CSCI-GA.3033-008

Graphics Processing Units (GPUs): Architecture and Programming

Lecture 11: What’s Next: Hardware

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
This Lecture

- Power and temperature
- Reliability
- Nvidia Kepler
Power and Temperature
The Power-Wall

- Why do we have power-wall?
- Techniques to solve the problem
- Real-life example processor
- What can you do in software?
  - Is there power-aware software?
Increasing number of cores

Source: http://www.prism.gatech.edu/~shong9/ISCA_2010.pptx (disappeared!)
Power is also increasing!

Source: http://www.prism.gatech.edu/~shong9/ISCA_2010.pptx (disappeared!)
The Problem

• Cooling for GPUs is becoming prohibitively expensive.
  – Exasperated by the low profit margins in these market segments

• Today’s cooling solutions are designed for worst-case behavior.

• Reducing the hot spots will help reduce cooling requirements.
Why Power Aware?

- Servers and Workstations
  - Packaging cost
  - High temperature = more expensive cooling system
- Embedded Devices
  - Battery Life
  - No place for fans, etc.

![Graph: Cooling cost vs Thermal dissipation](image)
Moore’s Law

More transistors/mm$^2$
Moore's Law

- More transistors/mm$^2$
- More activity/area
- Higher speed
Moore’s Law

- More transistors/mm²
  - More activity/area
    - Higher speed
  - More switches/cycle
  - More power density
    - Higher temperature
    - Lower speed
Moore's Law

More transistors/mm\(^2\) → More activity/area → Higher speed

More switches/cycle

More power density → Higher temperature

Lower speed
Power-Aware Computing

- Dynamic Power Consumption
- Static Power Consumption
- Temperature
So ... What is it about Moore’s law?

- Power and temperature are becoming crucial
- \( \text{GPU power consumption} = \text{Runtime power} + \text{idle power} \)
- \( \text{Power} = \text{dynamic} + \text{static} \)
- Given power budget, how to get best performance?
- Given required performance, how to achieve it with lowest power?
GeForce 285 GTX

- FMA with only single thread
- Fully exercise computing units
- Copies a chunk of 200MB to GPU DRAM
- Mult of two 8K matrices

Bar chart showing average power in watts for different tasks:
- Idle: 47 watts
- FMA1: 92.8 watts
- FMA: 123.1 watts
- Memcpv: 171.8 watts
- Matmul: 168.8 watts
Be Careful!

• Static power is no longer trivial
• Higher utilization does not necessarily mean higher performance but for sure means higher power consumption/dissipation
• **Goal:** maximize performance/watt
Power-Aware Computing is:

Reducing power without losing performance
Dynamic Power Consumption

\[ P_{\text{dynamic}} = \alpha CV_{DD}^2 f A \]
Dynamic Power Consumption

\[ P_{\text{dynamic}} = \alpha CV_{DD}^2 f A \]

- depends on the wire lengths
- supply voltage
- clock frequency
- between 0 and 1
- how often wires transition
Static Power Consumption

• 20% or more in sub-micron era
• Mostly leakage
  - represents the power dissipated by a transistor whose gate is intended to be off

\[ P = V \left( ke^{-qV_{th}/(ak_a T)} \right) \]
Temperature

• Lost power
• Leakage increases by order of magnitude at high temperature
• Higher temperature = lower mean-time-to-failure (MTTF)
• We need temperature-aware computing
Temperature -> Hot Spot

applu benchmark on a single core (source: Kevin Skadron Tutorial in ISCA’04)
What To Do About Dynamic Power

• DFVS
• Reducing switching activity
What To Do About Leakage?

- Stacking transistor
- Dynamically resized caches (mainly I-caches)
  - gated Vdd
  - Non-state-preserving
- Drowsy caches
  - Scale supply voltage to reduce leakage
What To Do About Temperature?

• Better sensors position
• Predicting temperature at places without sensors
• Avoid hot spots
• Must be taken care of from design-time
Real-Life Example: SandyBridge

PMA: Power Management Agent
PCU: Package Control Unit
DMI: Direct Media Interface
SVID: Serial Voltage ID
PECI: Platform Environment Control Interface
IMC: Integrated Memory Controller
Real-Life Example: SandyBridge

Two independent power planes:
• CPU cores, LLC, and ring
  • Each core can be turned off indept.
  • Portion of the LLC can be tuned off
• GPU

• On chip logic and embedded controller running power management firmware
• Communicates internally with cores, ring
• Monitors physical conditions Voltage, temperature, power consumption
• Controls power states CPU and GPU voltage and frequency
Real-Life Example: SandyBridge

Power Performance Management Is:

**Enhance User Experience:**
- Throughput performance
- Responsiveness - burst performance
- CPU / PG performance
- Battery life / Energy bills
- Ergonomics (acoustic noise, heat)

**Given Physical Constraints:**
- Silicon capabilities
- System Thermo-Mechanical capabilities
- Power delivery capabilities
- S/W and Operating system explicit control
- Workload and usage
CPU-GPU Interaction

The graph illustrates the performance boost over a 0-percentage graphics budget at varying power allocations to graphics. Two benchmarks are shown: Crysis-dx9 and UT3-dx9. The performance boost increases with higher power allocation percentages.

- **Crysis-dx9**: The performance boost is significant and increases rapidly at lower power allocations before leveling off at higher percentages.
- **UT3-dx9**: The performance boost is more gradual compared to Crysis-dx9, with a slower increase and a more consistent performance boost across the range of power allocations.
Power-Aware Software!!
Is It for Real?
What Can A Software Application Do?

• Use less expensive operations
• Less stress on power-hungry parts
• Access and make use of internal GPU performance counters
  – PAPI
  – Nvidia Management Library (NVL)
• Interaction of three players:
  – The application software
  – The Compiler
  – The OS
Power-Aware Applications

- Applications must be Designed and tested for power management
- Applications must handle sleep transitions seamlessly
- You can differentiate your application with power management features
  - Handle power management events
  - Scale behavior based on user’s power preference
NVIDIA NML

- Nvidia Management Library
- C based interface for monitoring and managing various states within NVIDIA GPUs
- You call it from the host
- Compile with -lnvidia-ml
- You must have installed Nvidia CUDA toolkit and Nvidia development drivers
#include <stdio.h>
#include <nvml.h>
int main()
{
    nvmlReturn_t result;
    unsigned int device_count, i;
    char version[80];
    result = nvmlInit();
    result = nvmlSystemGetDriverVersion(version, 80);
    result = nvmlDeviceGetCount(&device_count);
}
Power Breakdown: GeForce 285 GTX
GPU Reliability
Reliability

- Error-rates are expected to increase with future process technology
- Reliability in GPGPUs is not as addressed as other aspects
- Graphics applications may be fault tolerant, but other applications running on GPGPUs are not.
- Large scale GPU failure after shipment/deployment is not uncommon
- How can the hardware help?
- Can software help in this?
We target permanent faults of SMs,
In our experiment we found that a loss of an SM in an 8-SM GPU can cause performance loss as high as 16%!
What if one of the SMs fails??

First Solution:
Turn off the faulty part.

Can we do better?
What if one of the SMs fails??

Better Solution:
Use faulty parts to give hints to non-faulty part to speed them up
A Hint??

• **What?** is a piece of information which can be used to speed up the execution

• **Why?** A hint can enhance performance of execution by prefetching from memory, etc.

• **How?** Make a faulty SM generate hints to be used by another SM
Which Hints to Use?

• We explored several types
• Based on their effectiveness and hardware requirements we narrowed them down to 3
  – Instruction cache prefetch to the SM instruction cache
  – Inter-SM warp memory coalescing
  – Instruction prefetch to the shared L2 cache
Hint 1: Per-SM Cache Prefetch

• Each SM has a private instruction cache
• The faulty SM does not commit any results so can execute faster
• The faulty SM gives hints to the functional SM that it may need in the future
Hint 2: Memory Coalescing

- The coalescing hardware already exists
- The faulty SM generates addresses to the memory controller that can make the controller bring data needed by the functional SM in the future
Hint 3: Shared Instruction Cache Prefetch

• Similar to hint 1 but here for the instruction cache
Why Targeting SM and not SP?

- SPs are usually executing the same code and work in lockstep
- So we cannot speed up another SP in the same SM
Hardware Needed

- Hint gathering unit
- Hint processing unit (also known as hint distribution unit)
- Hint disabling unit
- These units are present in all SMs as we don’t know when an SM will fail
Instruction Classifications

TEX: Texture – MEM: loads and stores
SFU/DPU: Special function and Divide and Multiply units
Control: branches - MAD: Multiply and Add – ALU: Arithmetic and Logic
Speedup

![Bar chart showing speedup over baseline for various categories: AES, BFS, LIB, LPS, MUM, NN, NOU, RAY, STO, WP.]
What Do you think you can do as a programmer regarding Reliability?
NVIDIA Kepler
Kepler

- Provides much enhanced support vs. Fermi (Exposed in NVML and nvidia-smi)
  - Set power limit
  - Set fixed maximum clocks
  - Query performance limiting factors
- **Hyper-Q**: allows 32 simultaneous, hardware-managed streams
- **GPUDirect**: allows direct access to GPU memory by third-party devices such as NICs, and SSDs.
- **Dynamic parallelism**: In Kepler GK110 any kernel can launch another kernel, and can create the necessary streams, events and manage the dependencies needed to process additional work without the need for host CPU interaction.
Control must be transferred back to CPU before a new kernel can execute.

Only return to CPU when all GPU operations are completed. Why is this faster?
Kepler GK110 supports the new CUDA Compute Capability 3.5

<table>
<thead>
<tr>
<th>Feature</th>
<th>FERMI GF100</th>
<th>FERMI GF104</th>
<th>KEPLER GK104</th>
<th>KEPLER GK110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps / Multiprocessor</td>
<td>48</td>
<td>48</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / Multiprocessor</td>
<td>1536</td>
<td>1536</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks / Multiprocessor</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32-bit Registers / Multiprocessor</td>
<td>32768</td>
<td>32768</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>255</td>
</tr>
<tr>
<td>Max Threads / Thread Block</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size Configurations (bytes)</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
</tr>
<tr>
<td></td>
<td>48K</td>
<td>48K</td>
<td>32K</td>
<td>32K</td>
</tr>
<tr>
<td>Max X Grid Dimension</td>
<td>$2^{16}-1$</td>
<td>$2^{16}-1$</td>
<td>$2^{32}-1$</td>
<td>$2^{32}-1$</td>
</tr>
<tr>
<td>Hyper-Q</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic Parallelism</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Compute Capability of Fermi and Kepler GPUs
New streaming multiprocessor (now called SMX)

192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

Full Kepler GK110 has 15 SMXs
Some products may have 13 or 14 SMXs
Conclusions

- There are several walls that GPU must face: power/temperature, reliability, and bandwidth.
- You must keep track of these walls when writing your software.
- Correctness + speed + power efficiency + reliability