The Fermi Architecture

Michael C. Shebanow

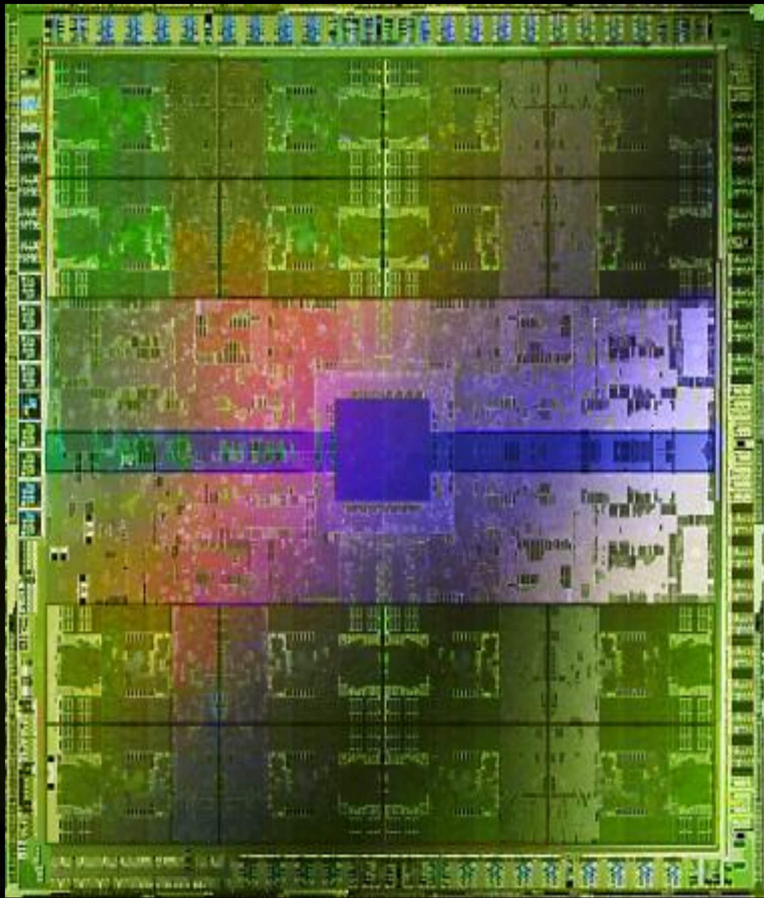
Principal Research Scientist,

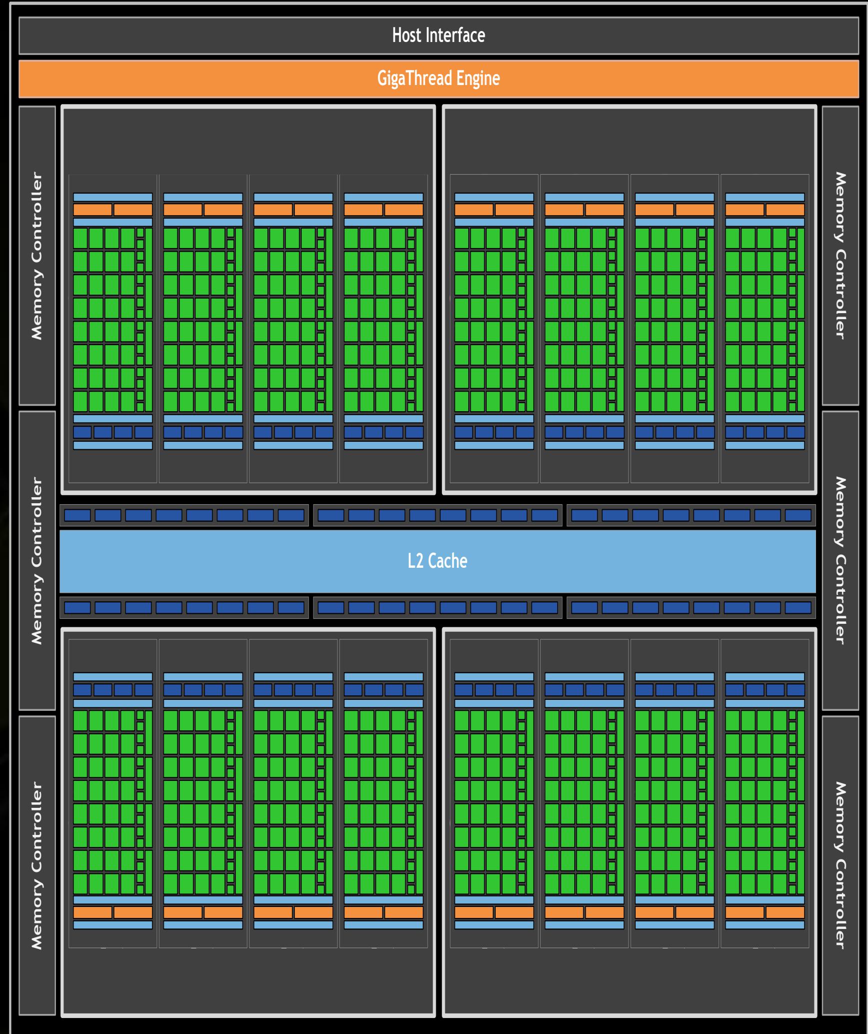
NV Research

mshebanow@nvidia.com

Fermi GF100
Overview

Fermi GF100 GPU





© NVIDIA 2010

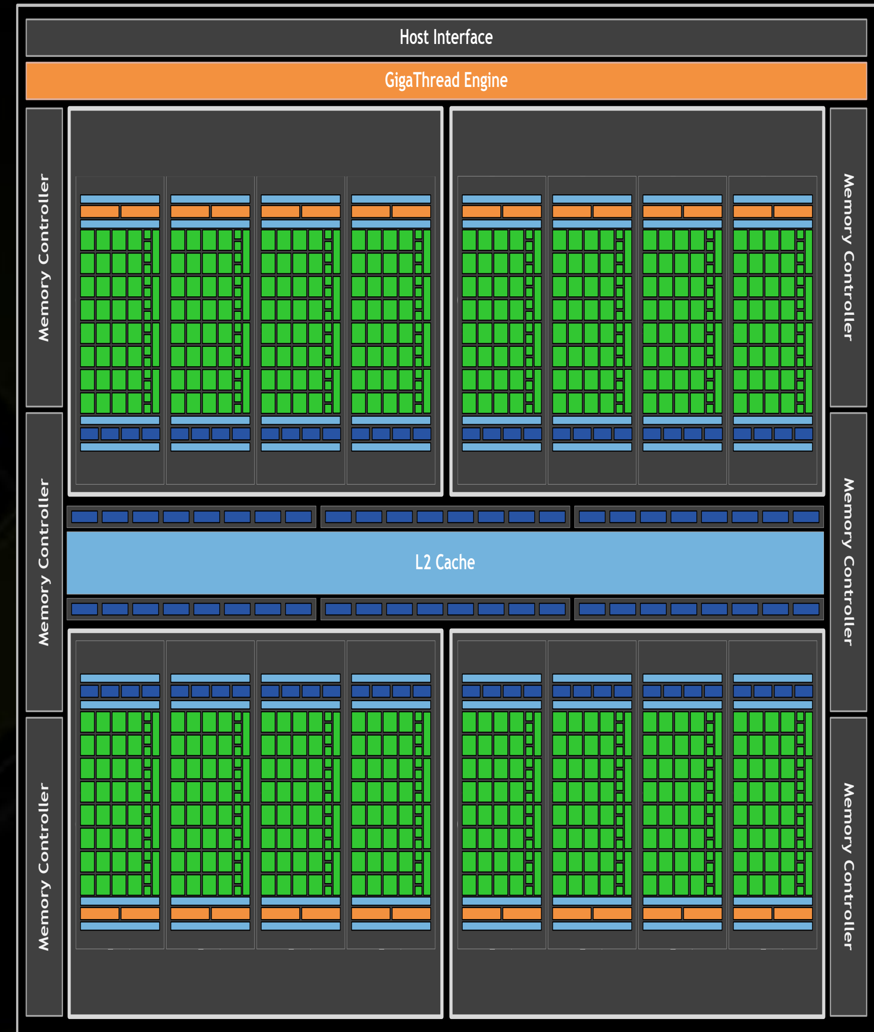
Comparing Fermi with GT200 & G80

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

© NVIDIA 2010

Soul of Fermi

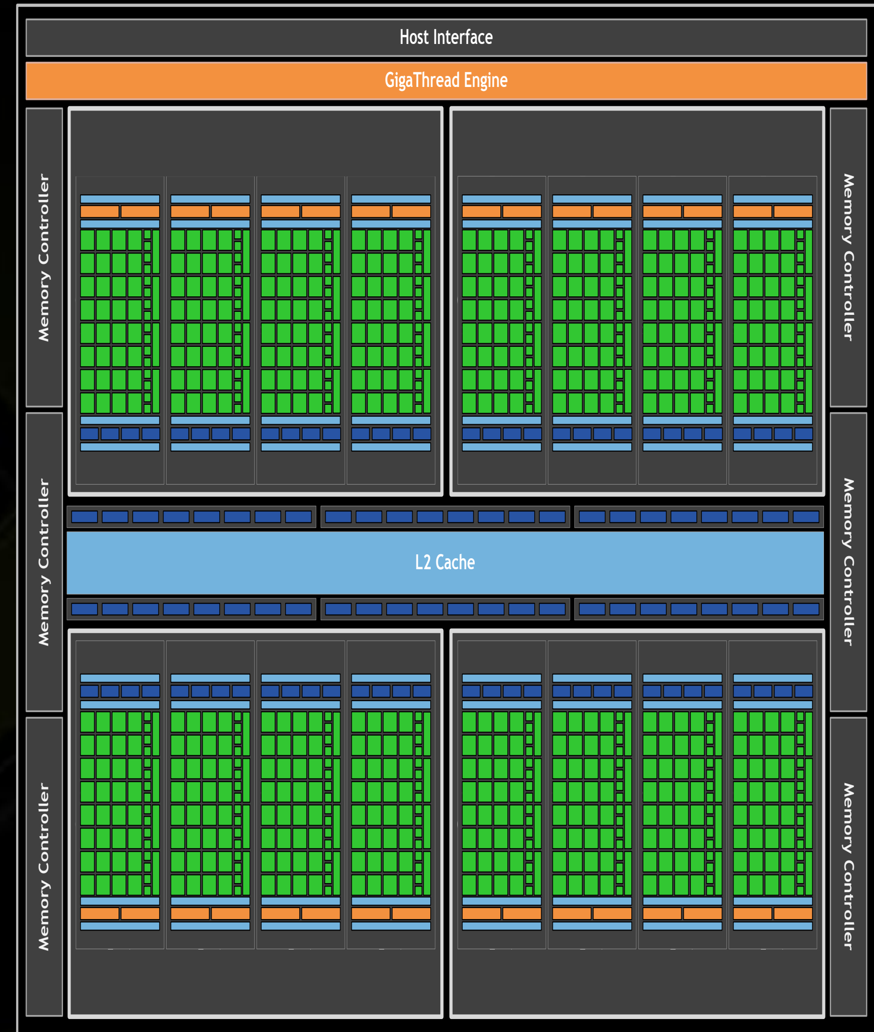
- **Expand performance sweet spot of the GPU**
  - Caching
  - Concurrent kernels
  - FP64
- **Bring more users, more applications to the GPU**
  - C++
  - Visual Studio Integration
  - ECC



© NVIDIA 2010

Fermi Focus Areas

- Improved peak performance
- Improved efficiency throughput
- Broader applicability
- Full integration within modern software development environment



© NVIDIA 2010

GeForce GTX 480 GeForce GTX 470

Memory	1536MB / 384-bit GDDR5
Cores	480
Gfx / Proc / Mem Clock	700 / 1401 / 1848 MHz
Power Connectors	6-pin + 8-pin
Power	250W
SLI	3-way
Length	10.5 inches
Thermal	Dual Slot Fansink
Outputs	DL-DVI DL-DVI mini-HDMI

Memory	1280MB / 320-bit GDDR5
Cores	448
Gfx / Proc / Mem Clock	607 / 1215 / 1674 MHz
Power Connectors	2x 6-pin
Power	215W
SLI	3-way
Length	9.5 inches
Thermal	Dual Slot Fansink
Outputs	DL-DVI DL-DVI mini-HDMI

© NVIDIA 2010

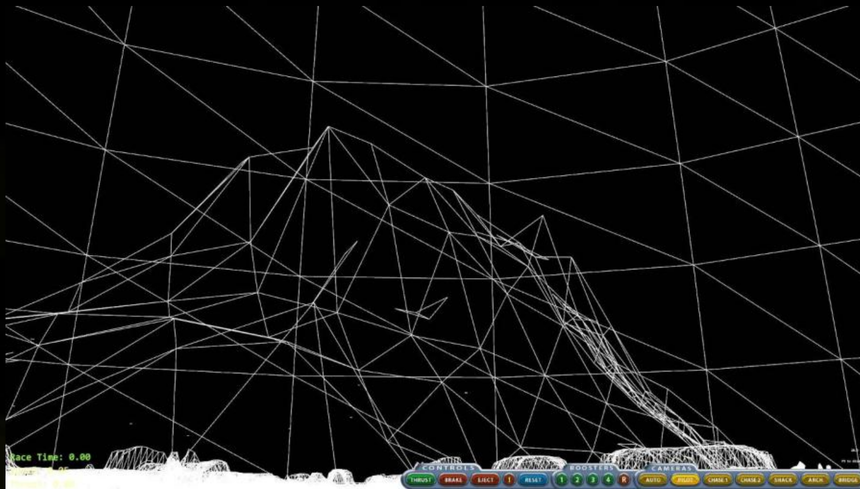
Improving Geometric Realism: Tessellation

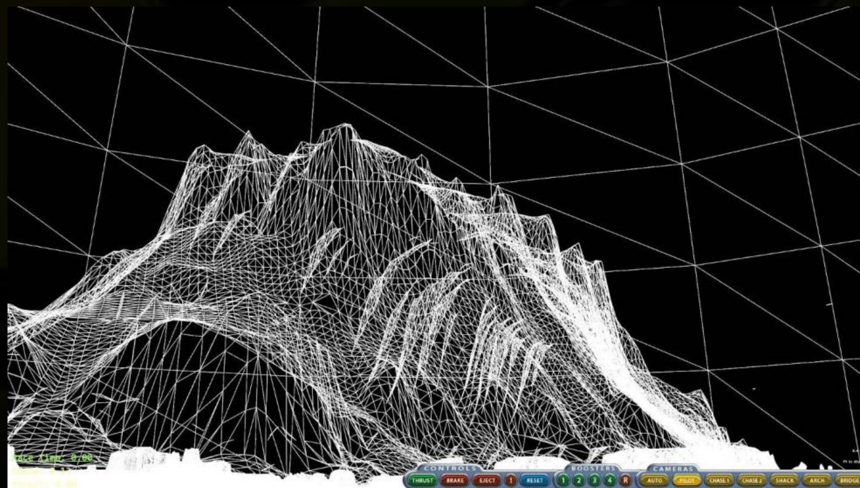
State of the Art in Games

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

© NVIDIA 2010

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

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

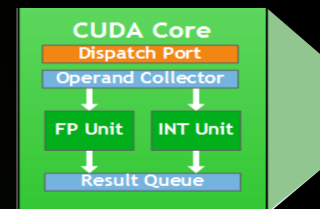
© NVIDIA 2010

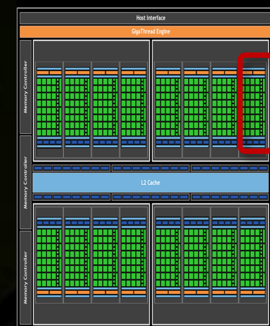
Fermi SM Architecture

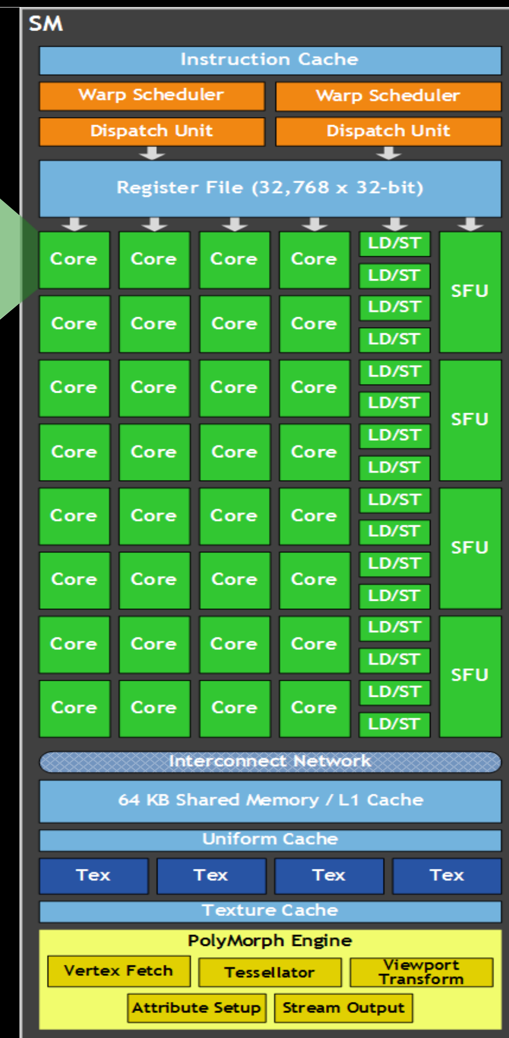
NVIDIA

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







	FP32	FP64	INT	SFU	LD/ST
Ops / clk	32	16	32	4	16

© NVIDIA 2010

CUDA Core Architecture

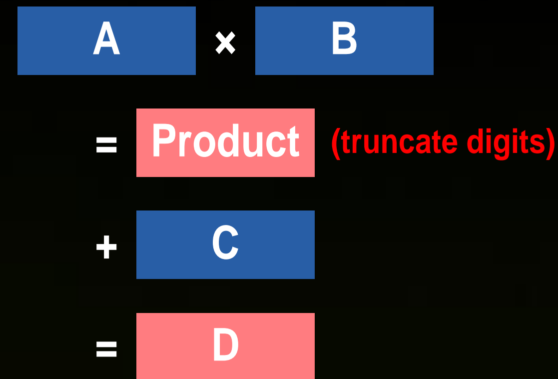
- Decoupled FP32 and integer execution datapaths
- Double precision throughput increased: now 50% of single precision peak
- Integer operations optimized for extended precision
  - 64 bit and wider data element size
- Predication field for all instructions
- Fused multiply add FP datapath

© NVIDIA 2010

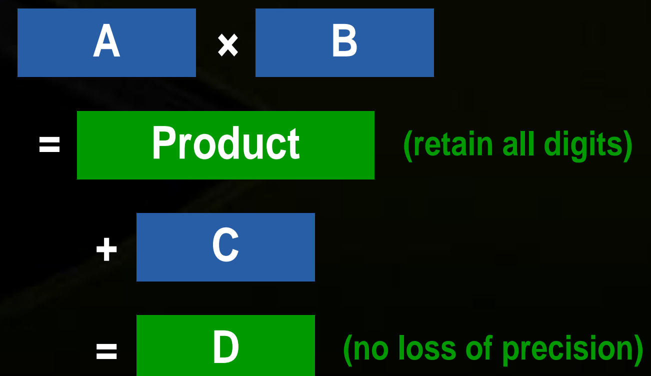
IEEE 754-2008 Floating Point

- 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*B + C;



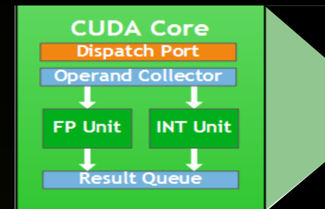
Fused Multiply-Add (FMA): D = A*B + C;

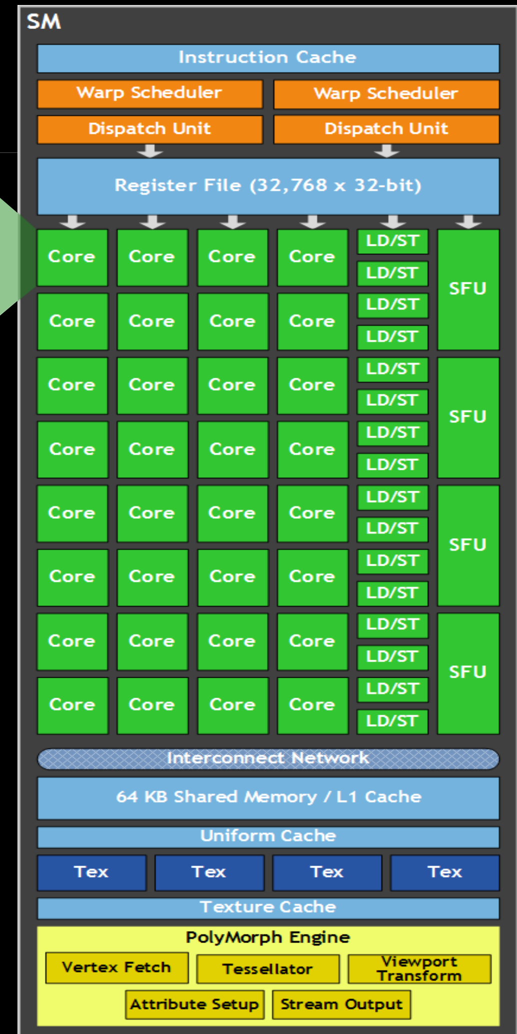


© NVIDIA 2010

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.



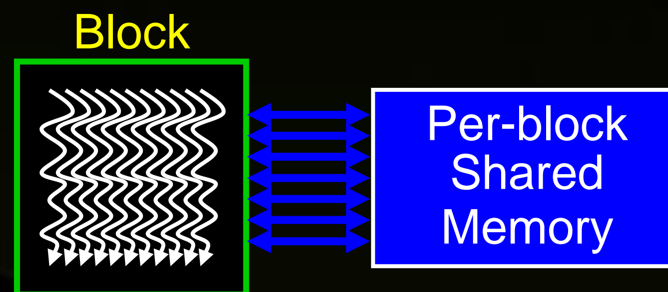


© NVIDIA 2010

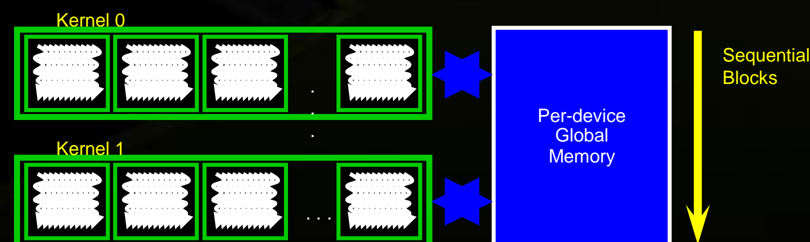
Multiple Memory Scopes

- 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

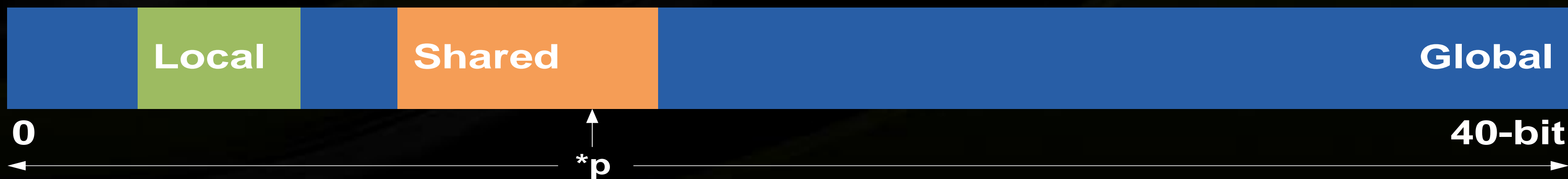


- Main memory
  - GPU frame buffer
  - Can be accessed by any thread in any thread block



© NVIDIA 2010

Unified Load/Store Addressing



© NVIDIA 2010

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**

© NVIDIA 2010

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

© NVIDIA 2010

SM Operational Block Diagram

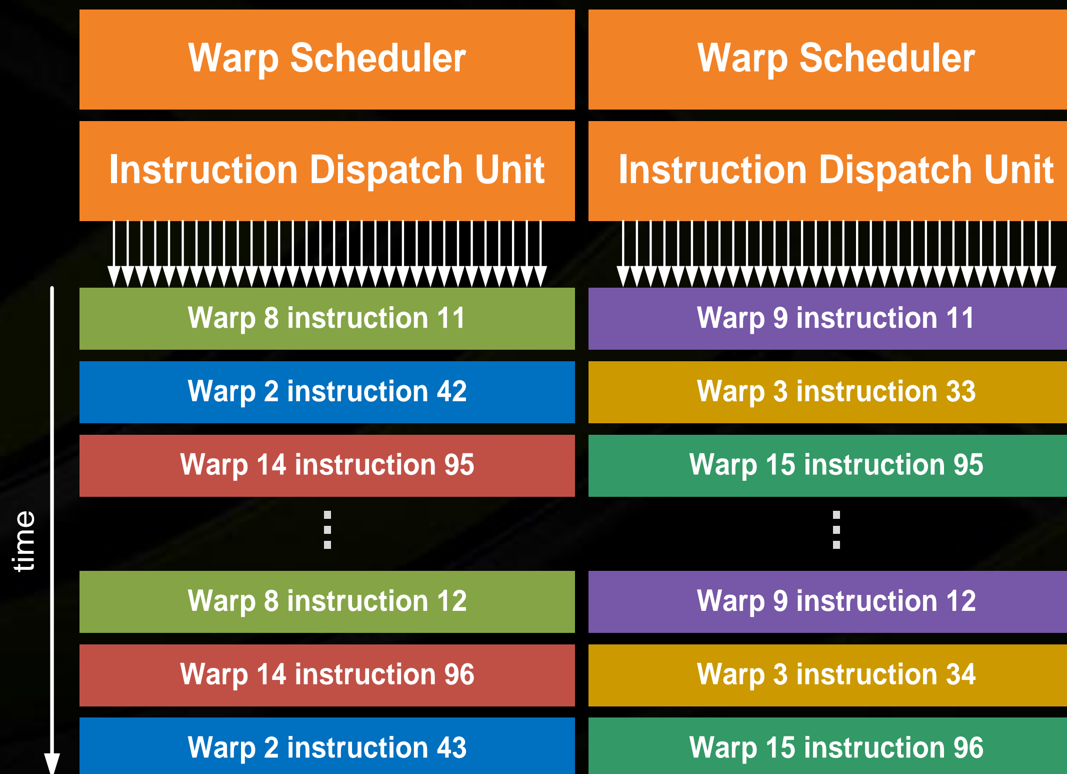
Fermi Dual Issue

Warp Scheduler
Warp Scheduler
Instruction Dispatch Unit
Instruction Dispatch Unit
CUDA Cores (x16)
CUDA Cores (x16)
SFUs (x4)
LD/ST Units (x16)
FADD
FFMA
IADD
FFMA
IADD
FFMA
FFMA
FADD
FFMA
IADD
MOV
ICMP
IADD
RCP
SIN
LD
LD
ST

© NVIDIA 2010

Instruction Issue and Control Flow

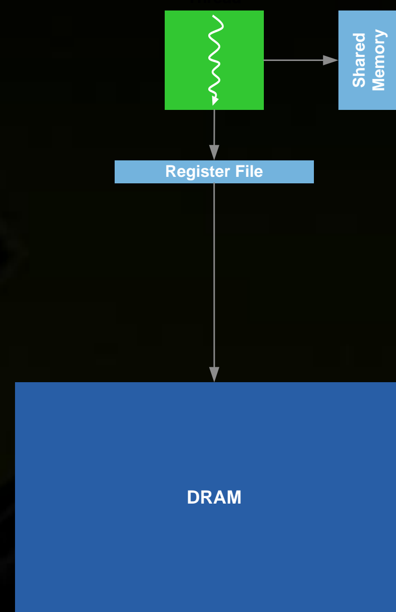
- **Decouple internal execution resources**
  - Deliver peak IPC on branchy / int-heavy / LD-ST - heavy codes
- **Dual issue pipelines select two warps to issue**

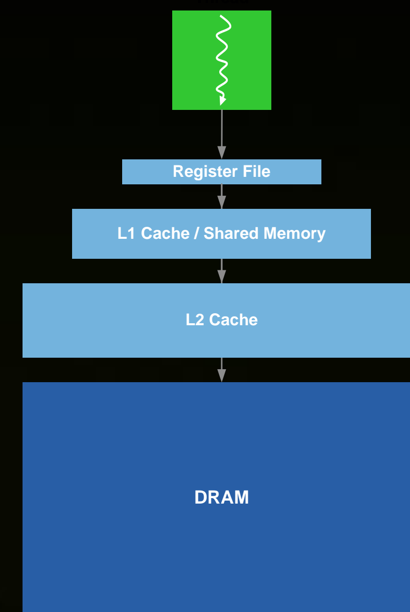


© NVIDIA 2010

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

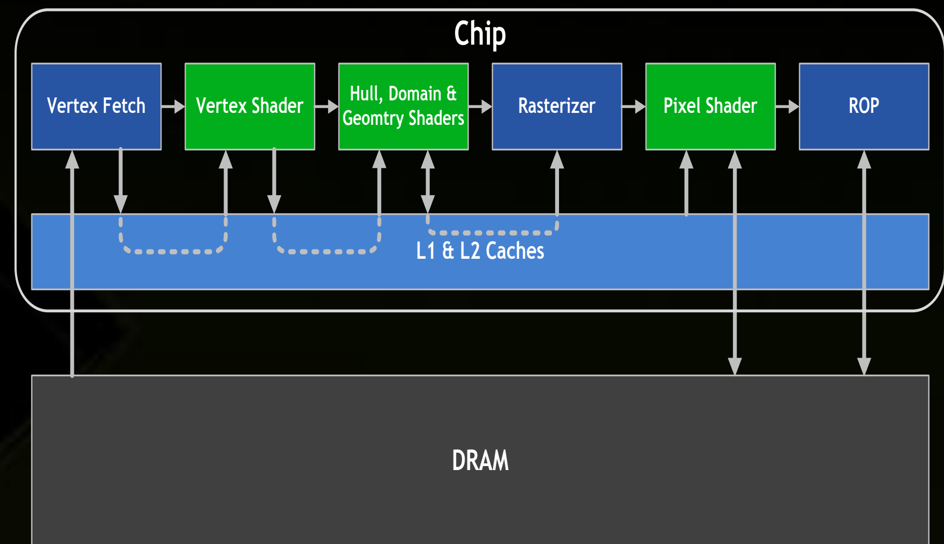




© NVIDIA 2010

Cache Usage: Graphics

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



© NVIDIA 2010

Fermi Application Tuning

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

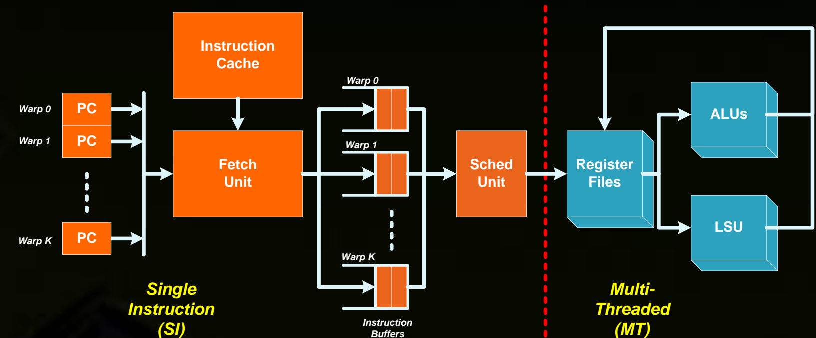
© NVIDIA 2010

SM Limiter Theory Block Diagram

© 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
  - = min(all limiters)



© NVIDIA 2010

Bandwidth Limiters

- Thread blocks arrive at some rate λ_{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 λ_{TB}
  - # of threads S in a block
  - Operation count N_{op} in each thread
- BW "supply" λ_{op}
  - Throughput availabl[illegible] peration[illegible]
- BW limiter equation:

Demand

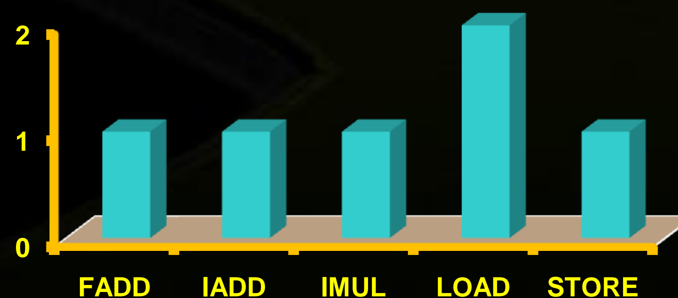
Supply

BW Limiter Equation

$$(\lambda_{TB} \times S \times N_{op}) \le \lambda_{op} \Rightarrow \lambda_{TB} \le \lambda_{op} / (S \times N_{op})$$

BW "supply" is an upper bound on throughput

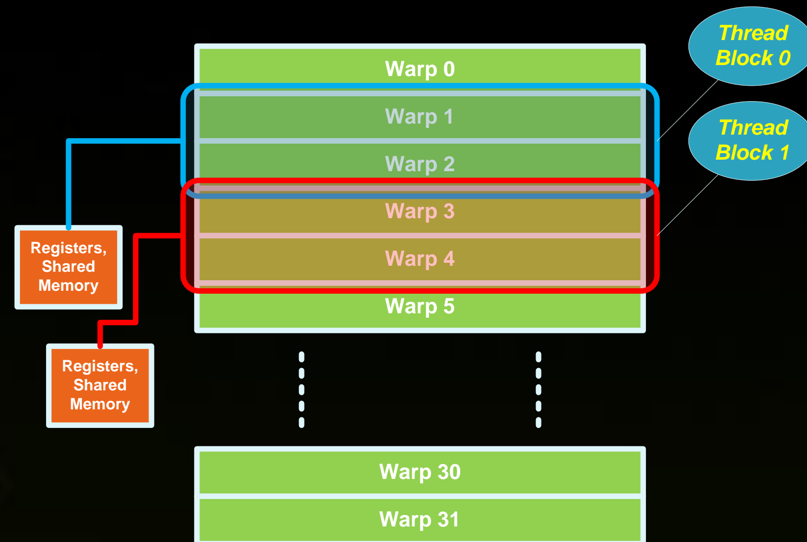
```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```



© NVIDIA 2010

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"
λ = arrival rate (throughput)
L = service latency

Inverted:

$$\lambda \leq N / L$$

© NVIDIA 2010

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"

© NVIDIA 2010

Hiding LD Latency

- **Principle:**
  - **Little's Law again:**

    $$N = \lambda L$$

  - **N = "number in flight"**
  - **λ = 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**

W_0T_0 Load Req

W_0T_1 Load Req

W_0T_2 Load Req

W_0T_0 Load Resp

W_0T_1 Load Resp

W_0T_2 Load Resp

W_kT_{29} Load Req

W_kT_{30} Load Req

W_kT_{31} Load Req

Time

© NVIDIA 2010

Hiding LOAD Latency w/ Fewer Threads

- Use *batching*
- Group independent LDs together
- Modified law:

$$N = \frac{\lambda L}{B}$$

  - B = batch size

```
// 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*

© NVIDIA 2010

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