The Fermi Architecture

Michael C. Shebanow
Principal Research Scientist,
NV Research
mshebanow@nvidia.com
Fermi GF100 Overview
Fermi GF100 GPU
## Comparing Fermi with GT200 & G80

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>GT200</th>
<th>GF100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>681 million</td>
<td>1.4 billion</td>
<td>3.0 billion</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double Precision Floating Point</td>
<td>None</td>
<td>30 FMA ops / clock</td>
<td>256 FMA ops /clock</td>
</tr>
<tr>
<td>Single Precision Floating Point</td>
<td>128 MAD ops/clock</td>
<td>240 MAD ops / clock</td>
<td>512 FMA ops /clock</td>
</tr>
<tr>
<td>Special Function Units / SM</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Warp schedulers (per SM)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared Memory (per SM)</td>
<td>16 KB</td>
<td>16 KB</td>
<td>Configurable 48 KB or 16 KB</td>
</tr>
<tr>
<td>L1 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>Configurable 16 KB or 48 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>ECC Memory Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent Kernels</td>
<td>No</td>
<td>No</td>
<td>Up to 16</td>
</tr>
<tr>
<td>Load/Store Address Width</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>
Soul of Fermi

- Expand performance sweet spot of the GPU
  - Caching
  - Concurrent kernels
  - FP64

- Bring more users, more applications to the GPU
  - C++
  - Visual Studio Integration
  - ECC
Fermi Focus Areas

- Improved peak performance
- Improved efficiency throughput
- Broader applicability
- Full integration within modern software development environment
### GeForce GTX 480

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>1536MB / 384-bit GDDR5</td>
</tr>
<tr>
<td>Cores</td>
<td>480</td>
</tr>
<tr>
<td>Gfx / Proc / Mem Clock</td>
<td>700 / 1401 / 1848 MHz</td>
</tr>
<tr>
<td>Power Connectors</td>
<td>6-pin + 8-pin</td>
</tr>
<tr>
<td>Power</td>
<td>250W</td>
</tr>
<tr>
<td>SLI</td>
<td>3-way</td>
</tr>
<tr>
<td>Length</td>
<td>10.5 inches</td>
</tr>
<tr>
<td>Thermal</td>
<td>Dual Slot Fansink</td>
</tr>
<tr>
<td>Outputs</td>
<td>DL-DVI, DL-DVI, mini-HDMI</td>
</tr>
</tbody>
</table>

### GeForce GTX 470

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>1280MB / 320-bit GDDR5</td>
</tr>
<tr>
<td>Cores</td>
<td>448</td>
</tr>
<tr>
<td>Gfx / Proc / Mem Clock</td>
<td>607 / 1215 / 1674 MHz</td>
</tr>
<tr>
<td>Power Connectors</td>
<td>2x 6-pin</td>
</tr>
<tr>
<td>Power</td>
<td>215W</td>
</tr>
<tr>
<td>SLI</td>
<td>3-way</td>
</tr>
<tr>
<td>Length</td>
<td>9.5 inches</td>
</tr>
<tr>
<td>Thermal</td>
<td>Dual Slot Fansink</td>
</tr>
<tr>
<td>Outputs</td>
<td>DL-DVI, DL-DVI, mini-HDMI</td>
</tr>
</tbody>
</table>
Improving Geometric Realism: Tessellation

State of the Art in Games

State of the Art in Film

© Disney Enterprises, Inc. and Jerry Bruckheimer, Inc. All rights reserved. Image courtesy Industrial Light & Magic.
What is Tessellation?
Tessellation Applications

- Realistic water: Up to 1.6e9 triangles/sec
- Hair: 18,000 hair strands (~4x vs. prior demos)
Tessellation Off/On
**Much Better Compute**

- **Programmability**
  - C++ Support
  - Exceptions/Debug support

- **Performance**
  - Dual issue SMs
  - L1 cache
  - Larger Shared
  - Much better DP math
  - Much better atomic support

- **Reliability: ECC**

<table>
<thead>
<tr>
<th></th>
<th>GT200</th>
<th>GF100</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Texture</td>
<td>12 KB</td>
<td>12 KB</td>
<td>Fast texture filtering</td>
</tr>
<tr>
<td>Cache (per quad)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dedicated</td>
<td></td>
<td>X</td>
<td>Efficient physics and ray tracing</td>
</tr>
<tr>
<td>L1 LD/ST Cache</td>
<td></td>
<td>16 or 48 KB</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>16KB</td>
<td>16 or 48 KB</td>
<td>More data reuse among threads</td>
</tr>
<tr>
<td>Shared Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256KB (TEX read only)</td>
<td>768 KB (all clients read/write)</td>
<td>Greater texture coverage, robust compute performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double Precision</td>
<td>30 FMAs/clock</td>
<td>256 FMAs/clock</td>
<td>Much higher throughputs for Scientific codes</td>
</tr>
<tr>
<td>Throughput</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fermi SM Architecture
Fermi SM

- **Objective – DX11 support**
  - Polymorph engine

- **Objective – Optimize for GPU computing**
  - New ISA
  - Revamp issue / control flow
  - New CUDA core architecture

- **32 cores per SM**
  - (512 cores total)

- **64KB configurable L1$ / shared memory**
CUDA Core Architecture

- Decoupled FP32 and integer execution datapaths
- Double precision throughput increased: now 50% of single precision peak
- Integer operations optimized for extended precision
  - 64 bit and wider data element size
- Predication field for all instructions
- Fused multiply add FP datapath
IEEE 754-2008 Floating Point

- IEEE 754-2008 results
  - 64-bit double precision
  - 32-bit single precision
  - Rounding, subnormals
  - NaNs, +/- Infinity
- IEEE 754-2008 rounding:
  - nearest even, zero, +inf, -inf
- Full-speed subnormal operands and subnormal results
- IEEE 754-2008 Fused Multiply-Add (FMA)
  - D = A*B + C;
  - No loss of precision
  - IEEE divide & sqrt use FMA

Multiply-Add (MAD):  \( D = A \times B + C; \)

\[
\begin{array}{c}
A \\
\times \\
B \\
\end{array}
\]

\[
= \text{Product} \quad \text{(truncate digits)}
\]

\[
+ \quad C
\]

\[
= \quad D
\]

Fused Multiply-Add (FMA):  \( D = A \times B + C; \)

\[
\begin{array}{c}
A \\
\times \\
B \\
\end{array}
\]

\[
= \text{Product} \quad \text{(retain all digits)}
\]

\[
+ \quad C
\]

\[
= \quad D \quad \text{(no loss of precision)}
\]
Instruction Set Architecture

- Enables C++: virtual functions, new/delete, try/catch
- Unified load/store addressing
- 64-bit addressing for large problems
- Optimized for CUDA C, OpenCL & Direct Compute
  - Native (x,y)-based LD/ST operations with format conversion
- Enables system call functionality – stdio.h, etc.
Multiple Memory Scopes

- **Per-thread private memory**
  - Each thread has its own local memory
  - Stacks, other private data

- **Per-thread-block shared memory**
  - Small memory close to the processor, low latency
  - Allocated per thread block

- **Main memory**
  - GPU frame buffer
  - Can be accessed by any thread in any thread block
Unified Load/Store Addressing

Non-unified Address Space

- Local
  - \( *p_{\text{local}} \)
- Shared
  - \( *p_{\text{shared}} \)
- Global
  - \( 0 \) to \( 32\)-bit
  - \( *p_{\text{global}} \)

Unified Address Space

- Local
- Shared
- Global
  - \( 0 \) to \( 40\)-bit
  - \( *p \)
Atomic Operations
(Read / Modify / Writes)

- **Fast inter-block, thread-safe communication**
  - Significantly more efficient chip-wide synchronization
  - Much faster parallel aggregation
    - Faster ray tracing, histogram computation, clustering and pattern recognition, face recognition, speech recognition, BLAS, etc
  - Accelerated by cached memory hierarchy

- **Fermi increases atomic performance by 5x to 20x**
ECC

- All major internal memories are ECC protected
  - Register file, L1 cache, L2 cache

- DRAM protected by ECC
  - ECC supported for GDDR5 as well as SDDR3 memory configurations

- ECC is a must have for many computing applications
  - Clear customer feedback
SM Operational Block Diagram

Fermi Dual Issue

- Warp Scheduler
- Instruction Dispatch Unit
- CUDA Cores (x16)
  - FADD
  - FFMA
  - IADD
- CUDA Cores (x16)
  - FFMA
  - IADD
  - MOV
- SFUs (x4)
  - RCP
  - SIN
- LD/ST Units (x16)
  - LD
  - ST
**Instruction Issue and Control Flow**

- **Decouple internal execution resources**
  - Deliver peak IPC on branchy / int-heavy / LD-ST - heavy codes
- **Dual issue pipelines select two warps to issue**

---

**Diagram Description**

- **Warp Scheduler**
  - Instruction Dispatch Unit
  - Time
  - Warps:
    - Warp 8: Instruction 11
    - Warp 2: Instruction 42
    - Warp 14: Instruction 95
    - Warp 8: Instruction 12
    - Warp 14: Instruction 96
    - Warp 2: Instruction 43
  - Warps:
    - Warp 9: Instruction 11
    - Warp 3: Instruction 33
    - Warp 15: Instruction 95
    - Warp 9: Instruction 12
    - Warp 3: Instruction 34
    - Warp 15: Instruction 96
Caches

- Configurable L1 cache per SM
  - 16KB L1$ / 48KB Shared Memory
  - 48KB L1$ / 16KB Shared Memory
- Shared 768KB L2 cache
- Compute motivation:
  - Caching captures locality, amplifies bandwidth
  - Caching more effective than Shared Memory RAM for irregular or unpredictable access
    - Ray tracing, sparse matrix multiply, physics kernels …
  - Caching helps latency sensitive cases
Cache Usage: Graphics

- Data stays on die
- L1 cache
  - Register spilling
  - Stack ops
  - Global LD/ST
- L2 Cache
  - Vertex, SM, Texture and ROP Data
GPU Computing Key Concepts

- **Hardware (HW) thread management**
  - HW thread launch and monitoring
  - HW thread switching
  - Tens of thousands of lightweight, concurrent threads
  - Real threads: PC, private registers, ...

- **SIMT execution model**

- **Multiple memory scopes**
  - Per-thread private memory
  - Per-thread-block shared memory
  - Global memory

- **Using threads to hide memory latency**

- **Coarse grain thread synchronization**
SM Limiter Theory Block Diagram

```
Warp 0
Warp 1
Warp K
Fetch Unit
Instruction Cache
Register Files
Warp 0
Warp 1
Warp K
Instruction Buffers
Sched Unit
PC
PC
PC
Instruction Cache
ALUs
LSU
Single Instruction (SI)
Multi-Threaded (MT)
```

© NVIDIA 2010
Limiter Theory

- SM a form of queuing system
- Use “limiter theory” to predict SM performance
  - Supply vs. Demand
- There are three types of limits on the performance of the SM:
  - Bandwidth resource limiters
  - Per-thread-block space limiters
  - Per-thread space limiters
- The most constraining limiter is called the critical limiter
  - $\text{min}(\text{all limiters})$
Bandwidth Limiters

- Thread blocks arrive at some rate $\lambda_{TB}$
- Threads composed of some distribution of operations (FMUL, FADD, LD, etc.)
  - Each arriving thread block of $S$ threads contributes to a distribution of operations to be performed
- Per operation type, the offered load, or BW “demand”, is the product of:
  - Thread block arrival rate $\lambda_{TB}$
  - # of threads $S$ in a block
  - Operation count $N_{op}$ in each thread
- BW “supply” $\lambda_{op}$
  - Throughput available for some operation “op”
- BW limiter equation:
  \[
  (\lambda_{TB} \times S \times N_{op}) \leq \lambda_{op} \Rightarrow \lambda_{TB} \leq \frac{\lambda_{op}}{S \times N_{op}}
  \]

BW “supply” is an upper bound on throughput

// Compute vector sum C = A+B  // Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
Space Limiters

SM also has space resources. Examples:
- Finite limit on warp count
- Finite limit on register file space
- Finite limit on shared memory size

Space resources:
- Allocated on thread block launch
- Deallocated on thread block completion
- Consumption computed using Little’s Law

Thread Latency (L)
- Complex computation
- Varies with memory behavior

Little’s Law:

\[ N = \lambda L \]

\( N = \) number “in flight”
\( \lambda = \) arrival rate (throughput)
\( L = \) service latency

Inverted:

\[ \lambda \leq \frac{N}{L} \]
Implications of Limiter Theory

- Limiter theory assumes uniform workloads
  - Breaks down if “traffic jam” behavior
  - Limiter theory is an ok 1st order approximation

- Kernel code has to pay careful attention to operation “mix”
  - Math-to-memory operation ratios for example
  - Do not want to bottleneck on one function unit leaving other units idling
  - Ideal: all units equally critical

- Don’t “traffic jam” kernel code
  - Making thread blocks too large so that only a few execute on the SM at a time
    a bad idea
  - “Bunching” operations of a similar type in one section of a kernel will
    aggravate the problem
  - Ideal: lots of small thread blocks with uniform distribution of operation
    densities

- Focus on space resource consumption
  - Ideal: use as few resources necessary to “load the SM”
Hiding LD Latency

- Principle:
  - Little’s Law again:
    \[ N = \lambda L \]
  - \( N \) = “number in flight”
  - \( \lambda \) = arrival rate
  - \( L \) = memory latency

- Arrival Rate product of:
  - Desired execution rate (IPC)
  - Density of LOAD instructions (%)
  - \( N \) = # of threads needed to cover latency \( L \)
Hiding LOAD Latency w/ Fewer Threads

- Use *batching*
- Group independent LDs together
- Modified law:
  \[
  N = \frac{\lambda L}{B}
  \]
  \(B = \text{batch size}\)

```c
// batch size 3 example
float *d_A, *d_B, *d_C;
float a, b, c, result;

a = *d_A; b = *d_B; c = *d_C;
result = a * b + c;
```

- The values ‘a’, ‘b’, and ‘c’ are loaded independently before being used
- Implication is that we can execute 3 loads from one thread before the first use (‘a’ in this case) causes a stall
Final Performance Tuning Thoughts

- **Threads are free**
  - A common mistake in GPU Computing kernels is to make threads do too much
  - Keep them short, sweet, & balanced
    - Example: one thread per vector element
  - HW provides LOTs of them (10s of thousands)
  - HW launch => near zero overhead to create them
  - HW context switching => near zero overhead scheduling

- **Barriers are cheap**
  - Single instruction
  - HW synchronization of thread blocks
  - Partition kernel code into producer-consumer
  - DON’T use spin locks!

- **Partition on results, not sources**
Thank You