  - Multi-threaded processor core
  - Fundamental processing unit for a thread block

- **SP** – Streaming Processor, now CUDA PE
  - Scalar ALU for a single thread
    - With 1K of registers
Thread Life Cycle in HW

- Kernel is launched on the SPA
  - Kernels known as grids of thread blocks

- Thread Blocks are serially distributed to all the SM's
  - Potentially >1 Thread Block per SM
  - At least 96 threads per block

- Each SM launches Warps of Threads
  - 2 levels of parallelism

- SM schedules and executes Warps that are ready to run

- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks
Load/Store (Memory read/write)
Clustering/Batching

• Use LD to hide LD latency (non-dependent LD ops only)
  – Use same thread to help hide own latency
• Instead of:
  – LD 0 (long latency)
  – Dependent MATH 0
  – LD 1 (long latency)
  – Dependent MATH 1
• Do:
  – LD 0 (long latency)
  – LD 1 (long latency - hidden)
  – MATH 0
  – MATH 1
• Compiler handles this!
  – But, you must have enough non-dependent LDs and Math
Bandwidths of GeForce 9800 GTX

- Frequency
  - 600 MHz with ALUs running at 1.2 GHz

- ALU bandwidth (GFLOPs)
  - \((1.2 \text{ GHz}) \times (16 \text{ SM}) \times ((8 \text{ SP}) \times (2 \text{ MADD}) + (2 \text{ SFU})) = \sim 400 \text{ GFLOPs}\)

- Register BW
  - \((1.2 \text{ GHz}) \times (16 \text{ SM}) \times (8 \text{ SP}) \times (4 \text{ words}) = 2.5 \text{ TB/s}\)

- Shared Memory BW
  - \((600 \text{ MHz}) \times (16 \text{ SM}) \times (16 \text{ Banks}) \times (1 \text{ word}) = 600 \text{ GB/s}\)

- Device memory BW
  - 2 GHz GDDR3 with 256 bit bus: 64 GB/s

- Host memory BW
  - PCI-express: 1.5 GB/s or 3 GB/s with page locking
Communication

• How do threads communicate?

• Remember the execution model:
  – Data parallel streams that represent independent vertices, triangles, fragments, and pixels in the graphics world
  – These never communicate

• Some communication allowed in compute mode:
  – Shared memory for threads in a thread block
    • No special communication within warp or using registers
  – No communication between thread blocks
  – Kernels communicate through global device memory

• **Mechanisms designed to ensure portability**
Synchronization

• Do threads need to synchronize?
  – Basically no communication allowed

• Threads in a block share memory – need sync
  – Warps scheduled OoO, can’t rely on warp order
  – Barrier command for all threads in a block
  – __synchthreads()

• Blocks cannot synchronize
  – Implicit synchronization at end of kernel
  – Use atomics to form your own primitives
Control

• Each SM has its own warp scheduler
• Schedules warps OoO based on hazards and resources
• Warps can be issued in any order within and across blocks
• Within a warp, all threads always have the same position
  – Current implementation has warps of 32 threads
  – Can change with no notice from NVIDIA
Conditionals within a Thread

• What happens if there is a conditional statement within a thread?

• No problem if all threads in a warp follow same path

• **Divergence**: threads in a warp follow different paths
  - HW will ensure correct behavior by (partially) serializing execution
  - Compiler can add predication to eliminate divergence

• Try to avoid divergence
  - If (TID > 2) {...} → If(TID / warp_size > 2) {...}
Control Flow

• Recap:
  – 32 threads in a warp are executed in SIMD (share one instruction sequencer)
  – Threads within a warp can be disabled (masked)
    • For example, handling bank conflicts
  – Threads contain arbitrary code including conditional branches

• How do we handle different conditions in different threads?
  – No problem if the threads are in different warps
  – Control **divergence**
  – **Predication**
Control Flow Divergence

if (TID % 2 == 0) {
    f2();
    if (TID % 4 == 0) {
        f4();
    } else {
        f2'();
    }
} else {
    f1();
    if (TID % 3 == 0) {
        f3();
    } else {
        f1'();
    }
}
Mask Stack Enables Divergence

```java
1: if (TID % 2 == 0) {
2:   f2();
3:   if (TID % 4 == 0) {
4:     f4();
5:   }
6:   else {
7:     f2'();
8:   }
9: }
10: else {
11:   f(1);
12:   if (TID % 3 == 0) {
13:     f3();
14:   }
15:   else {
16:     f1'();
17:   }
18: }
```
Mask Stack Enables Divergence

```plaintext
1: if (TID % 2 == 0) {
2:   f2();
3:   if (TID % 4 == 0) {
4:     f4();
5:   }
6:   else {
7:     f2'();
8:   }
9: }
10: else {
11:   f(1);
12:   if (TID % 3 == 0) {
13:     f3();
14:   }
15:   else {
16:     f1'();
17:   }
18: }
```
### Mask Stack Enables Divergence

<table>
<thead>
<tr>
<th>IP</th>
<th>enable mask</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: if (TID % 2 == 0) {</td>
<td>1 1 1 1 1 1 1 1</td>
<td>9 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>2: f2();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: if (TID % 4 == 0) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: f4();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: else {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: f2’();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8: }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9: }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: else {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11: f(1);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12: if (TID % 3 == 0) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13: f3();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14: }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15: else {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16: f1’();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17: }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18: }</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mask Stack Enables Divergence

```bash
1: if (TID % 2 == 0) {
2:   f2();
3:   if (TID % 4 == 0) {
4:     f4();
5:   }
6:   else {
7:     f2'();
8:   }
9: }
10: else {
11:   f(1);
12:   if (TID % 3 == 0) {
13:     f3();
14:   }
15:   else {
16:     f1'();
17:   }
18: }
```
Mask Stack Enables Divergence

IP

1: if (TID % 2 == 0) {
2:     f2();
3:     if (TID % 4 == 0) {
4:         f4();
5:     }
6:     else {
7:         f2'();
8:     }
9: }
10: else {
11:     f(1);
12:     if (TID % 3 == 0) {
13:         f3();
14:     }
15:     else {
16:         f1'();
17:     }
18: }

enable mask

stack

9 1 1 1 1 1 1 1
Mask Stack Enables Divergence

```c
1: if (TID % 2 == 0) {
2:     f2();
3:     if (TID % 4 == 0) {
4:         f4();
5:     }
6:     else {
7:         f2'();
8:     }
9: }
10: else {
11:     f(1);
12:     if (TID % 3 == 0) {
13:         f3();
14:     }
15:     else {
16:         f1'();
17:     }
18: }
```
Mask Stack Enables Divergence

1: if (TID % 2 == 0) {
2:   f2();
3:   if (TID % 4 == 0) {
4:     f4();
5:   }
6:   else {
7:     f2'();
8:   }
9: }
10: else {
11:   f(1);
12:   if (TID % 3 == 0) {
13:     f3();
14:   }
15:   else {
16:     f1'();
17:   }
18: }

IP enable mask stack

1 0 0 0 1 0 0 0
5 1 0 1 0 1 0 1 0
9 1 1 1 1 1 1 1 1
Mask Stack Enables Divergence

```
IP                      enable mask
1: if (TID % 2 == 0) {
2:   f2();
3:   if (TID % 4 == 0) {
4:     f4();
5:   }
6:   else {
7:     f2'();
8:   }
9: }
10: else {
11:   f(1);
12:   if (TID % 3 == 0) {
13:     f3();
14:   }
15:   else {
16:     f1'();
17:   }
18: }
```

```
stack
5 1 0 1 0 1 0 1 0
9 1 1 1 1 1 1 1 1
```

1 0 0 0 1 0 0 0
Mask Stack Enables Divergence

```java
import java.util.Stack;

public class Divergence {
    public static void main(String[] args) {
        int TID = 0;
        Stack<Integer> maskStack = new Stack<>();

        if (TID % 2 == 0) {
            f2();
            if (TID % 4 == 0) {
                f4();
            } else {
                f2();
            }
        } else {
            f1();
            if (TID % 3 == 0) {
                f3();
            } else {
                f1();
            }
        }
    }

    public static void f2() {
        int val = 2;
        maskStack.push(val);
    }

    public static void f4() {
        int val = 4;
        maskStack.push(val);
    }

    public static void f1() {
        int val = 1;
        maskStack.push(val);
    }

    public static void f3() {
        int val = 3;
        maskStack.push(val);
    }
}
```
Mask Stack Enables Divergence

IP

<table>
<thead>
<tr>
<th>enable mask</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 1 0 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

1: if (TID % 2 == 0) {
2:   f2();
3: if (TID % 4 == 0) {
4:     f4();
5: }
6: else {
7:     f2’();
8: }
9: }
10: else {
11:   f(1);
12: if (TID % 3 == 0) {
13:     f3();
14: }
15: else {
16:     f1’();
17: }
18: }
Mask Stack Enables Divergence

```java
1: if (TID % 2 == 0) {
2:     f2();
3:     if (TID % 4 == 0) {
4:         f4();
5:     }
6:     else {
7:         f2'();
8:     }
9: }
10: else {
11:     f(1);
12:     if (TID % 3 == 0) {
13:         f3();
14:     }
15:     else {
16:         f1'();
17:     }
18: }
```
Mask Stack Enables Divergence

```c
1: if (TID % 2 == 0) {
2:   f2();
3:   if (TID % 4 == 0) {
4:     f4();
5:   }
6:   else {
7:     f2'();
8:   }
9: }
10: else {
11:   f(1);
12:   if (TID % 3 == 0) {
13:     f3();
14:   }
15:   else {
16:     f1'();
17:   }
18: }
```
Mask Stack Enables Divergence

IP

<table>
<thead>
<tr>
<th>enable mask</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1: if (TID % 2 == 0) {
2:   f2();
3:   if (TID % 4 == 0) {
4:     f4();
5:   }
6:   else {
7:     f2'();
8:   }
9: }
10: else {
11:   f(1);
12:   if (TID % 3 == 0) {
13:     f3();
14:   }
15:   else {
16:     f1'();
17:   }
18: }

DirectX 10 specifies 4-deep stack
Predication Eliminates Branches (and Divergence)

```c
if (TID % 2 == 0) {
    f2();
    if (TID % 4 == 0) {
        f4();
    }
    else {
        f2p();
    }
}
else {
    f(1);
    if (TID % 3 == 0) {
        f3();
    }
    else {
        f1p();
    }
}
```
Predication Eliminates Branches (and Divergence)

\[ p_1 = (TID \% 2 == 0) \]
\[ f2(); \]
\[ \text{if} \ (TID \% 2 == 0) \{ \]
\[ f2(); \]
\[ \text{if} \ (TID \% 4 == 0) \{ \]
\[ f4(); \]
\[ \} \]
\[ \text{else} \ { \]
\[ f2'(); \]
\[ \} \]
\[ \} \]
\[ \text{else} \ { \]
\[ f(1); \]
\[ \text{if} \ (TID \% 3 == 0) \{ \]
\[ f3(); \]
\[ \} \]
\[ \text{else} \ { \]
\[ f1'(); \]
\[ \} \]
Predication Eliminates Branches (and Divergence)

\[
p_1 = (\text{TID} \mod 2 == 0) \\
f_2(); \\
p_1 \quad p_2 = (\text{TID} \mod 4 == 0) \\
f_4(); \\
\]

if (TID \mod 2 == 0) {
  f_2();
} else {
  f_2'();
}

if (TID \mod 4 == 0) {
  f_4();
} else {
  f_2'();
}

else {
  f(1);
  if (TID \mod 3 == 0) {
    f_3();
  } else {
    f_1'();
  }
}

}
Predication Eliminates Branches (and Divergence)

\begin{minipage}{.25\textwidth}
\begin{verbatim}
p1 = (TID % 2 == 0) p1
f2();
p1 p2 = (TID % 4 == 0) p2
f4();
p1 p3 = !p2 p3
f2'();
p4 = !p1 p4
f(1);
p4 p5 = (TID % 3 == 0) p5
f3();
p4 p6 = !p5 p6 f1'();
\end{verbatim}
\end{minipage}

\begin{minipage}{.75\textwidth}
\begin{verbatim}
\textbf{if} (TID \% 2 == 0) {
\quad f2();
\textbf{if} (TID \% 4 == 0) {
\quad f4();
\quad }\textbf{else} {
\quad f2'();
\quad }
\textbf{else} {
\quad f(1);
\textbf{if} (TID \% 3 == 0) {
\quad f3();
\quad }\textbf{else} {
\quad f1'();
\quad }
\}
\end{verbatim}
\end{minipage}
Equivalence of Divergence and Predication

\[
\begin{align*}
\text{p1} & = (\text{TID} \mod 2 == 0) \\
\text{if (TID} \mod 2 == 0) \{ & \text{f2();} \\
\text{if (TID} \mod 4 == 0) \{ & \text{f4();} \\
\text{else } & \{ \text{f2'();} \\
\}\} \\
\}\}
\end{align*}
\]

\[
\begin{align*}
\text{p1} & \quad \text{p2} = (\text{TID} \mod 4 == 0) \\
\text{p2} & \quad \text{f4();} \\
\text{p1} \quad \text{p3} & = \neg \text{p2} \\
\text{p3} & \quad \text{f2'();} \\
\text{p4} & = \neg \text{p1} \\
\text{p4} & \quad \text{f(1);} \\
\text{p4} \quad \text{p5} & = (\text{TID} \mod 3 == 0) \\
\text{p5} & \quad \text{f3();} \\
\text{p4} \quad \text{p6} & = \neg \text{p5} \\
\text{p6} & \quad \text{f1'();} \\
\end{align*}
\]
When to Predicate and When to Diverge?

• Divergence
  - No performance penalty if all warp branches the same way
  - Some extra HW cost
  - Static partitioning of stack resources (to warps)

• Predication
  - Always execute all paths
  - Expose more ILP
  - Add predication registers to instruction encoding

• Selects - software predication
  - Simpler HW and just as flexible mode
  - Simple instruction encoding
  - Need to use more registers and insert select instructions
Outline

• CUDA
  - Overview
  - Development process
  - Performance Optimization
  - Syntax

• Most slides courtesy Massimiliano Fatica (NVIDIA)
Compute Unified Device Architecture

• CUDA is a programming system for utilizing the G80 processor for compute
  - CUDA follows the architecture very closely

• General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor

Matches architecture features
Specific parameters not exposed
The CUDA Platform

- The G80 GPU is not integrated into the CPU
  - Connects through a PCI Express bus
  - Communicates through OS (drivers)
CUDA Programming System

- **Targeted software stack**
  - Compute oriented drivers, language, and tools

- **Driver for loading computation programs into GPU**
  - Standalone Driver - Optimized for computation
  - Interface designed for compute-graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management
Overall Performance Can be Limited by Interface
Overall Performance Can be Limited by Interface
CUDA API and Language: Easy and Lightweight

- The API is an extension to the ANSI C programming language
  - Low learning curve

- The hardware is designed to enable lightweight runtime and driver
  - High performance
CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel

- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few
CUDA is an Extension to C

Integrated source
(foo.cu)

cudacc
EDG C/C++ frontend
Open64 Global Optimizer

GPU Assembly
(foo.s)

OCG

G80 SASS
(foo.sass)

CPU Host Code
(foo.cpp)

gcc / cl

© David Kirk/NVIDIA and
Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois,
Urbana-Champaign
CUDA is an Extension to C

- **Declspecs**
  - global, device, shared, local, constant

- **Keywords**
  - threadIdx, blockIdx

- **Intrinsics**
  - __syncthreads

- **Runtime API**
  - Memory, symbol, execution management

- **Function launch**

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
    __shared__ float region[M];
    ...
    region[threadIdx] = image[i];
    __syncthreads()
    ...
    image[j] = result;
}
```

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>>(myimage);