Introduction to GPU Architectures for HPC Computing

Michael E. Thomadakis
Supercomputing Facility
Texas A&M University
miket(at)tamu(dot)edu

Outline

- Brief History and Background for GPU Computing
- Classical CPU vs. GPU Architecture
- Introduction to the CUDA Programming Model
- Introduction to Nvidia GPUs
- Architectural Context for GPU Technology
- Credits
- Disclaimer

Disclaimers

- No graphics or graphical processing background
- No prior GPU background or experience
- No guarantee that material is accurate (any possible attempt was made) – *under continuous improvement*
- No attempt to be thorough or general
  - GPU processing relies heavily on graphics and related technologies and algorithms
  - it has been around for some time now
  - it is a collection of specialized niche technologies

Brief History

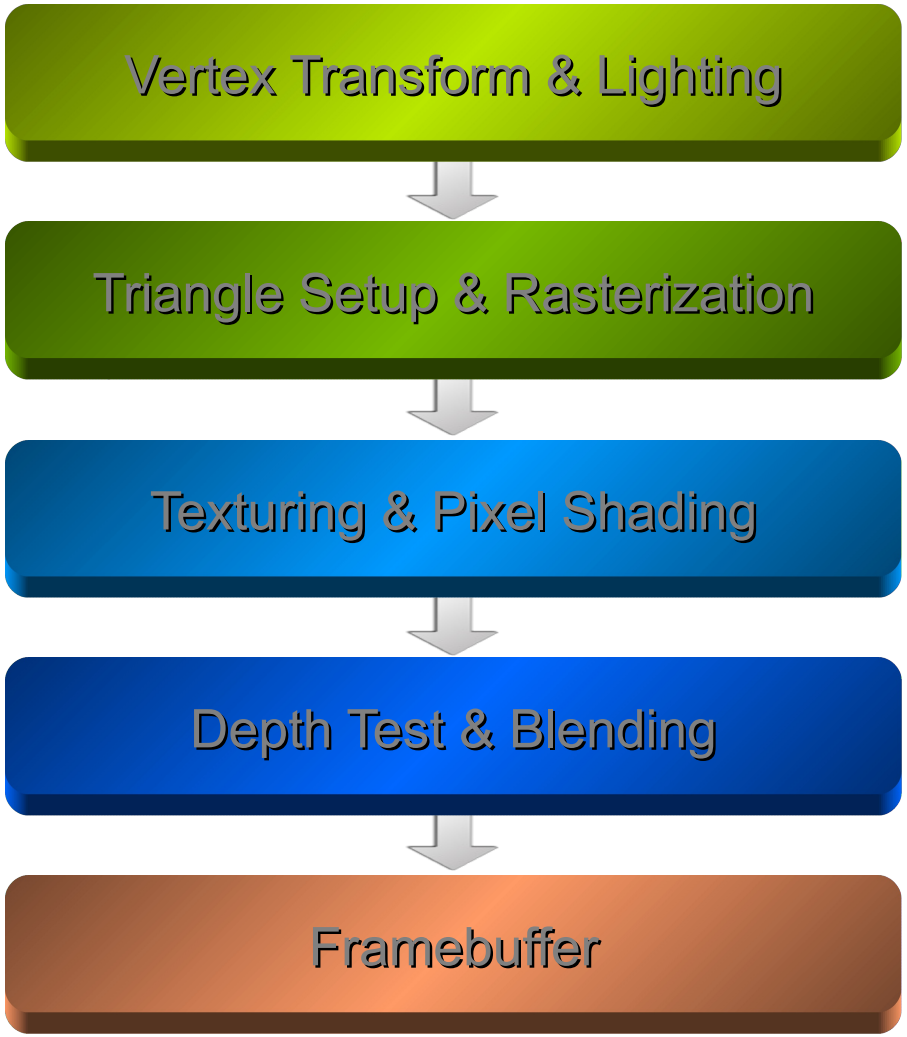
- Graphical Processing Units (GPUs) were invented in 1999 by nVidia
  - successors to "graphics pipeline" h/w (e.g., SGI)
  - h/w accelerator for real-life and real-time graphical, image and video / audio processing
  - GPUs have evolved into "massively" parallel specialized processing units
    - Moore's Law to increase on-chip parallelism and memory bandwidth
    - very high fixed and floating-point rates for specialized data-parallel computation
    - far outpace in FP arithmetic and memory access performance present day state of the art classical CISC or RISC cores
    - GPU h/w not suited well (yet) for general purpose scalar computing

Graphics h/w Backgrounder

- Graphics h/w is intended to off load the main CPU of tasks typical to graphical, image and video processing
  - real-life images
    - intricate shapes, visualization
    - complex optical effects (3D images, shading, illumination, etc.)
    - smooth motion
  - real-time rendering
    - meet hard real-time constraints in computation and memory access
- Massive parallelism by replication of integer and floatin-point ALUs, wide-memory access paths and pipelining

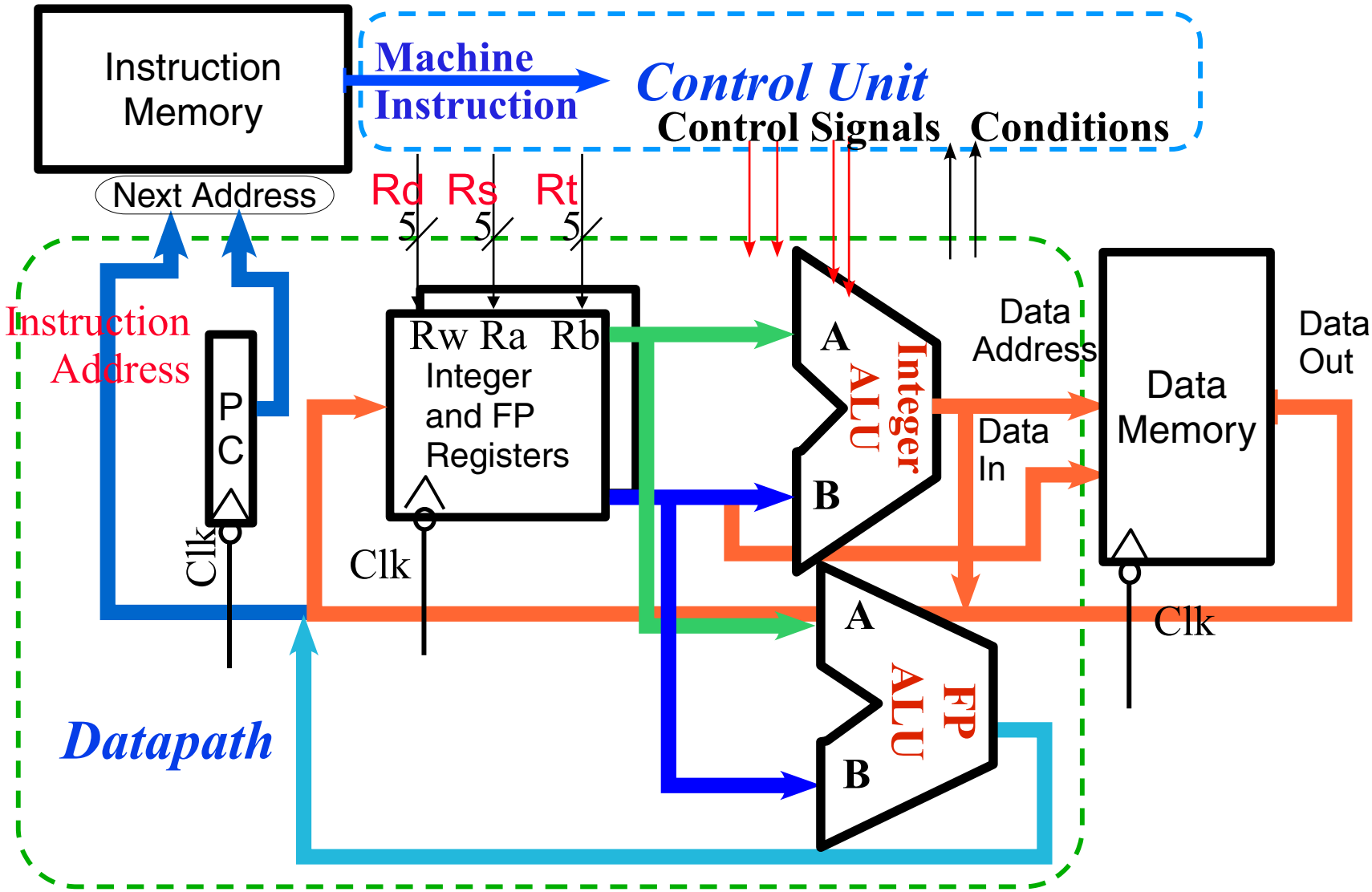
Graphics h/w Backgrounder

- **Traditional Graphics Pipeline h/w**



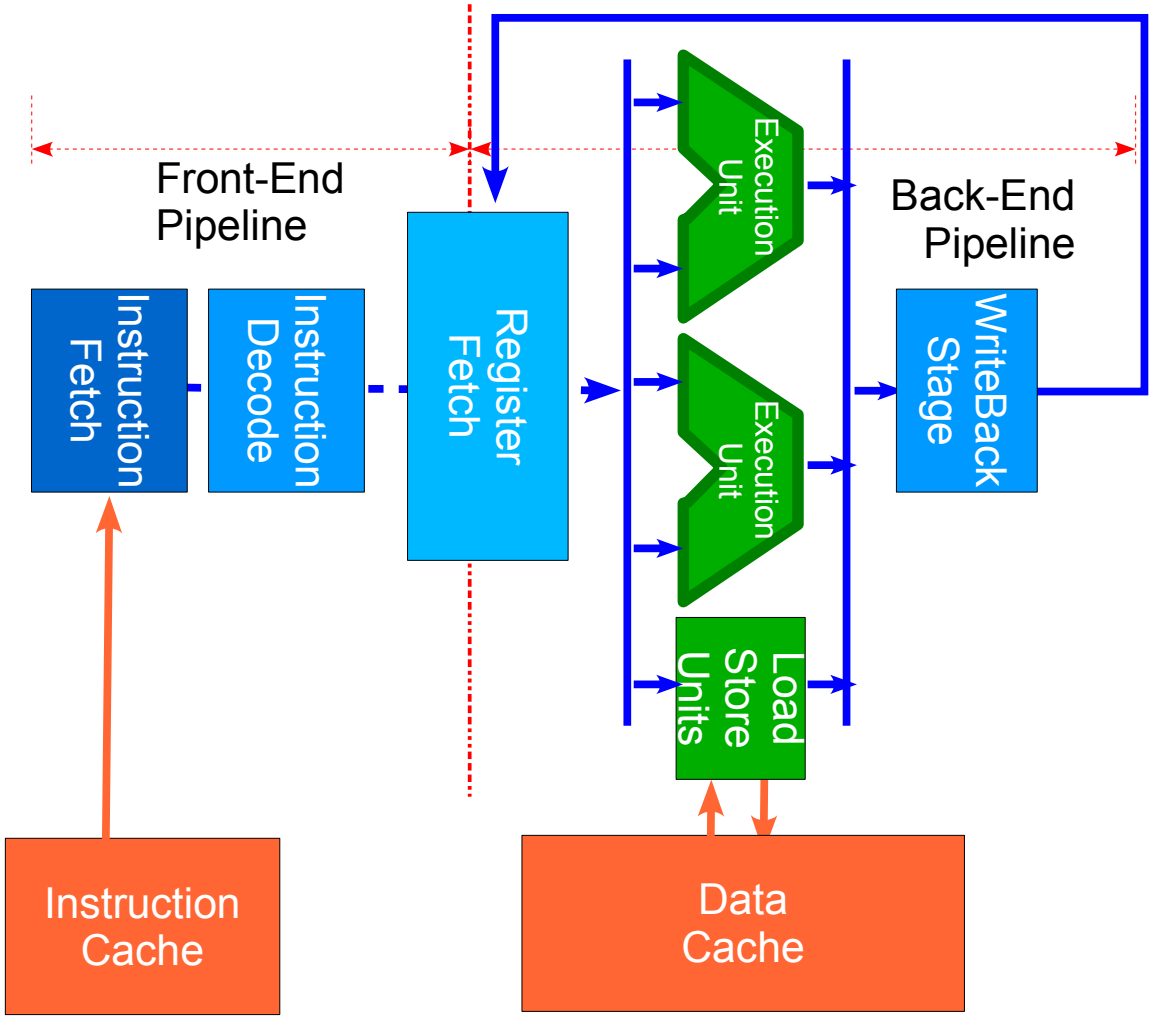
CPU vs. GPU

- Traditional Single-CPU Components



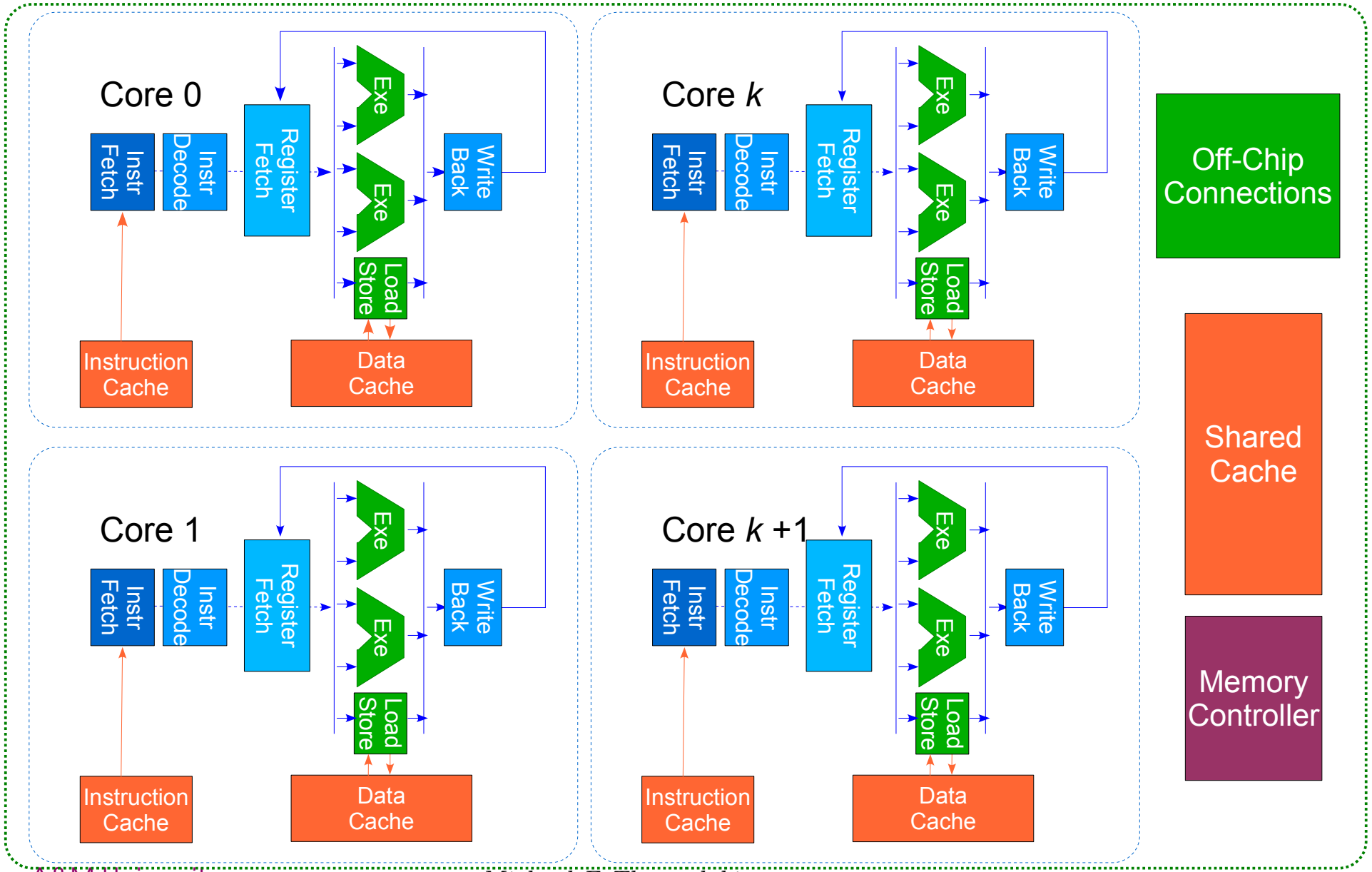
Simplified Super-Scalar CPU Pipeline

- Super-scalar CPU pipeline stages
  - instr fetch
  - instr decode
  - register fetch
  - execution, load/store
    - out-of-order execution
    - usually 5 – 10 instr on the execution units
    - 100s of instructions in progress at a time
    - limited by amount of available ILP
  - write-back



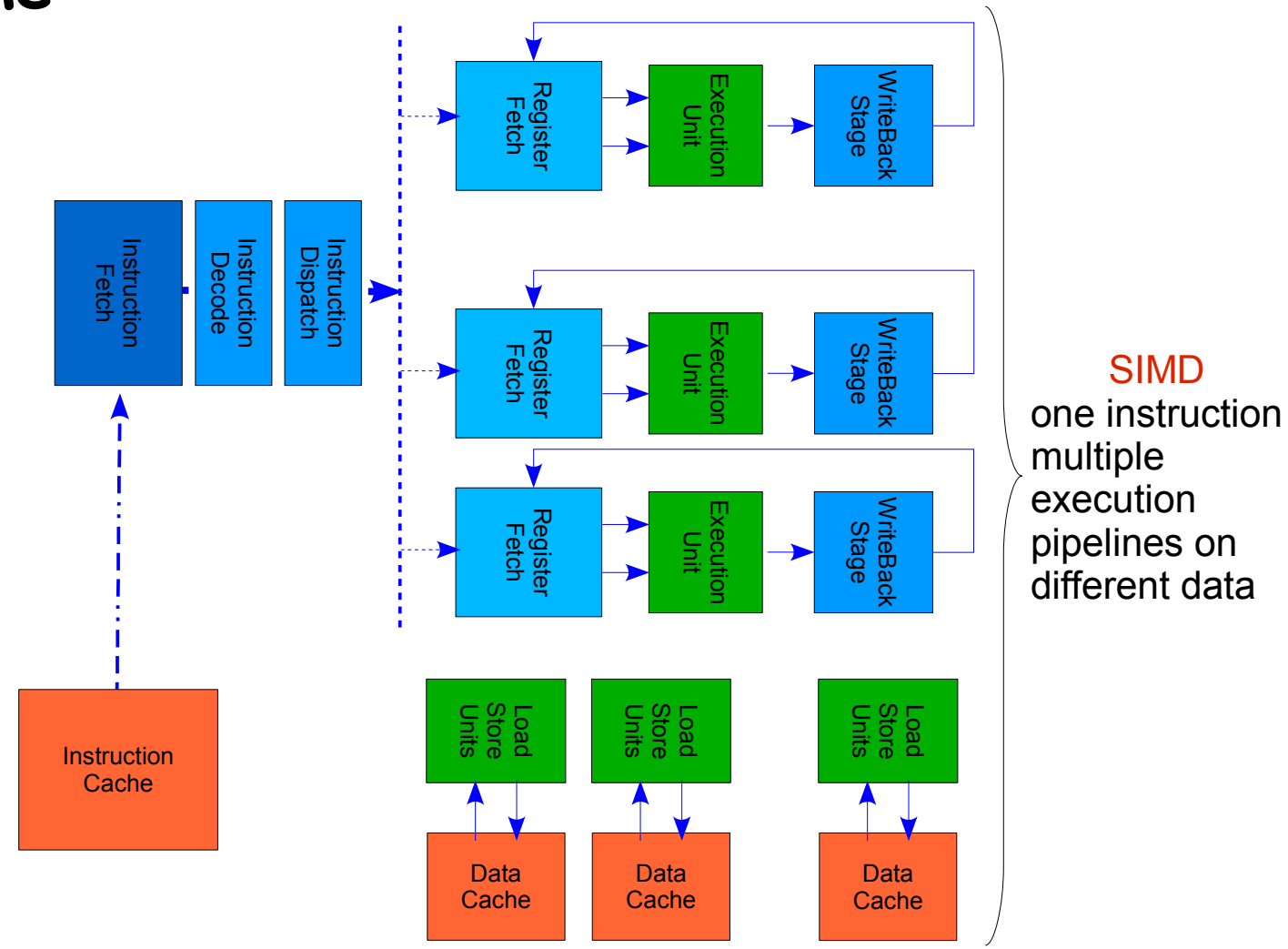
Simplified Multi-Core CPU

- **Multi-core CPU**



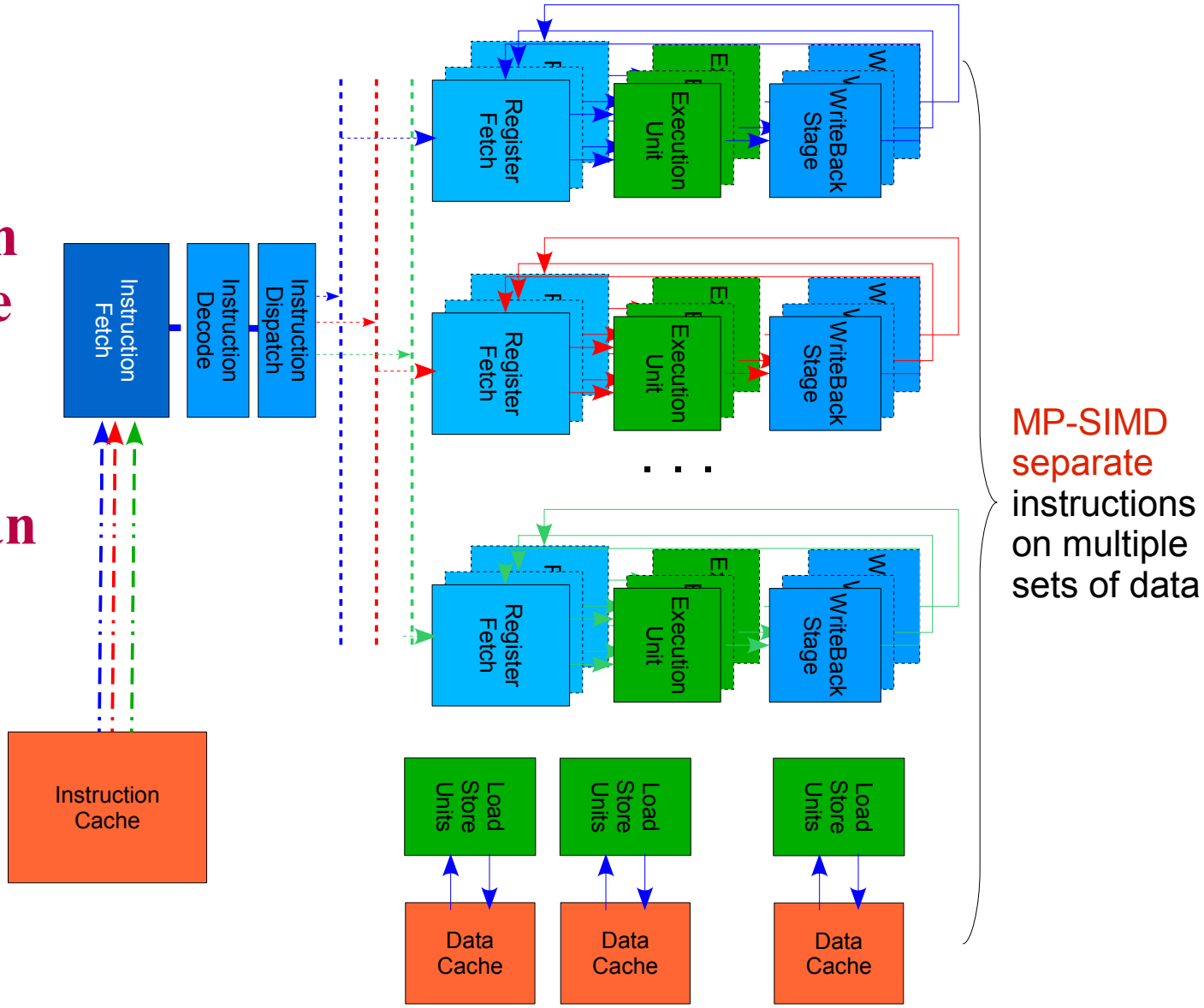
Simplified SIMD GPU Pipeline

- SIMD GPU pipeline stages
  - all execution units perform same operation on different data
  - instr fetch
  - instr decode / dispatch
  - multi-register fetch
  - multi-execution, multi-load/store
  - multi-write-back



Simplified Multi-SIMD GPU Pipeline

- MP-SIMD GPU pipeline stages
  - subsets of execution units perform same operation on different data
  - different subsets can execute different instructions



GPU Processing Concepts

- GPUs are specialized units with massive numbers of integer and FP ALUs and very wide DRAM paths
- Mostly SIMT execution model
  - *single* instruction *multiple* threads
  - this is a form of Single Instruction Multiple Data (SIMD) architecture
- Hardware light-weight thread management
  - h/w thread launch and monitoring
  - h/w thread switching
  - 10s of 1000s of lightweight, concurrent threads
  - each thread: PC, private registers, …
- Multiple memory scopes
  - per-thread private memory
  - per-thread-block shared memory
  - global device memory
- Using threads to hide memory latency
- Coarse grain thread synchronization

GPUs Beyond Pure Graphical Workloads

- Effort underway since 2003
- Initial approach: GP-GPU
  - high-level "shading" languages (OpenGL, DirectX, Cg, *etc.*) used to port non-graphical problems to use GPU h/w
    - protein folding
    - stock options pricing
    - SQL queries
    - MRI reconstruction, etc.
  - significant performance speed-ups
  - several drawbacks and inefficiencies lead to
- Need for hybrid computing model to combine non data-parallel and data parallel computation parts

nVidia Unified GPU Architectures for HPC

- Unified Graphics and Compute Architecture
- GPU Computing Approach for HPC (Tesla Platform)
  - provide a unified s/w and h/w environment for the development of data parallel and scalar code, alongside pure graphical code
  - CPU and GPU collaboration
- CUDA s/w and h/w architecture
  - environment enabling programming of GPU h/w by a variety of common (non-graphics specific) high-level languages
    - C/C++, Fortran and other high-level language bindings,
    - CPU and GPU algorithmic components

The CUDA Programming Model

- CUDA is a programming model and environment, targeting
  - nVidia many-core architectures (GPU h/w)
    - wide SIMD parallelism
    - highly scalable GPU h/w
- CUDA provides
  - a *thread* abstraction to assign work to SIMD units
  - hierarchical CUDA thread model
  - synchronization and data sharing between small groups of threads
- CUDA code is written in C/C++ plus extensions
  - can be used from other languages

The CUDA Execution Model

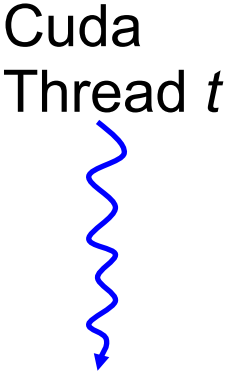
- Parallel kernels consist of many threads
  - all threads execute the same sequential program
- CUDA parallel code ("kernel") is launched and executed on GPU device by many CUDA threads
- Thread launching is hierarchical
  - threads are grouped into blocks
  - blocks are grouped into grids
- Serial code is written for a thread
  - each thread is free to execute a unique code path
  - CUDA provides build-in thread and block ID variables

CUDA Hierarchy of Concurrent Threads

- Parallelism in Cuda is maintained at a 4 level Thread hierarchy
  - a *Stream* is a list of *Grids* that execute in order; (Fermi GPU can execute multiple streams in parallel)
  - a *Grid* is a set of up to 2^{32} *Thread Blocks* executing the same kernel (parallel code)
  - a *Thread Block* is a set of up to 1024 *Cuda Threads*
- Each *Cuda Thread* is an independent, lightweight, *scalar* execution context
  - instruction unit dispatches instructions to each of the SIMD units in h/w
  - groups of 32 Cuda Threads form *Warps* which execute in lockstep SIMD fashion

Cuda Thread (CT)

- Each CT is a *very* lightweight, independent, logically MIMD execution context
  - computation unit dispatched to use a GPU pipeline
  - own control flow, PC, register file, local memory, call stack, *etc*.
  - may access any GPU global memory
  - CTs uniquely identified within a Grid by 5 integers `threadIdx.{x,y,z}, blockIdx.{x,y}`



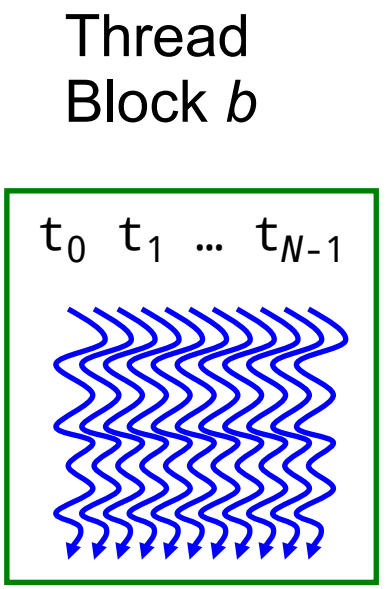
- CTs are meant to have very fine granularity
- Fermi
  - each CT can access up to 21 32-bit registers
  - 1536 CTs share 64KiB "shared memory" / L1 cache
  - GPUs have no by-pass operand networks , hide latency with concurrency

Cuda Warp

- Warp is the logical instruction dispatch width within Cuda GPU for CTs operating in SIMD mode
- Warps consist of up to 32 CTs
  - GPU execution h/w is most efficiently used when all CTs execute in pure SIMD mode: all execute instructions from the same PC
  - CTs in a Warp may diverge, *i.e.*, execute *different* code paths
    - some pipelines go idle
    - predication selects which cores idle, which active
  - aligned contiguous memory accesses are coalesced into single high BW access
  - Warp Id = CTId / 32

Cuda Thread Block

- Thread Block (TB) : a virtualized multi-threaded core
  - number of CTs, registers and shared memory are configured at kernel call time 1-1024 CTs can be in a TB
  - CTs can be organized in 1D, 2D or 3D geometry within TB
  - program should be valid for any interleaving of block executions
  - CTs in a TB execute a data parallel task
  - CTs in a TB synchronize via barriers and communicate via fast on-chip shared memory
- Fermi
  - working set should fit in 128KiB register file and 64KiB L1 cache
  - non-cacheable WS limited by GPU DRAM
  - all CTs in a TB share a small instruction cache



Cuda Grid (CG)

- A CG is a set of Thread Blocks belonging to the same a CUDA kernel
  - a CUDA kernel is the function which executes in the GPU with code performing a related computation
  - a CG can consist of up to 2^{32} TBs
  - CGs can arrange TBs in a 1D or 2D logical geometry
  - TBs have 1D or 2D identification
  - all CTs in a single kernel have the same entry point
  - TBs in a CG may execute any code they want
    switch (blockId.x) { ... } which incurs no additional overhead
  - Scalability requires many TBs / Grid

Cuda Stream (CS)

- A CS is a host initiated sequence of
  - host to GPU memory transfers
  - GPU kernel invocations
  - GPU to host memory transfers
- Multiple kernel calls can be in progress on separate asynchronous streams
- Fermi allows multiple kernel execution on GPU

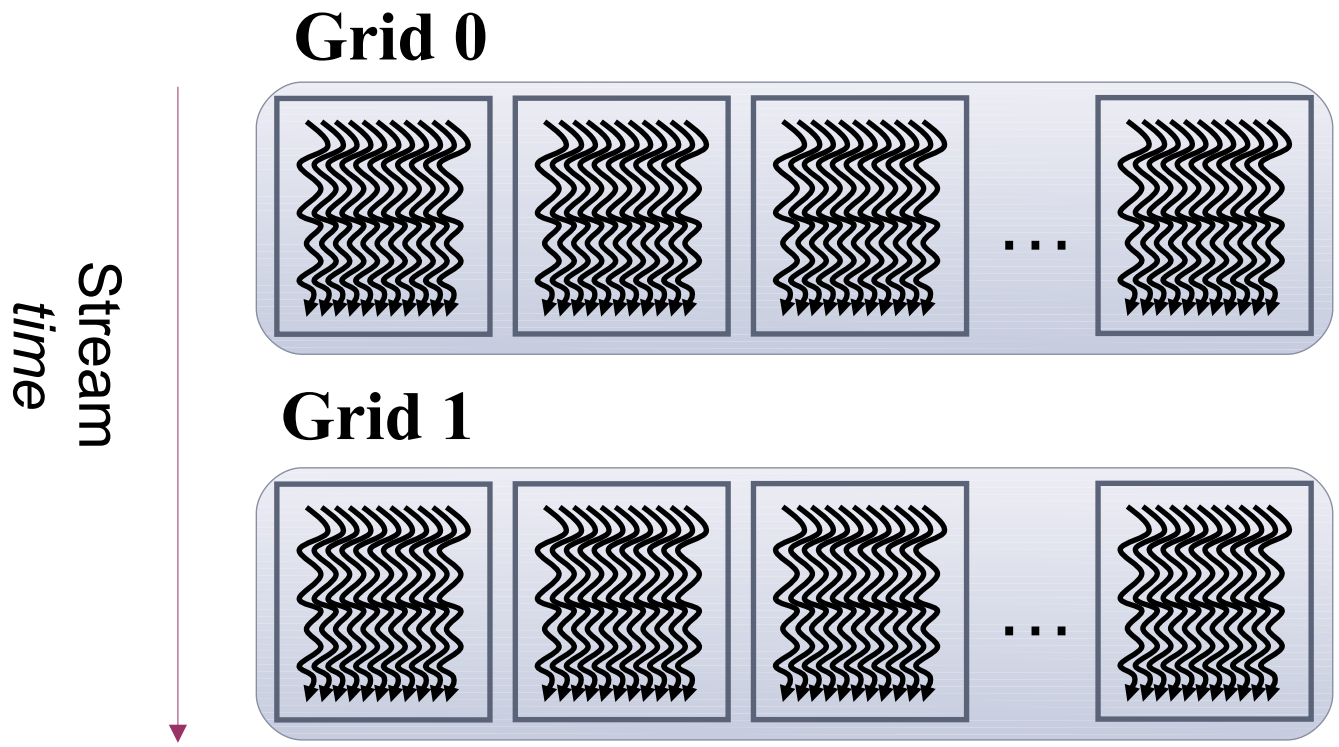
Cuda Stream (CS)

- Multiple streams can be created
  - Example

```
cudaStream_t s0, s1;
cudaStreamCreate(&s0);
cudaStreamCreate(&s1);

cudaMemcpyAsync(a0, cpu_a0, N0*sizeof(float),
   cudaMemcpyHostToDevice,s0);
/* GPU kenel */

vecAdd<<<N0/256,256,0,s0>>> (a0, b0, c0, N0);

cudaMemcpyAsync(a1, cpu_a1, N1* sizeof(float),
   cudaMemcpyHostToDevice,s1);

vecAdd<<<N1/256,256,0,s1>>>(a1,b1,c1,N1);
```

Cuda Thread Hierarchy

- Four levels of
  - *Stream* : a list of Grids executing in-order
  - *Grid* : set of up to 2^{32} TBs within same kernel
  - *Thread Block* : up to 1024 Cuda threads



Thread Synchronization

- CTs within CTBs if necessary use barriers to synchronize
- CTBs coordinate via atomic operations
  - e.g., by incrementing shared queue pointer with atomic operations
- Implicit Barrier between dependent kernels

CTBs Must be Independent

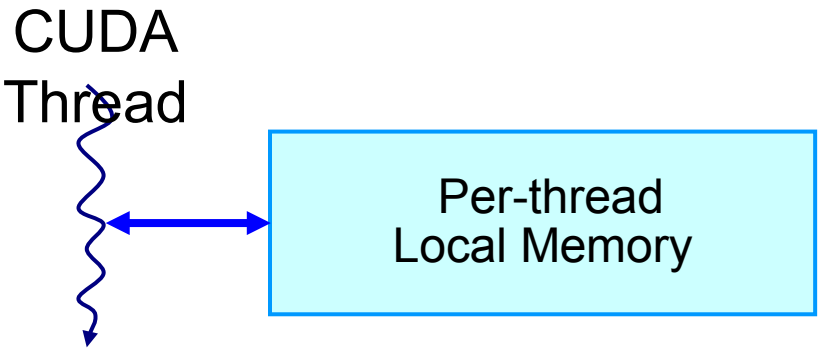
- Any possible interleaving of CTBs should be valid
  - run to completion without pre-emption
  - may run in any order
  - may run concurrently or sequentially
- CTBs may co-ordinate but not synchronize
  - shared queue pointer is OK
  - shared lock is not OK : deadlocks are likely
- Independence is good for scalability

CUDA Parallelism Models

- Thread Parallelism
  - each thread is independent
- Task Parallelism
  - different blocks are independent
  - independent kernels
- Data Parallelism
  - across threads in a CTB
  - across CTBs in a kernel

CUDA Threading Memory Models

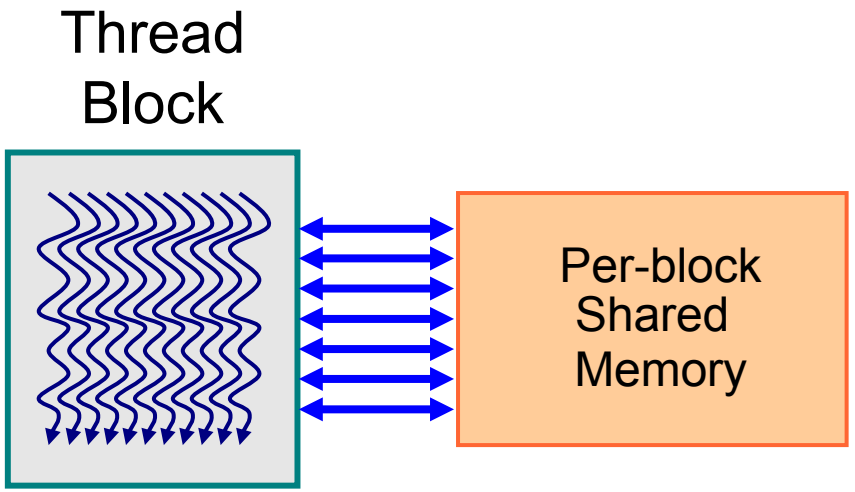
- Each Cuda Thread has private access to local memory
  - set aside at block instantiation
- Fermi
  - 128KiB register file is partitioned among all resident threads
  - Cuda program can trade degree of thread block concurrency for amount of per thread state
  - registers, stack woulf spill into "local" DRAM (cache on Fermi), if necessary, or in to global DRAM




```
/* in kernel */
float Lfloat; double Ldouble; ...
```

CUDA Threading Memory Models

- Each Thread Block has private access to a configurable amount of scratch-pad memory
  - Fermi SM's (discussed later) 64 KiB SRAM can be configured as
    - 16 KiB L1 cache + 48 KiB scratch-pad, or
    - 48 KiB L1 cache + 16 KiB scratch-pad
  - available scratch pad space is partitioned among resident thread blocks, providing another concurrency-state trade off




```
/* in kernel */
__shared__ float Sfloat;
__shared__ double Sdouble; ...
```

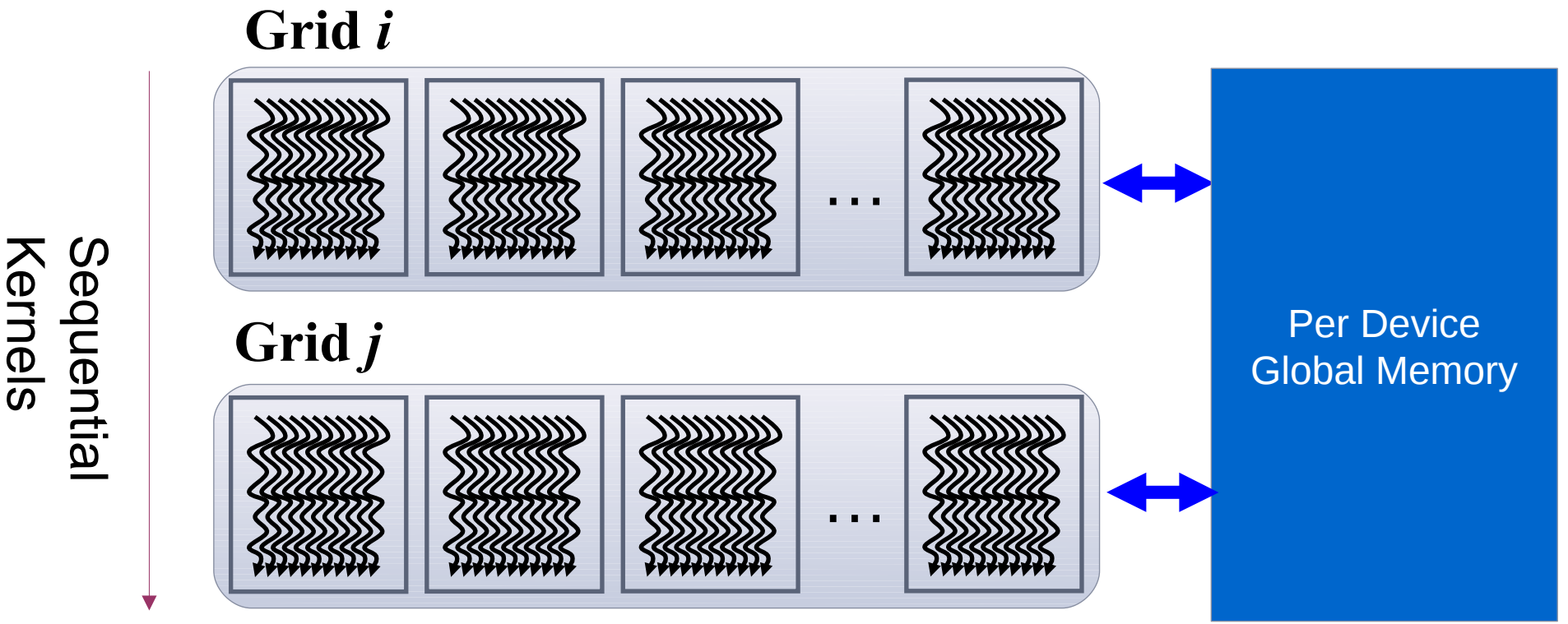
Cuda Global Memory Model (GM)

- Thread blocks in all Grids share access to GM a large pool of "Global", *separate* from the Host CPU's memory.
  - GM holds the application's persistent state, while the thread-local and block-local memories are temporary
- On Fermi, GM is *cached* in a 768 KiB shared L2 cache
  - GM is much more expensive than on-chip memories: *O*(100)x latency, *O*(1/50) x (aggregate) bandwidth

CUDA Memory Model

- TBs in all Grids share access to a large pool of Global memory

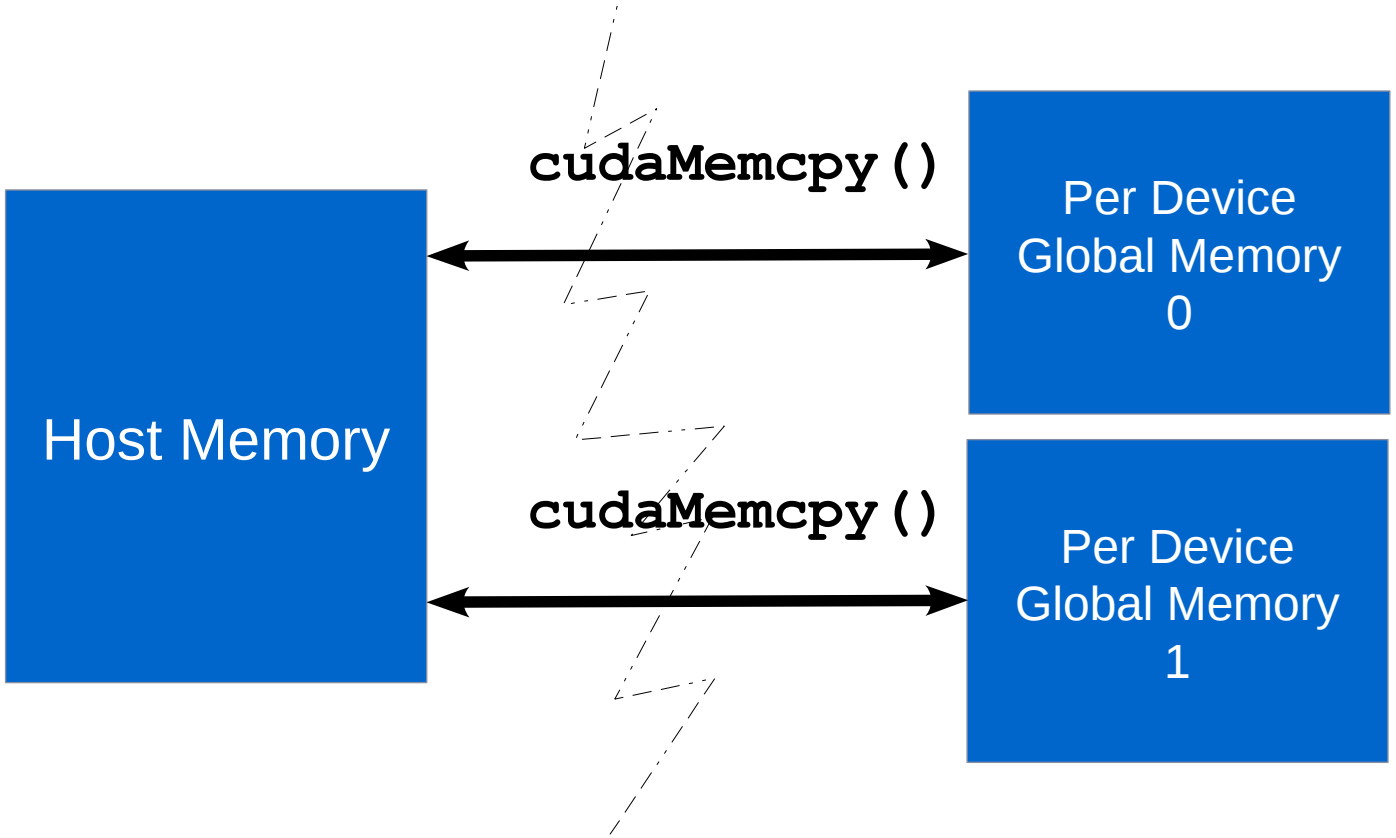
```
/* in kernel */
__device__ float Gfloat;
__device__ double Gdouble; ...
```



CUDA Memory Model

- Each Cuda device has its own Global memory *separate* from host's CPU memory
  - allocated/freed via `cudaMalloc()` / `cudaFree()`
- Host to device memory transfers via cudaMemcpy() over PCIe bus, low bandwidth/high latency
- Multiple GPU devices managed by multiple CPU threads

CUDA System Memory Model



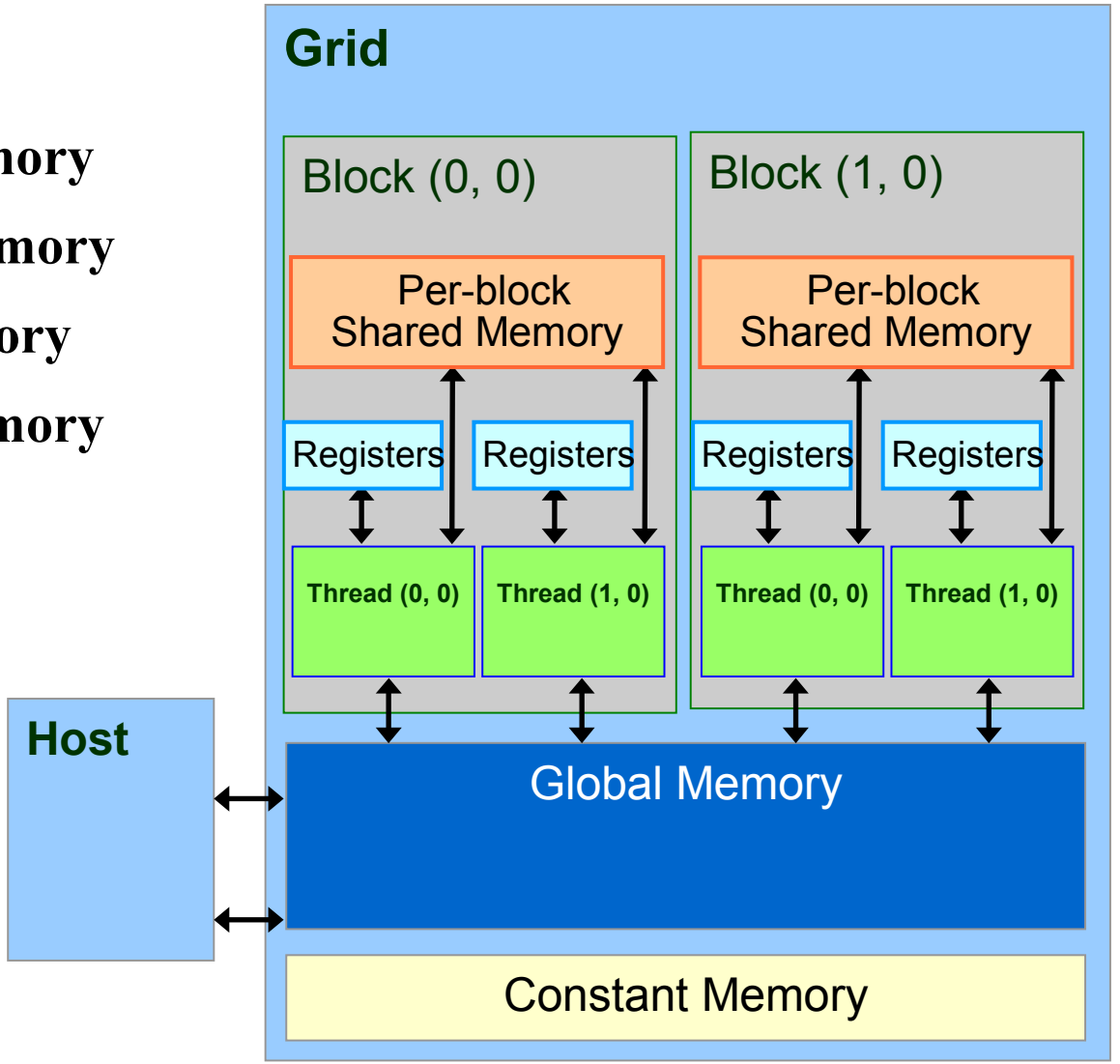
CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

Logical View of CUDA Memories

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read/only per-grid constant memory

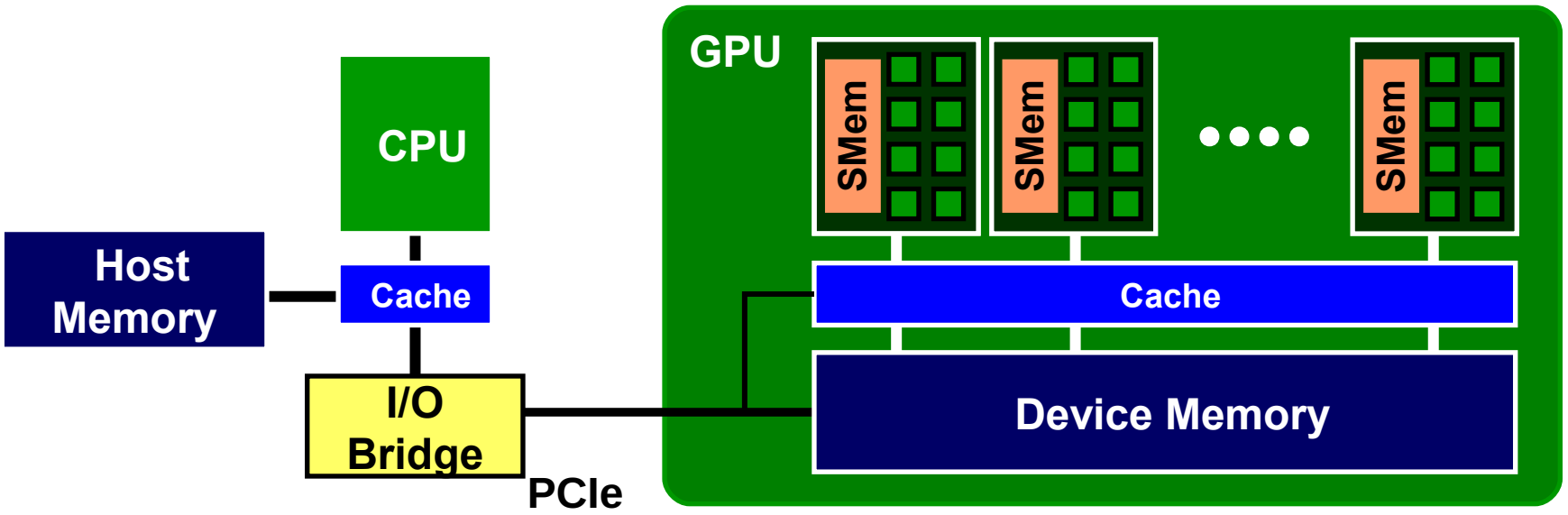


Outline of CUDA Basics

- Basic Kernels and Execution on GPU
- Basic Memory Management
- Coordinating CPU and GPU Execution

Using CPU+GPU Architecture

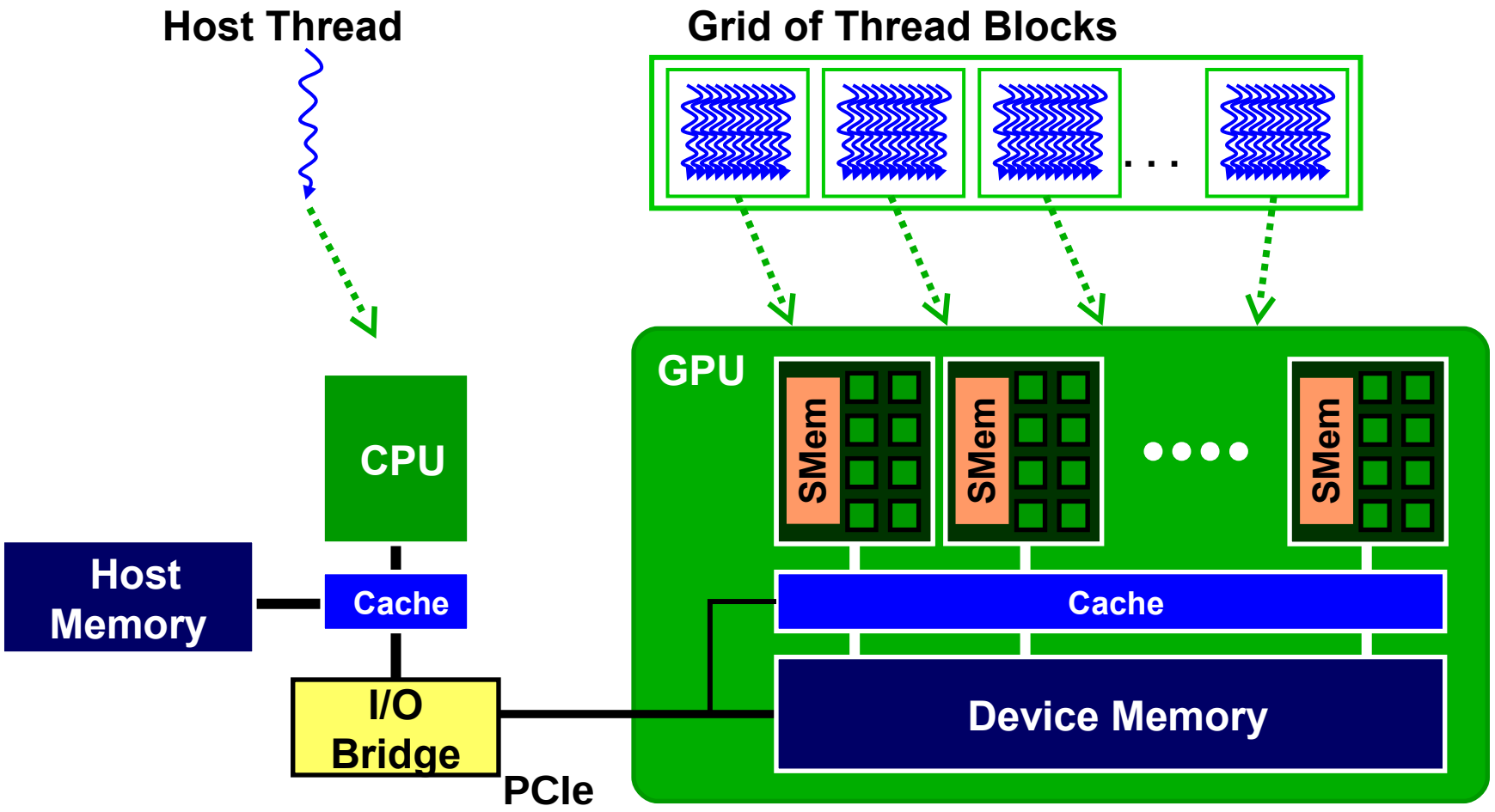
- Heterogeneous system architecture
- Use the right processor and memory for each task
- CPU excels at executing a few serial threads
  - Fast sequential execution
  - Low latency cached memory access
- GPU excels at executing many parallel threads
  - Scalable parallel execution
  - High bandwidth parallel memory access



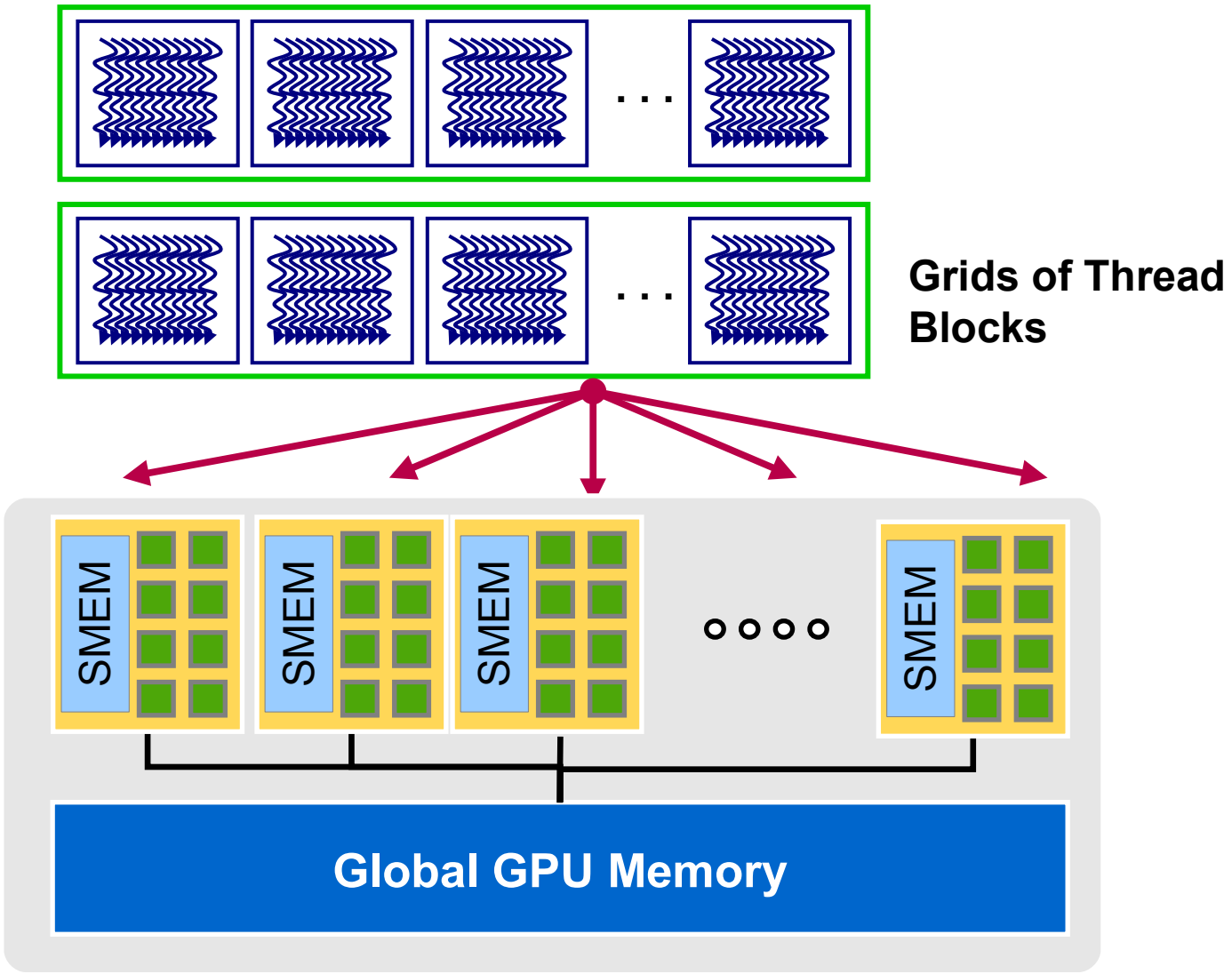
CUDA kernel maps to Grid of Blocks

- **Work and Data Flow**

```
kernel_func<<<nblk, nthread>>>(param, … );
```



Grids Run on GPU



Small Example

- Cuda "Hello World"
  - vector addition

Example: Vector Addition Kernel

Device Code

```
// 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];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

```
// 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];
}
```

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

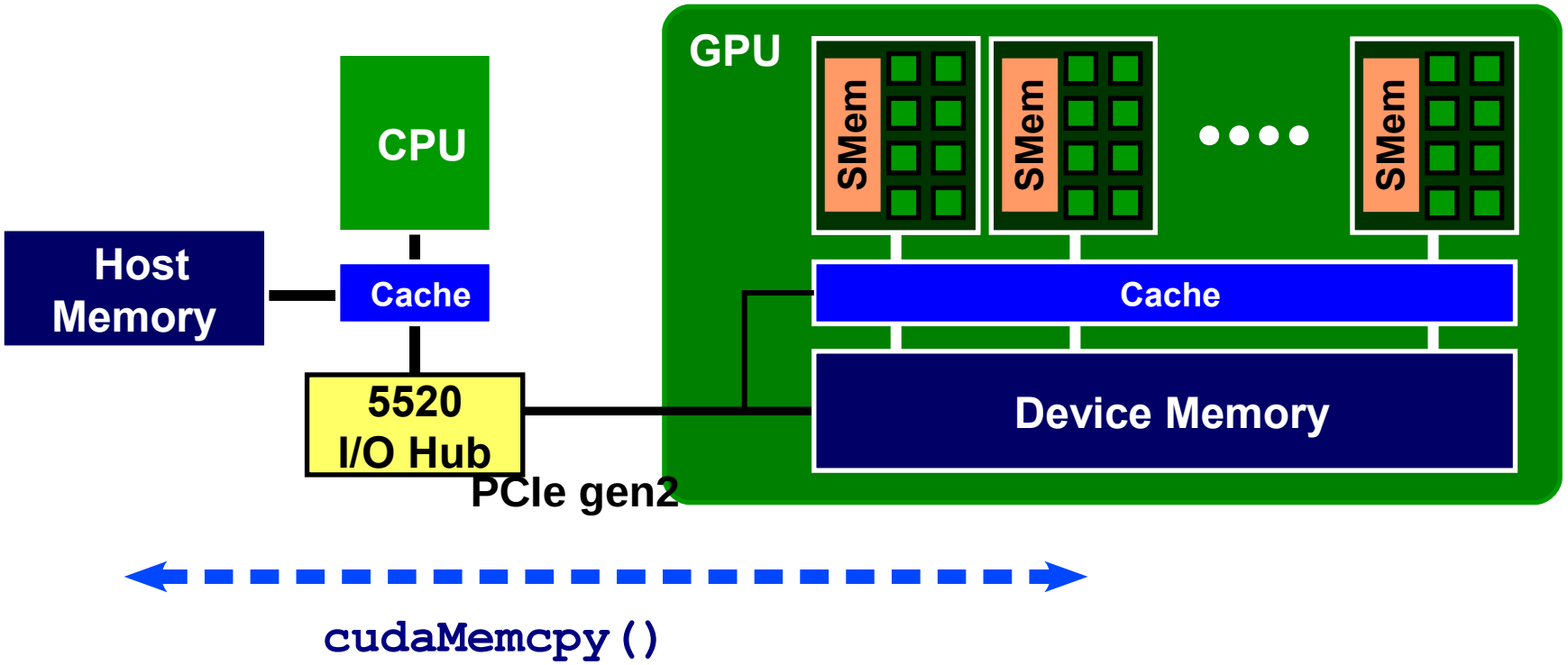
Example: Host code for vecAdd

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …; *h_C = …(empty)

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute grid of N/256 blocks of 256 threads each

      vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Example: Host code for vecAdd (2)

```
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// copy result back to host memory
cudaMemcpy( h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost) );

// do something with the result…

// free device (GPU) memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

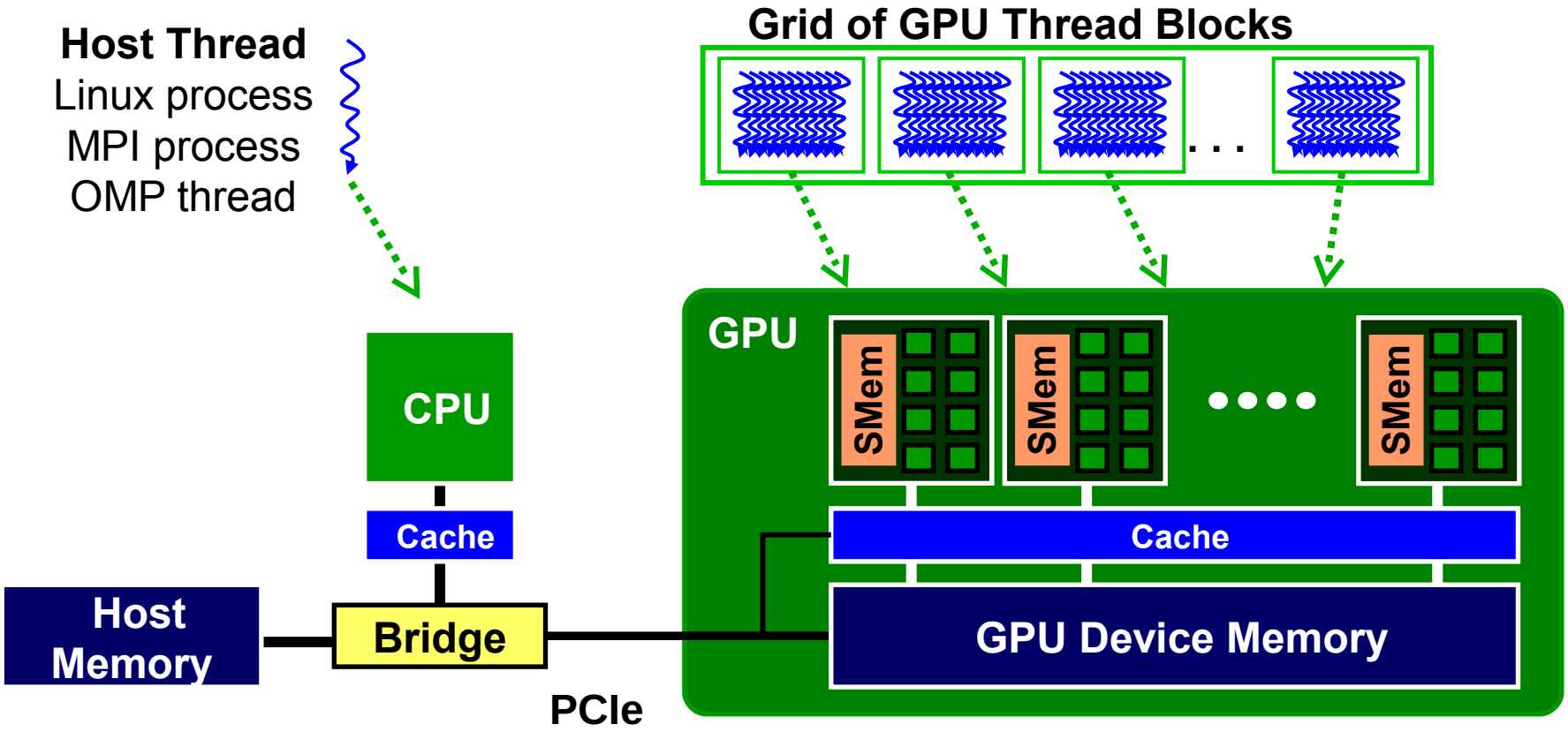
Using cudaMemCpy()



- cudaMemcpy() invokes a DMA copy engine
- Use data as long as possible in a given place
- PCIe gen2 per direction bandwidth = 8 GB/s *peak*, ~6GB/s *in practice*
  - minimize the number of copies
- GPU load/store DRAM peak bandwidth = 150 GB/s

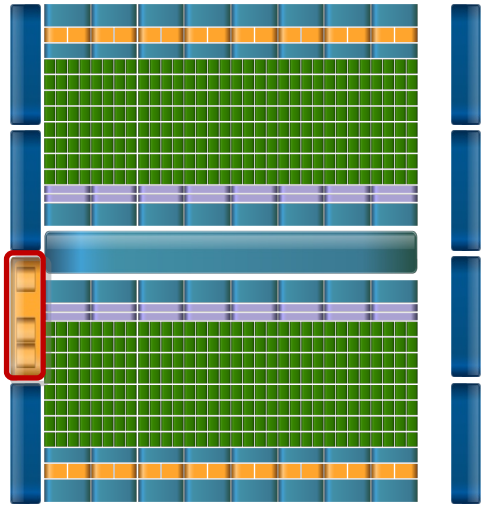
CUDA Parallelism

- Cuda Virtal Resources
  - CT <-> a virtual scalar processor
  - TB <-> a virtual multiprocessor
  - scheduled on actual GPU h/w without pre-emption and run to completion

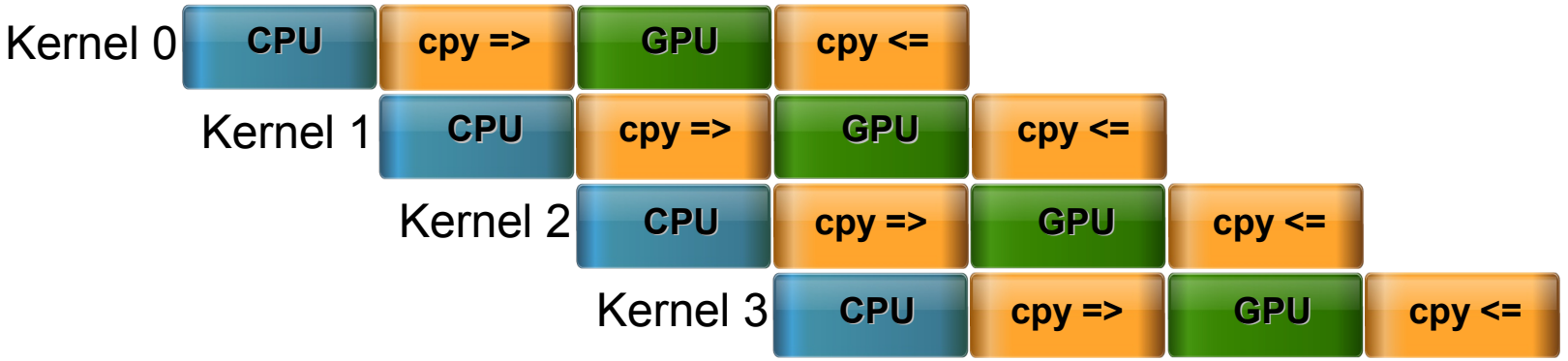


Overlap GPU Compution and CPU⇔GPU transfers

- cudaMemcpy() invokes data transfer engines
  - CPU→GPU and GPU→CPU data transfers
  - Overlap with CPU and GPU processing

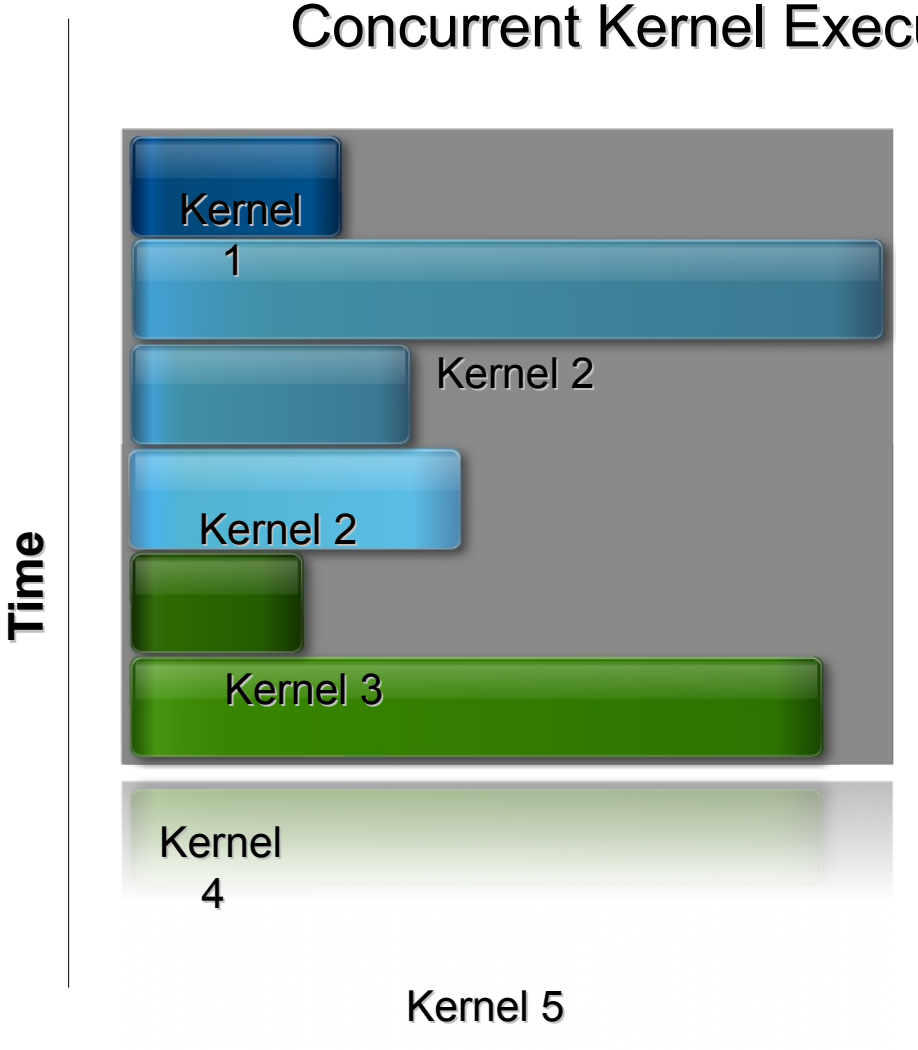


- Pipeline Snapshot:

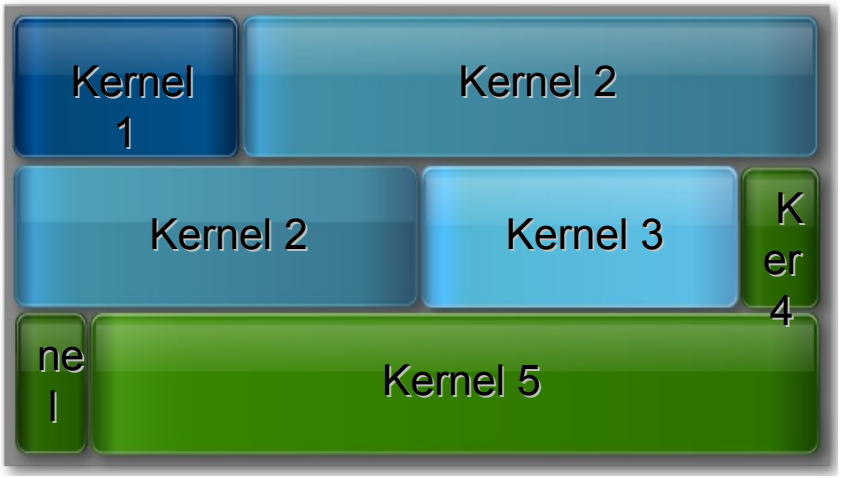


Fermi Runs Independent kernels in Parallel

Concurrent Kernel Execution + Faster Context Switch



Serial Kernel Execution



Parallel Kernel Execution

nVidia GPU micro-Architectures

- nVidia introduced 3 generations of micro-architectures for GPU computing
  - nVidia G80
  - nVidia GT200
  - nVidia GF100 “Fermi”

G80 – First Generation GPU Chip

- G80 GPU Introduced in Nov 2006
  - GeForce 8800, Quadro 8800, Quadro FX5600 and Tesla C870
  - supports C
  - 128 Streaming Processor Cores (SPCs)
  - replaced *vertex* and *pixel* pipelines with single *unified* processor
    - vertex
    - geometry, and
    - pixel
  - scalar thread processor (no need to program vector registers)
  - Single Instruction Multiple-Thread (SIMT) execution model where multiple independent threads execute concurrently with a single instruction
  - shared memory and barrier support for inter-thread communication

GT200 – Second Generation GPU Chip

- GT200 2nd Generation Unified GPU Architecture (June 2008)
  - major revision of G80
  - introduced in GeForce GTX 280, Quadro FX5800 and Tesla T10
  - 240 CUDA Cores (CCs)
  - processor register files 2×G80 → more threads can execute on-chip
  - h/w memory access coalescing
  - double-precision floating-point arithmetic for scientific and high-performance computing

"Fermi" – Third Generation GPU Computing

- **Fermi 3nd Generation Unified GPU micro-Architecture (2010)**
  - **implemented in GF100 chip for GPU**
  - **used in Ge Force, Quadro FX and Tesla platforms**
- **Major revision of G80 and GT200 GPUs**
  - ***improved double precision*** **performance, while single precision floating point performance was on the order of ten times the performance of desktop CPUs**
  - ***ECC support to*** **protected from memory errors**
  - ***more Shared Memory*** **above 16 KiB of SM**
  - ***faster context switching*** **between application processes and faster compute and graphics inter-operation**
  - ***faster atomic operations*** **(read-modify-write operations)**

nVidia Platforms based on GF100

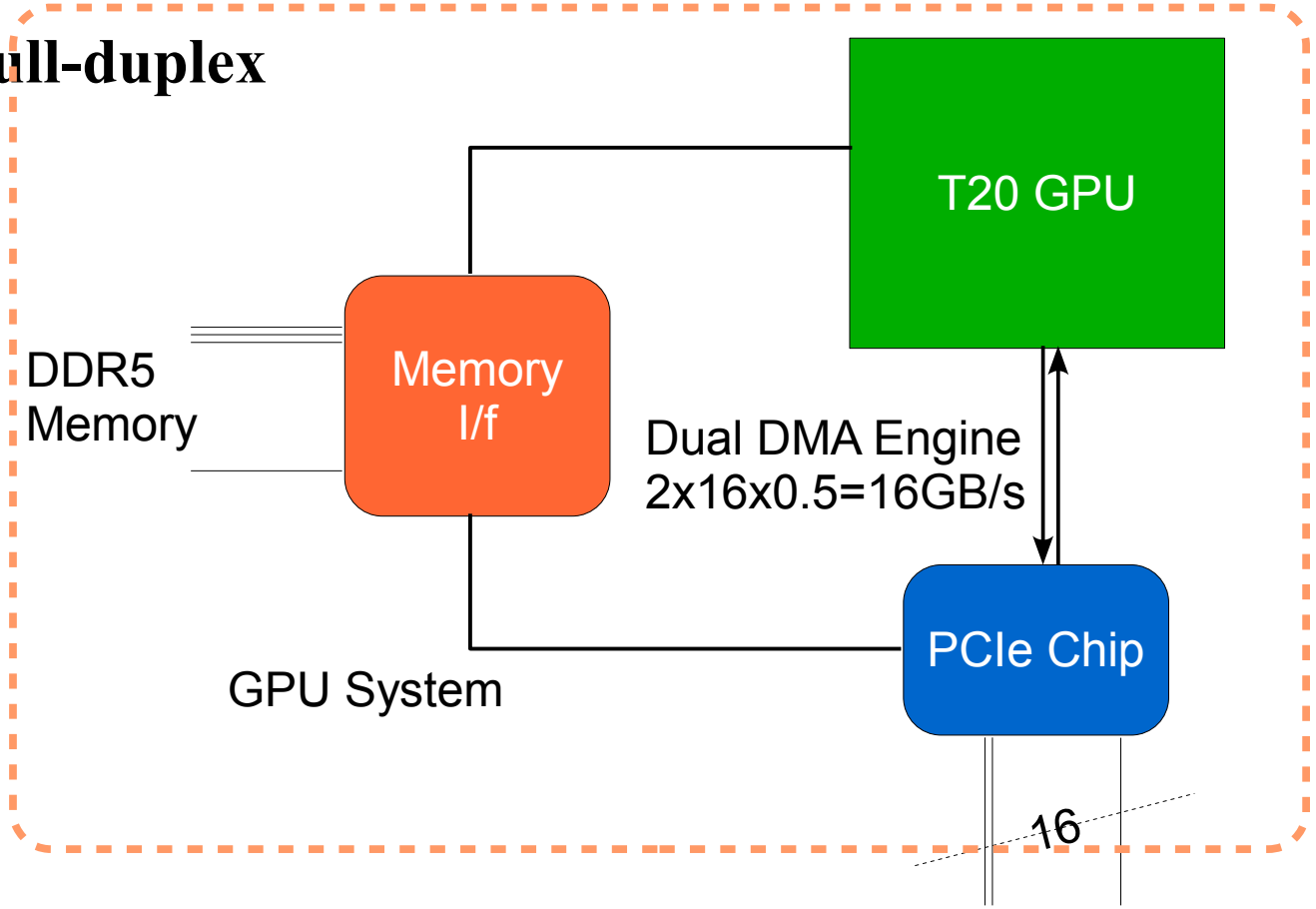
- GeForce GTX480 – Graphics, CAD, visualization
- Quadro – Image, Audio and Video processing
- Tesla – HPC computing

nVidia Tesla HPC Platform

- Tesla C2050/C2070 – For HPC Workstation (3/6GiB DDR5)
- Tesla S2050 – standalone GPU compute servers, (up to 4 GPUs / unit)
- Tesla M2050/M2070/M2070Q – GPU computing nodes in a cluster (3/6GiB DDR5)

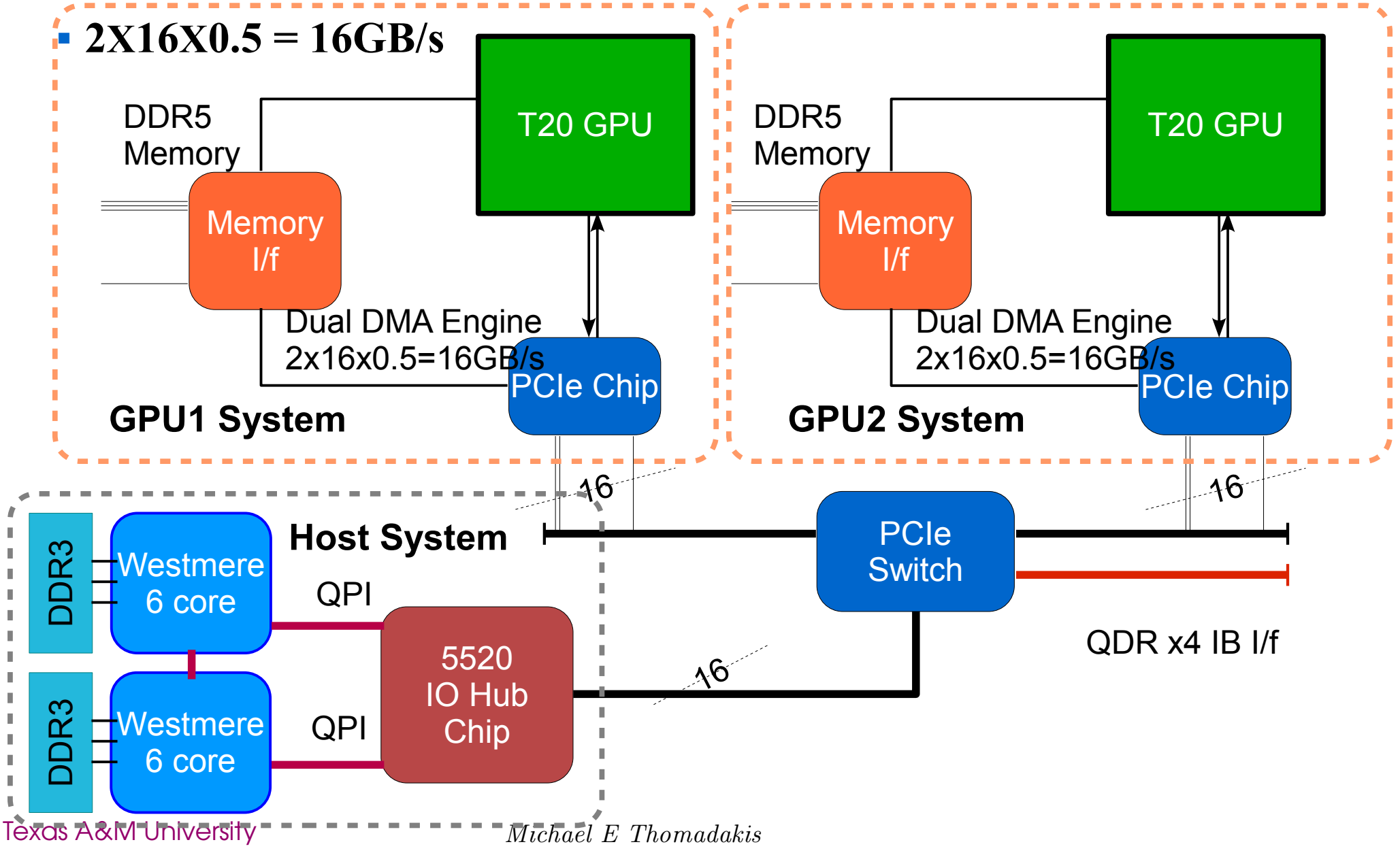
Tesla M2050/M2070 System Architecture

- T20 GPU (FX100) chip attaches to off-chip DDR5 memory
  - 6 X 64 bits = 384 bits data width
- Communicates to host via PCIe x16 gen2 host I/f
  - 2X16X0.5 = 16GB/s full-duplex



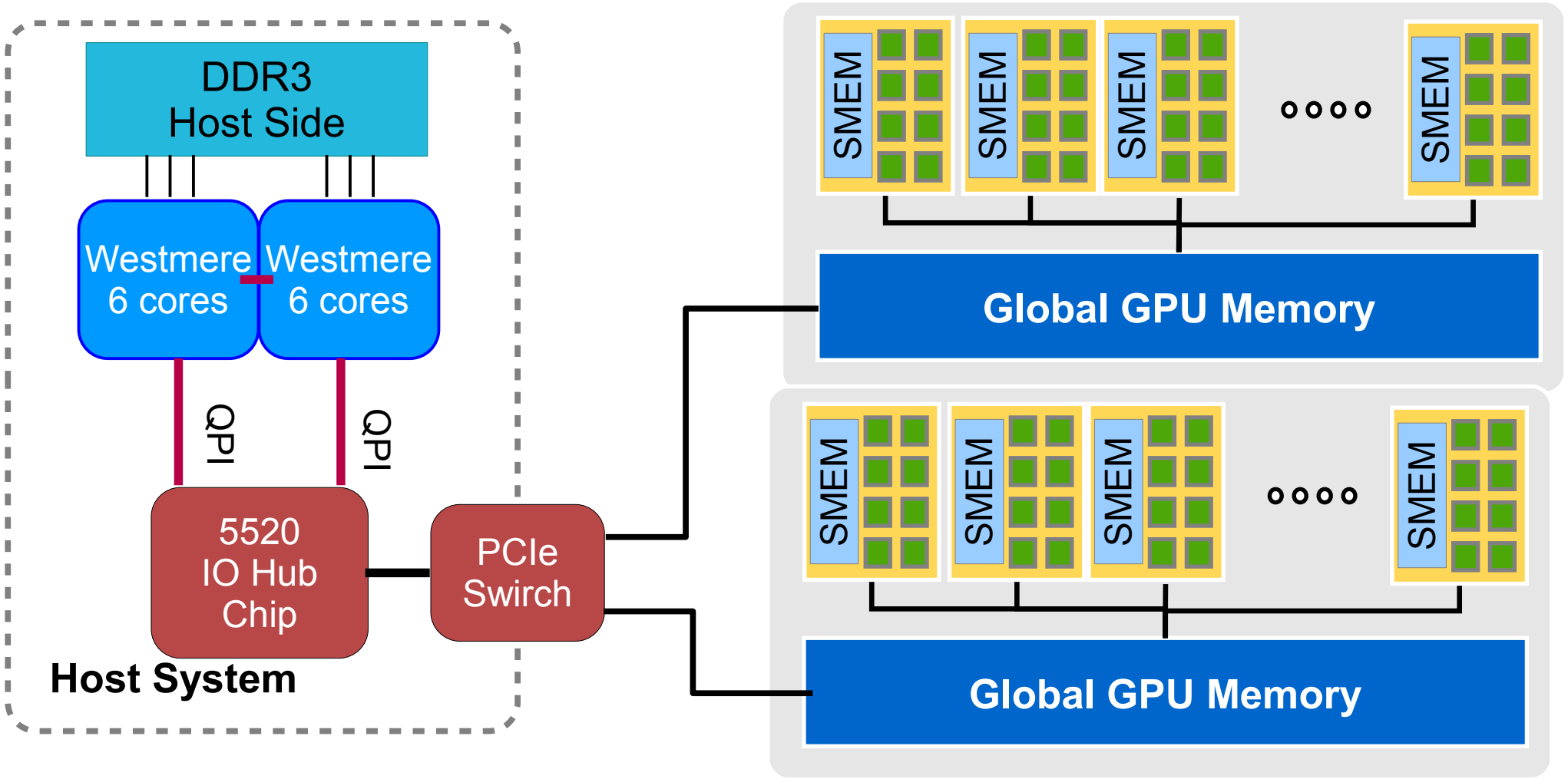
Tesla M2050/M2070 Host System

- Communicates to host via PCIe x16 gen2 host I/f



Connectivity of Host CPU and GPU Memories

- Host CPU copies to host DRAM data
- GPU copies data to GPU memory and vice versa

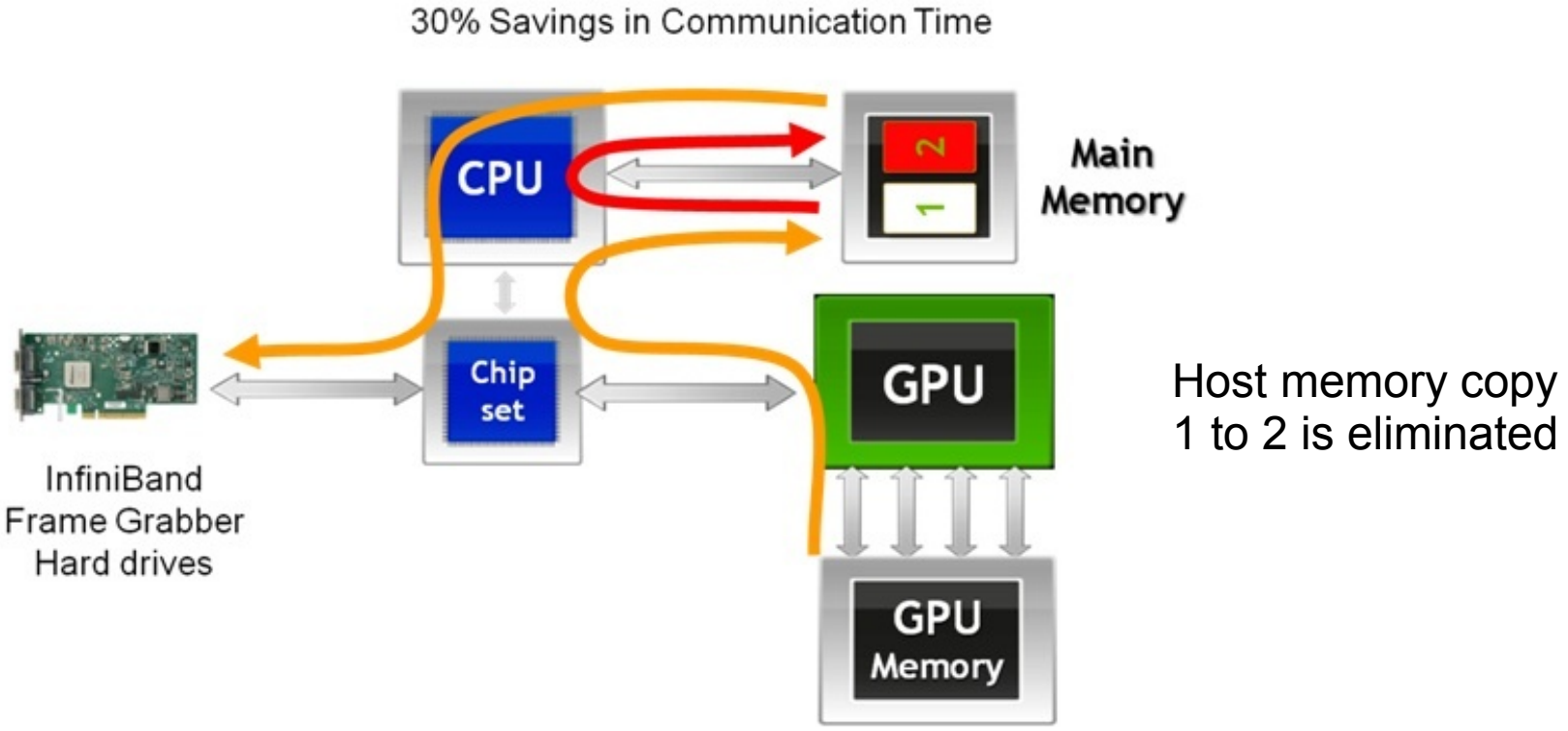


Tesla M2050/M2070 in Clusters

- **GPU Cluster Management**
  - **Bright Computing, ClusteCorp Rocks, Platform Computing**
  - **CUDA OS drivers support**
    - ***exclusive mode* : enables an application to get exclusive access to a GPU**
    - ***GPU visible devices* : enables the cluster management software to manage GPU resources by controlling which GPUs an application can use**

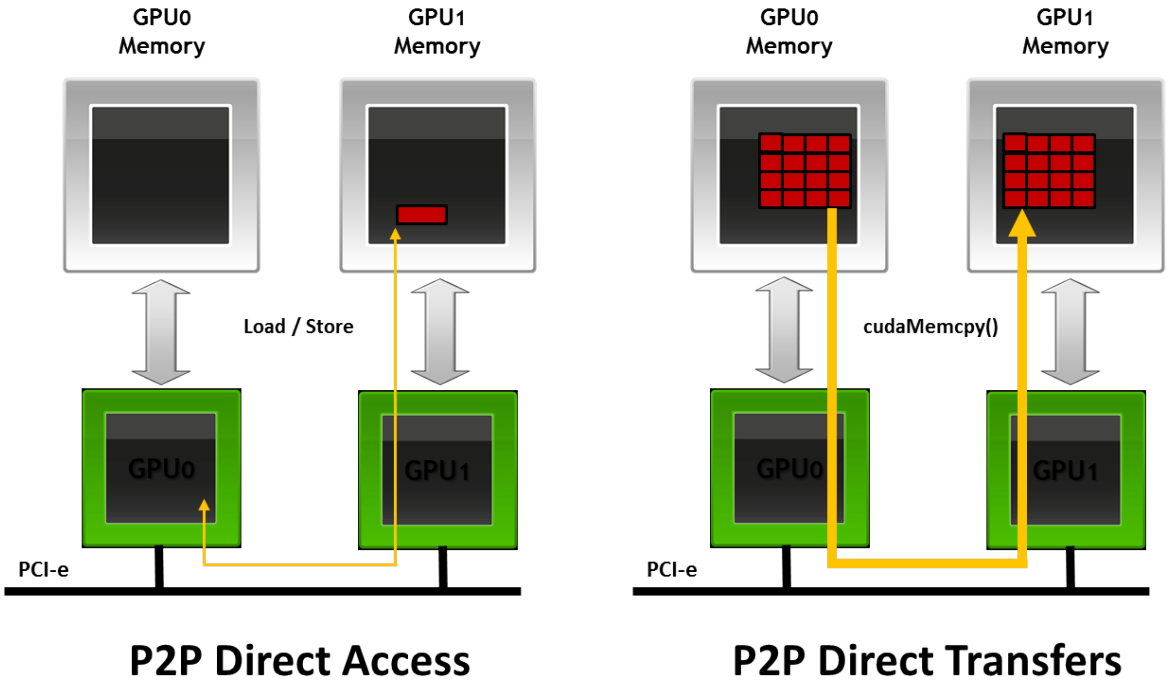
GPUDIRECT™ on Tesla Clusters

- **nVidia GPUDIRECT**
  - **NVIDIA GPUDirect™** is a technology enabling faster communication between the GPU and other devices on the PCIe bus by removing the overhead due to the CPU.
  - **Mellanox InfiniBand** is a the first device nVidia worked with to make communication between the GPU and InfiniBand faster with fewer memory copies in the main memory.



GPUDIRECT™v2.0 on Tesla Clusters

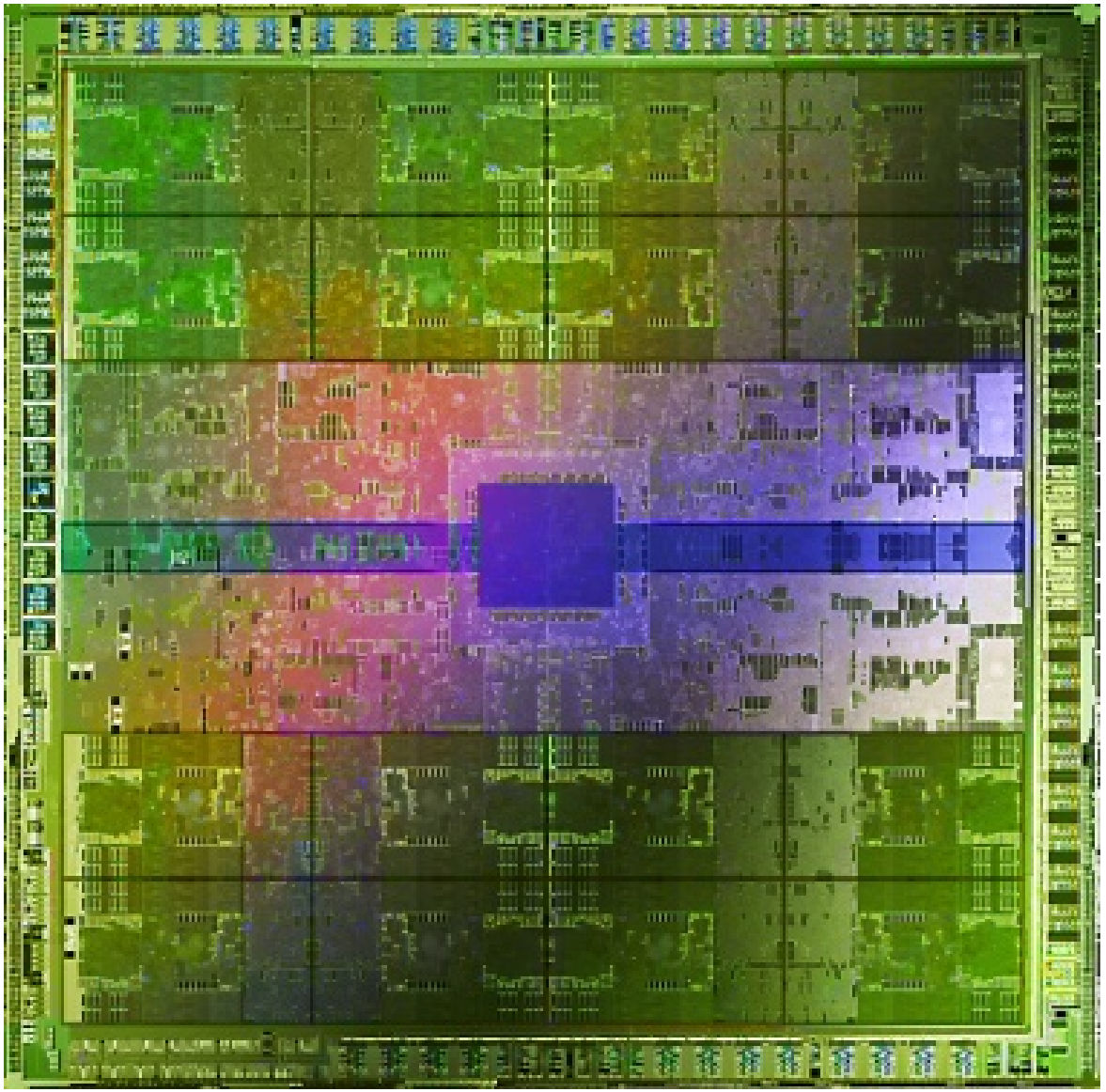
- **nVidia GPUDIRECT v2.0**
  - **NVIDIA GPUDirect™** v2.0 provides technology to enable Peer-to-Peer communication between GPUs on the same PCIe.
  - direct access (load/store) from a GPU to other GPU's memory
  - direct data copy across different GPUs' memories
  - foundation technology to allow direct communication among heterogeneous PCIe devices and GPUs



Intermediate copies to host memory eliminated

GF100 micro-architecture for "Fermi"

- **GF100 Chip Features**
  - 3 billion transistors in 40 nano-meter techonology
  - 42.5 mm X 42.mm
  - processor clock 1.15 GHz
  - 6 memory controllers, 64bit each, 384 bits total
  - memory interface GDDR5
    - 1.546 GHz (3GiB)
    - 1.566 GHz (6GiB)
    - 146 GB/sec
  - Host I/f PCIe x16 gen2



nVidia G80 vs. GT200 vs. Fermi

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

Fermi GF100 Chip Specification

- 16 Third Generation Streaming Multiprocessors (SMs)
  - 4 Graphical Processing Clusters (GPCs)
  - Up to 512 CUDA / unified "shader" "cores"
- Improved Memory Subsystem
- 384-bit GDDR5 memory interface, up to 6GiB
- ECC (register files, L1/L2 cache, "shared memory" and DRAM)
- Second Generation Parallel Thread Execution ISA
- NVIDIA GigaThread(TM) Engine
- PCIe x8 or x16 gen2 i/f for bi-directional transfers to host memory

Fermi GF100 Tesla2050/2070 Main Features

- 448 CUDA "cores" 1.15GHz core processor clock
  - 515 GFLOP/s double precision peak
  - 1030 GFLOP/s single precision peak
- DDR5 DRAM, 384 bit memory i/f, 148GB/s
  - 3 GBs at 1.546GHz, 2.62 GBs when ECC enabled
  - 6 GBs at 1.566GHz, 5.25 GBs when ECC enabled
- Asynchronous bi-directional DMA transfers over PCIe

Fermi/GF100 Chip Block Diagram

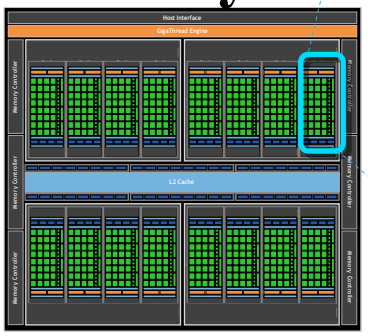
- 4 Graphical Processing Clusters
- 16 Stream Multiprocessors
  - 4 SMs / GPC
  - 32 CUDA cores / SM
  - 512 max
- 6 memory controllers, 64b each
- Memory Hierarchy
  - L1/L2 caches
- Giga-thread engine
- Host I/f

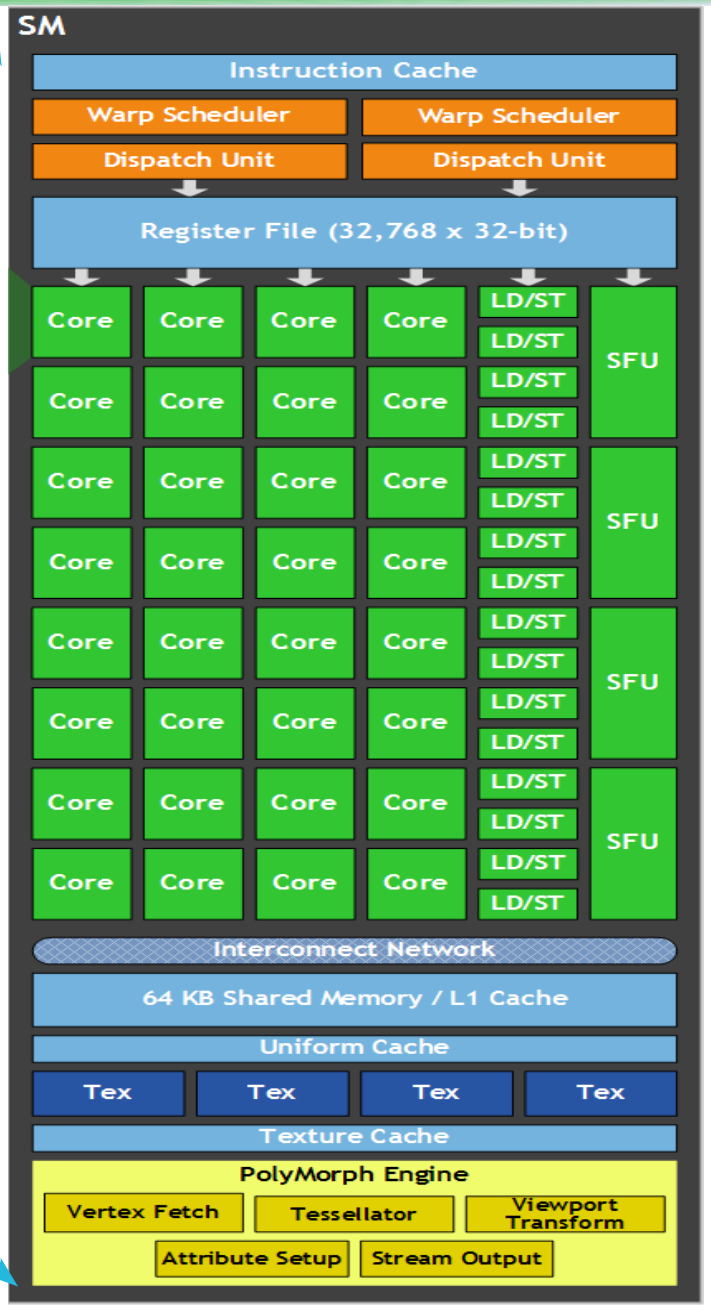
GF100 Stream Multiprocessor Architecture

- 3rd Generation Streaming Multiprocessor (SM)
  - essentially 2way SMT X16-way SIMD pipelines
- 32 CUDA cores (execution pipelines)
- 48 or 16 KiB *shared memory*
- 16 or 48 KiB L1 *cache memory*
- WARP scheduler
- Instruction Set Architecture Improvements for more general computing workloads
- 4 texture units, texture cache (graphics)
- 1 PolyMorphic engine (graphics)

Fermi 3rd Generation Stream Multi-Processor

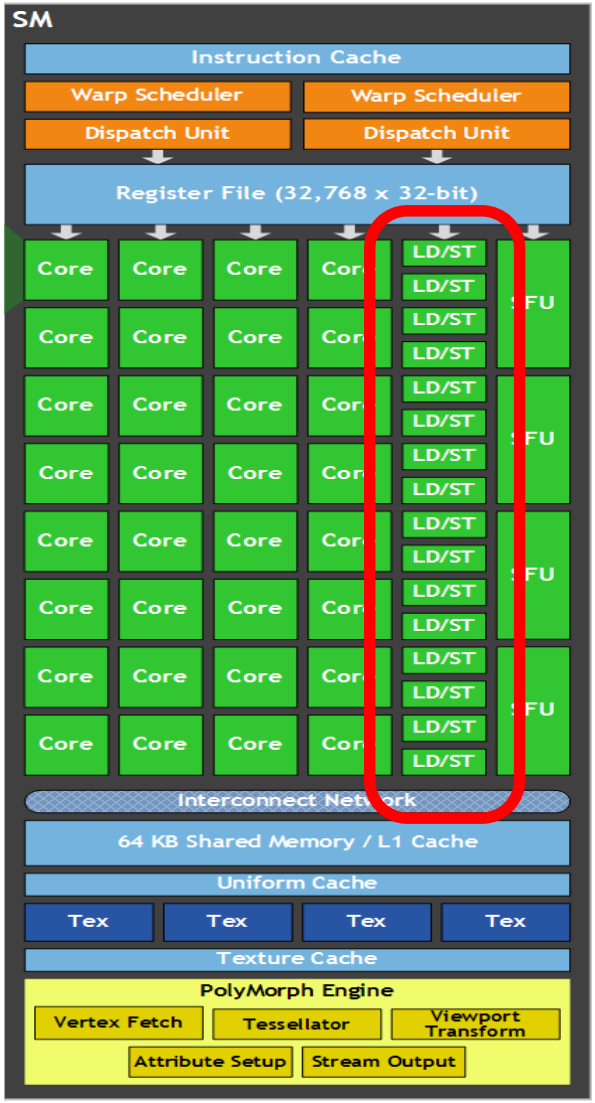
- Stream Multiprocessor
  - 4x8=32 “cores” / SM
- Register File for Cores
  - 32768 x 32bits (128KiB)
- 16 Load/Store Units
- 4 special Function Units
- Dual Warp Scheduler
- Memory
  - 64KiB L1 and Shared Memory
- Giga-thread Engine





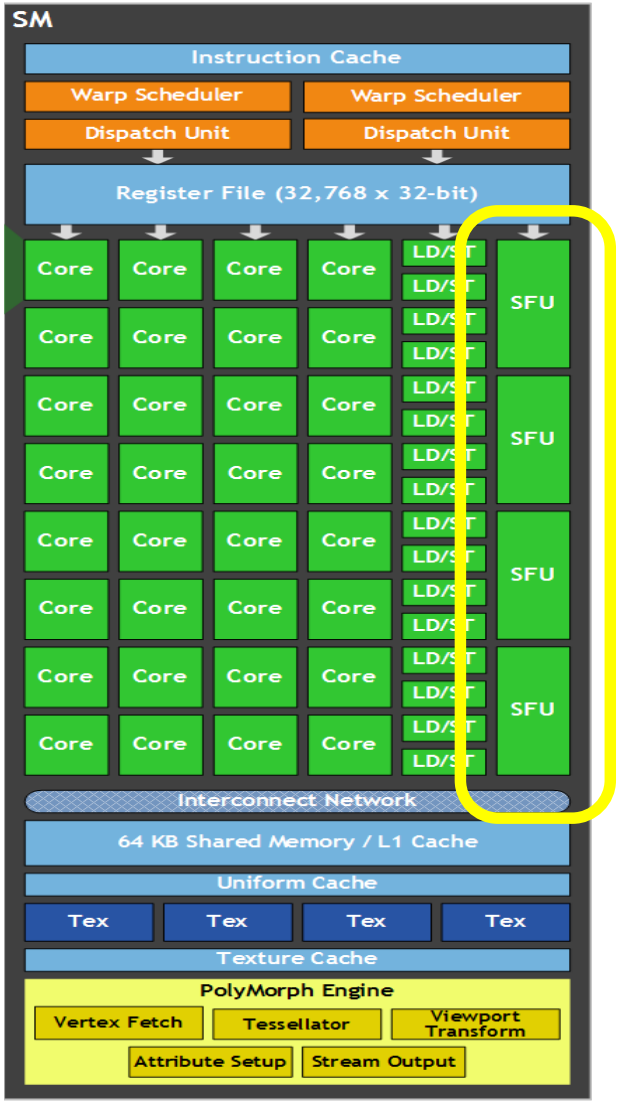
SM Load and Store Units

- Load and Store Units
  - 16 units
  - support up to 16 different memory addresses to be computed per cycle for 16 threads
  - 16 independent and concurrent loads or stores from device DRAM



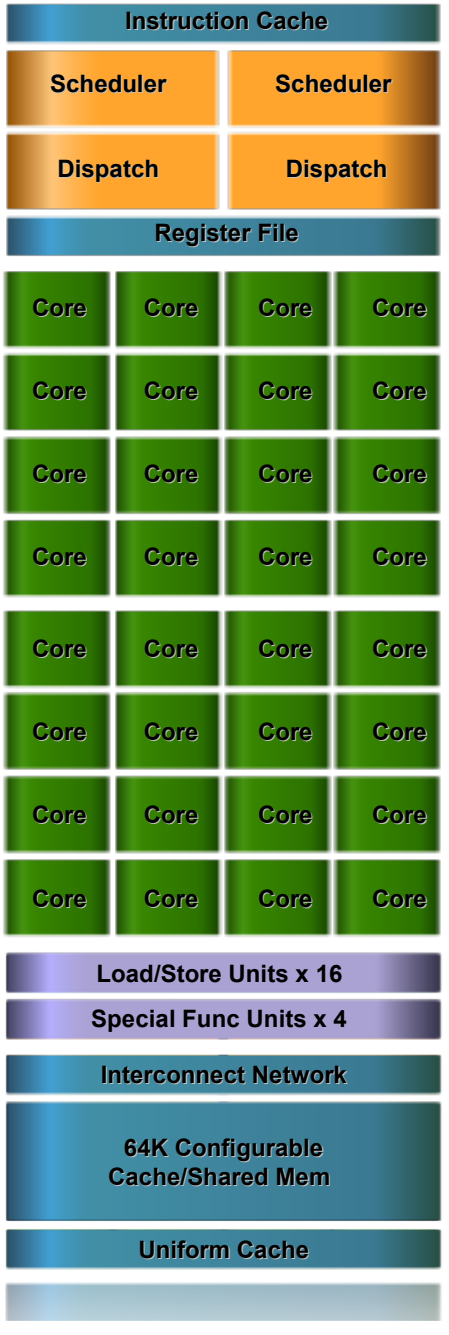
SM Special Function Units

- Special Function Units
  - 4 units per SM
  - execute transcendental instructions
    - sin, cos, 1/*x*, sqrt
  - each SFU issues one GPU instruction per thread per clock cycle
  - SFU pipeline decoupled from dispatch unit permitting other pipelines to be used while SFU pipeline is occupied



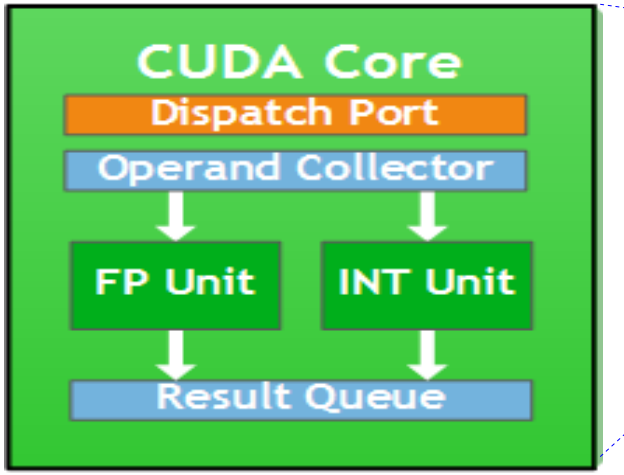
Parallel Instruction Execution in SMs

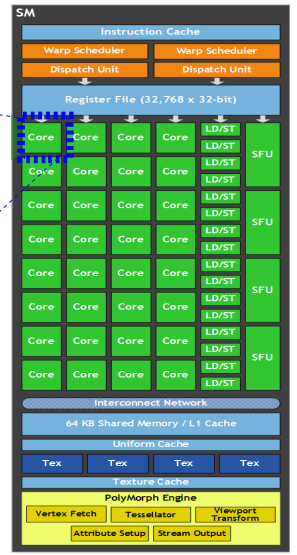
- SIMT (single instruction multiple thread) execution
  - CTs run in groups of Warps
  - Cts in a Warp share instruction unit (UI)
- H/W Multi-threading
  - h/w resource allocatiom and CT scheduling
  - h/w uses CTs to hide latency
- CTs dispatching
  - a Warp not waiting for something can run
  - Warp context switches have zero latency



CUDA "Core" (CC) Architecture

- Each CUDA Core (CC)
  - consists of 2 decoupled, fully pipelined execution units
  - 32/64 bit integer ALU optimized for 64-bit ops
    - boolean, shift, move,convert, bit-field, reverse, count, etc.
  - floating-point ALU
  - support for single and double precision IEEE 754-2008 standard
  - support for fused multiply-add (FMA)
  - predication field for each isntruction





2nd Generation Parallel Thread ISA

- Fermi is architecture to support the new Parallel Thread eXecution (PTX) 2.0 ISA.
  - PTX is a low level *virtual machine* and *ISA* designed to support the operations of a parallel thread processor.
  - At program install time, PTX instructions are translated to machine instructions by the GPU driver.
  - PTX features
    - IEEE FP arithmetic
    - 64-bit addressing and unified address space for variables and pointers
    - new instructions for OpenCL and DirectCompute
    - designed for full C++ support on GPUs

2nd Generation Parallel Thread ISA

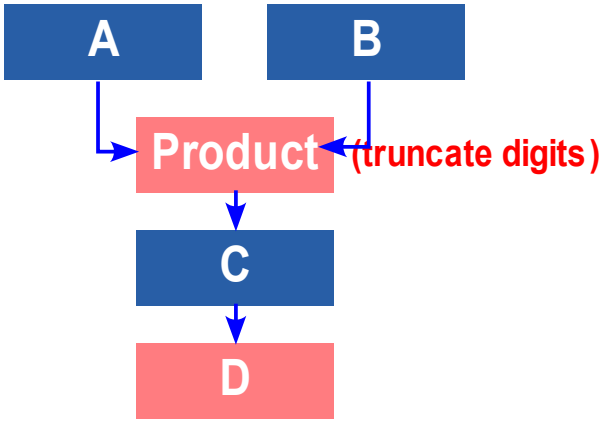
- PTX is a high level assembly code for the GPU
  - stable ISA across GPU h/w generations
  - machine independent ISA for C/C++, Fortran and other high-level languages
  - facilitate hand coding of libraries and performance kernels
  - source to code optimizers for specific GPU h/w
  - achieve high instruction throughput for compiled code
  - GPU resources size independent

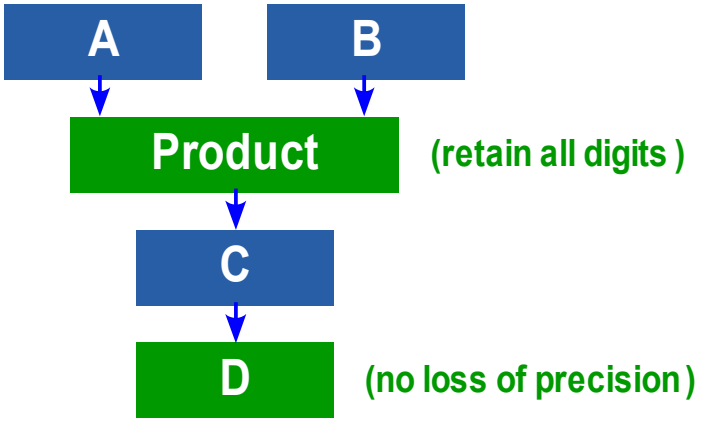
IEEE 32-bit Floating Point Arithmetic

- IEEE 754-2008 32-bit Floating Point instruction h/w support
  - NaNs, +/-Infinity exceptions and de-normal arithmetic done in h/w
    - full-speed de-normal operands and results
    - very costly to handle FP exceptions in s/w
    - or reduced accuracy (flush to zero)
  - IEEE 754-2008 four standard rounding modes in h/w
    - nearest even, zero, +inf, -inf

IEEE 754-2008 Floating Point Support

- IEEE 754-2008 Fused Multiply-Add (FMA)
  - $D = A \times B + C$;
  - No loss of precision
  - IEEE divide and sqrt use FMA
  - FMA maintains full precision until final operation
  - Multiply-Add (MAD) instruction
    - rounding after $A \times B$
    - before assigning to D





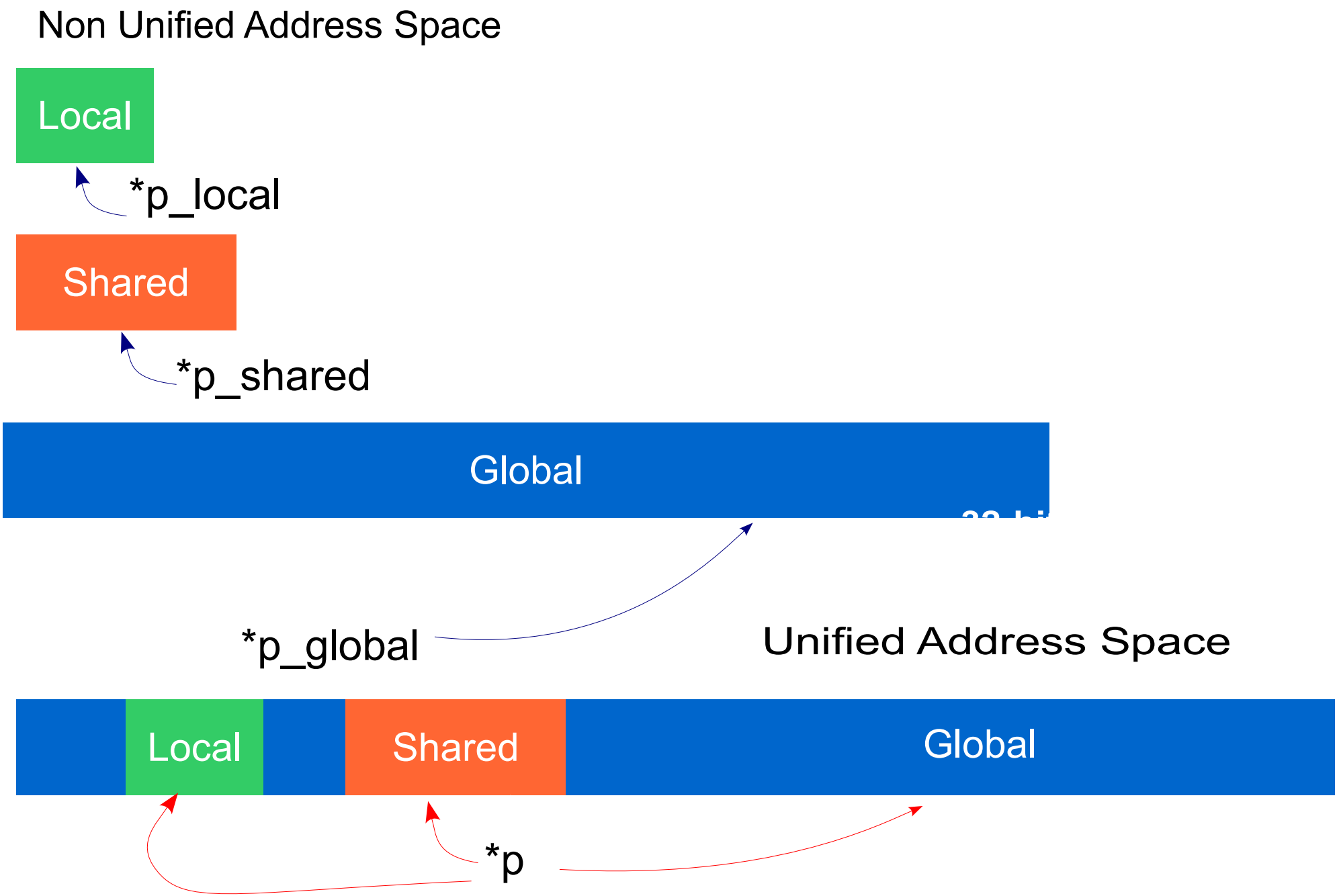
Double Precision Support

- Double precision arithmetic is key for high performance of HPC applications
  - Fermi architecture has been specifically designed to offer high performance in double precision;
  - up to 16 double precision fused multiply-add operations can be performed per SM, per clock

Unified Address Space for Full C++ Support

- Fermi and PTXv2.0 supports in h/w a 40-bit, unified address space for load/store ops covering
  - thread private local,
  - block shared, and
  - global memory
- Facilitates C/C++ pointer ops in code
- Support also for C++ virtual functions, function pointers, `new` and `delete`, `try` and `catch` for exemption handling

Unified Load/Store Addressing



Optimized Support for Other Environments

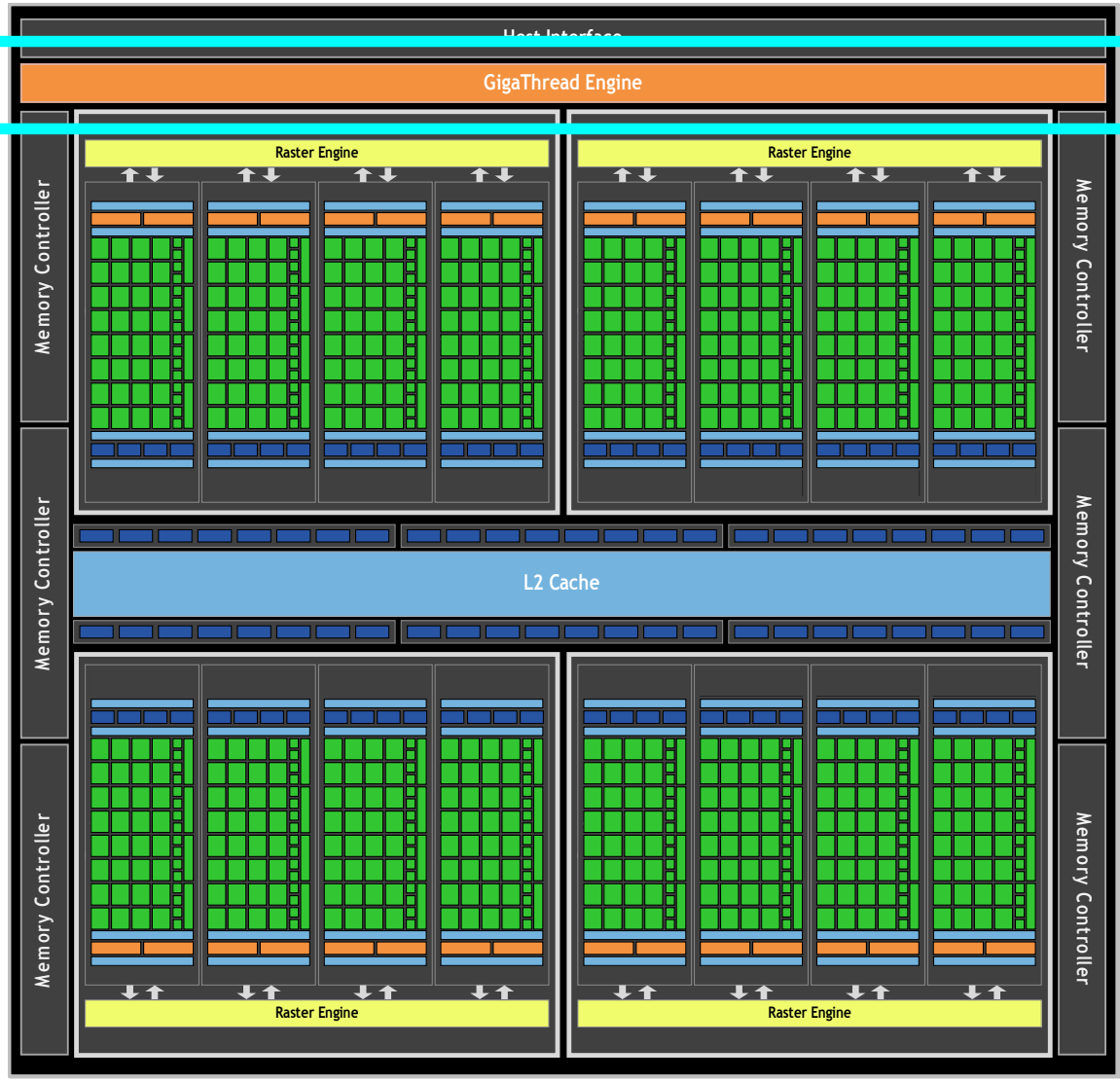
- OpenCL
  - closely related to CUDA machine
    - threads, thread blocks, grids of thread blocks, barrier synch, per block shared memory. global memory and atomic ops
  - open standard compliant and portable
- DirectCompute

Atomic Operations: Read, Modify, Write

- **Fast inter-block, thread-safe communication**
  - atomic ops add, min, max, compare and swap
  - synchronization via atomic read/modify/write instructions
    - Much faster parallel aggregation
      - faster ray tracing, histogram computation, parallel sorting, clustering and pattern recognition, face recognition, speech recognition, BLAS, *etc*.
      - Accelerated by cached memory hierarchy

Fermi/GF100 GigaThread Scheduler

- GigaThread Scheduler
- 2-level distributed thread scheduling for Fermi
  - Chip level global work distribution engine schedules thread blocks to SMs
  - SM level WARP scheduler distriburtes 32 threads to SM cores

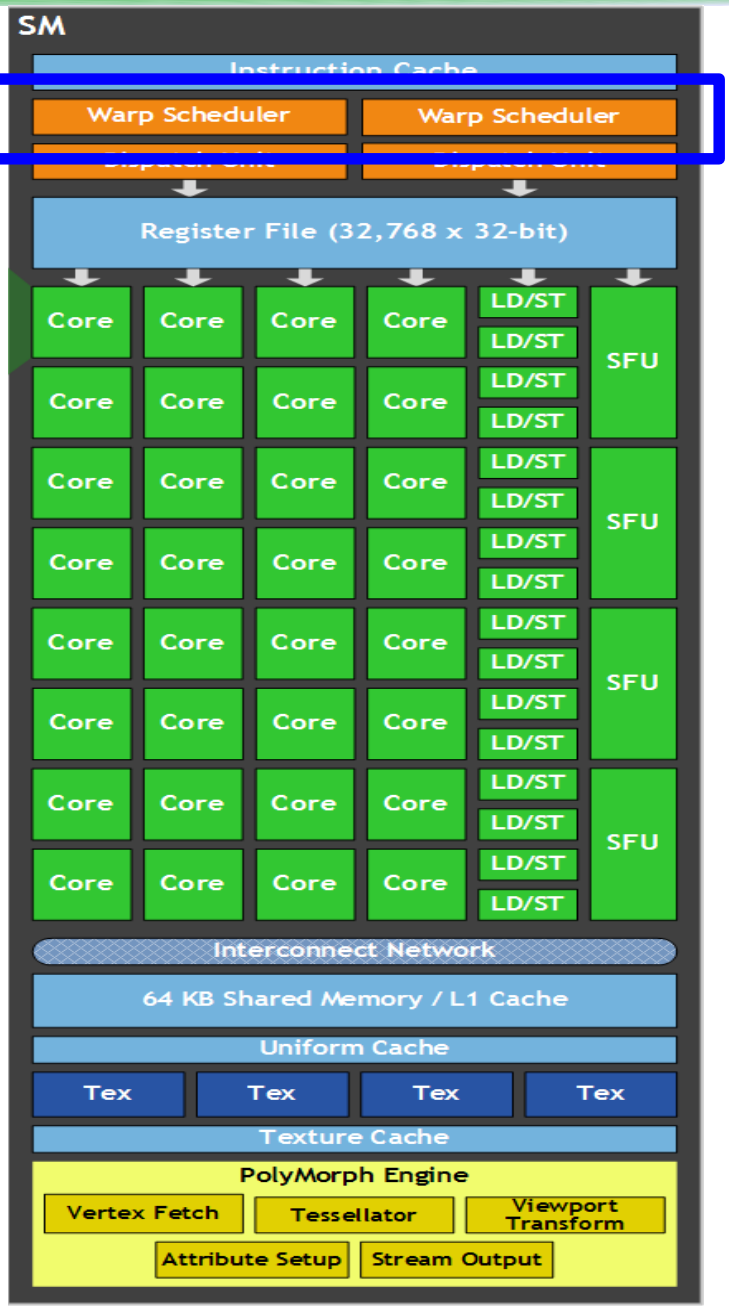


Fermi GigaThread Scheduler

- Efficient application context switching
  - thread blocks can be switched < 25 micro-seconds on SMs
- Concurrent Kernel Execution
  - different kernels of the same application can execute in the GPU concurrently
  - kernels from different application (CUDA contexts) can run sequentially

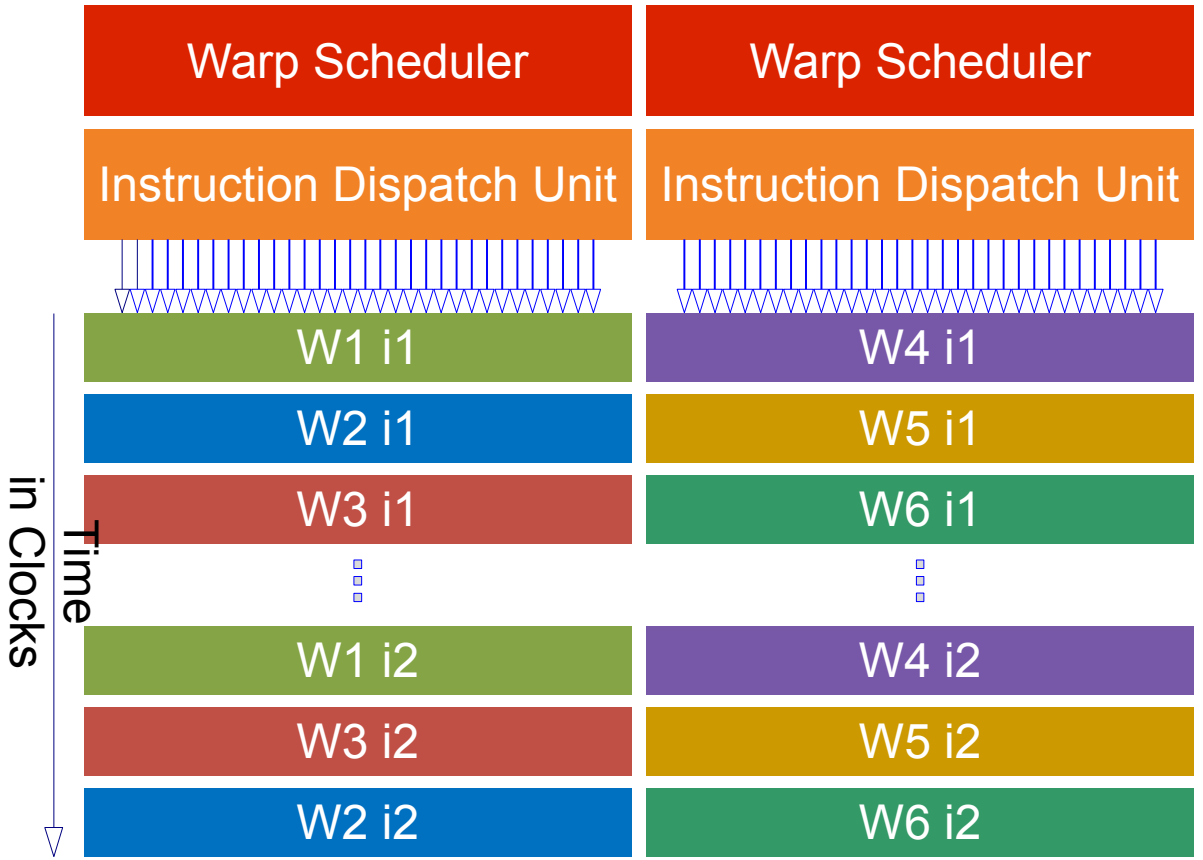
SM Dual Warp Scheduler (WS)

- SM schedules threads in WARPs
  - WARP = 32 threads
  - 2 WARP Schedulers / SM
  - 2 instruction dispatch units / SM
  - WS selects 2 WARPs and issues one instruction / WARP to a group of
    - 16 cores
    - 16 LD/ST units
    - 4 SFUs
  - most instructions can be dual issued
    - 2 int / 2 FP or
    - mix int/FP, LD/ST, SFU instruction
    - dual precision FP do NOT dual issue with other instructions



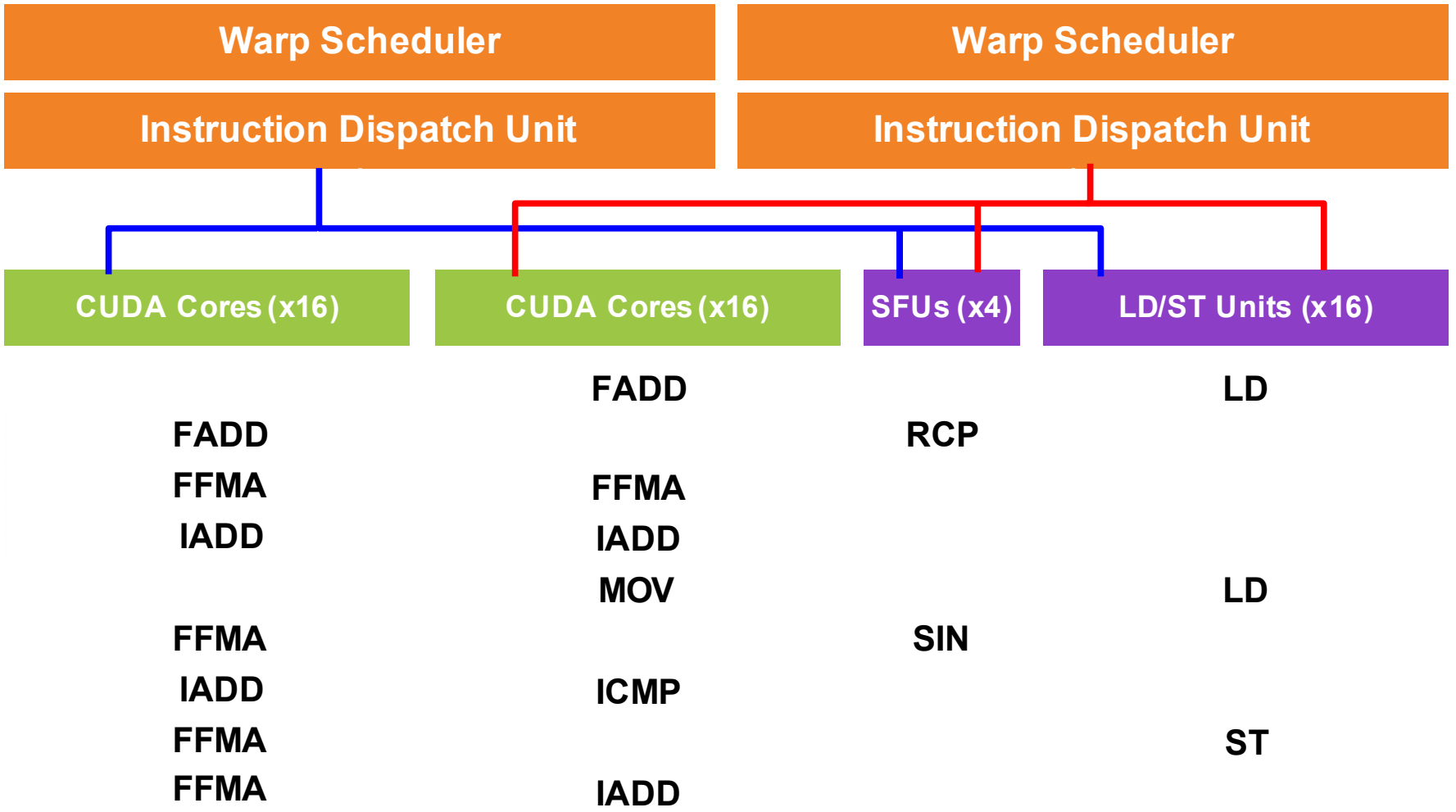
WS Instruction Issue and Control Flow

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



SM Operational Diagram

Fermi Dual Issue

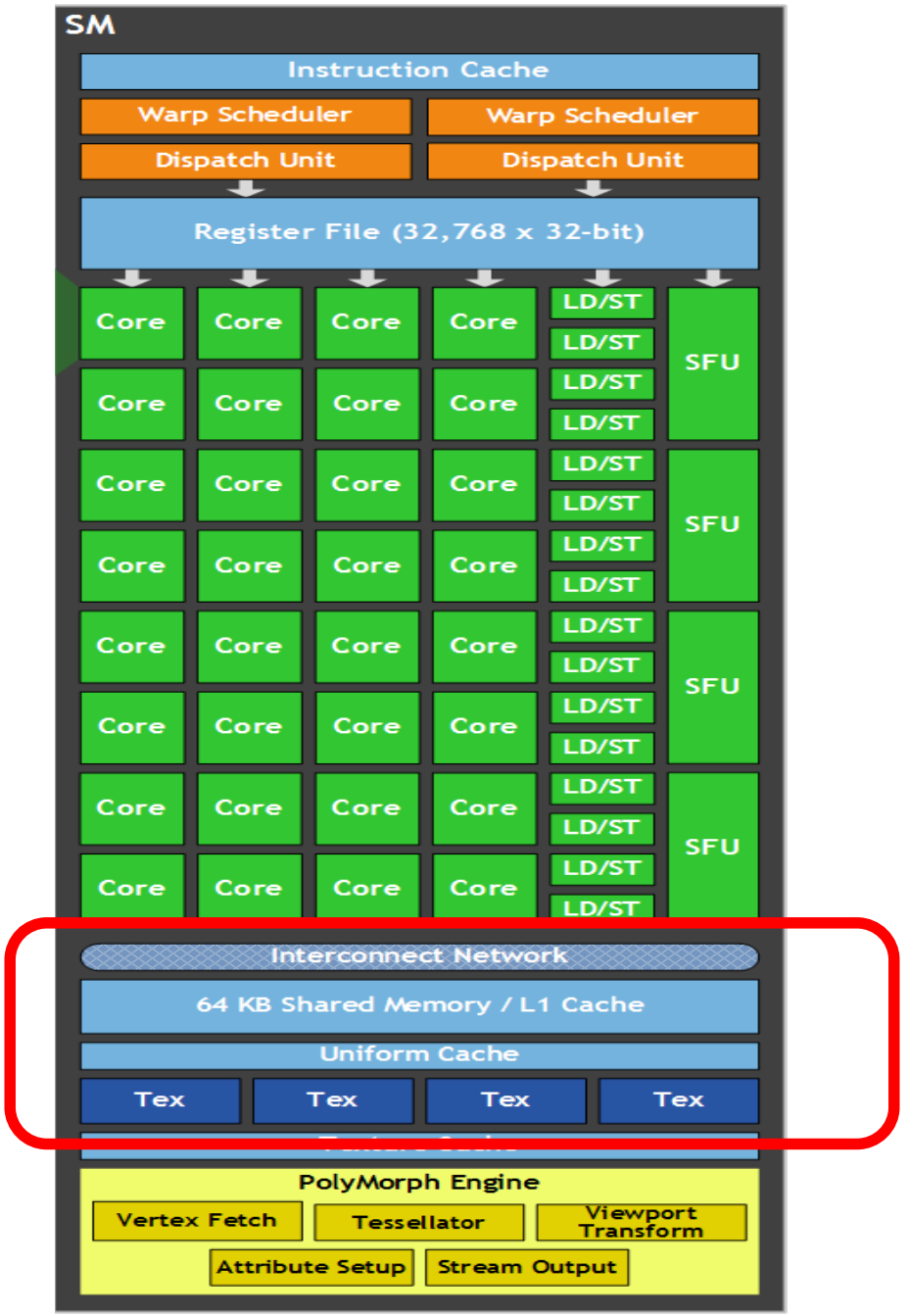


- Peak instruction throughput per instruction type per SM

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

First Level Caches in SM

- Configurable, on-chip L1 cache and shared memory per SM
  - 16KiB L1 and 48KiB Shared Memory OR 48KB L1 and 16KB Shared Memory
  - L1 more effective than shared memory for irregular or unpredictable access
    - Ray tracing, sparse matrix multiply, physics kernels …
  - Caching helps latency sensitive cases

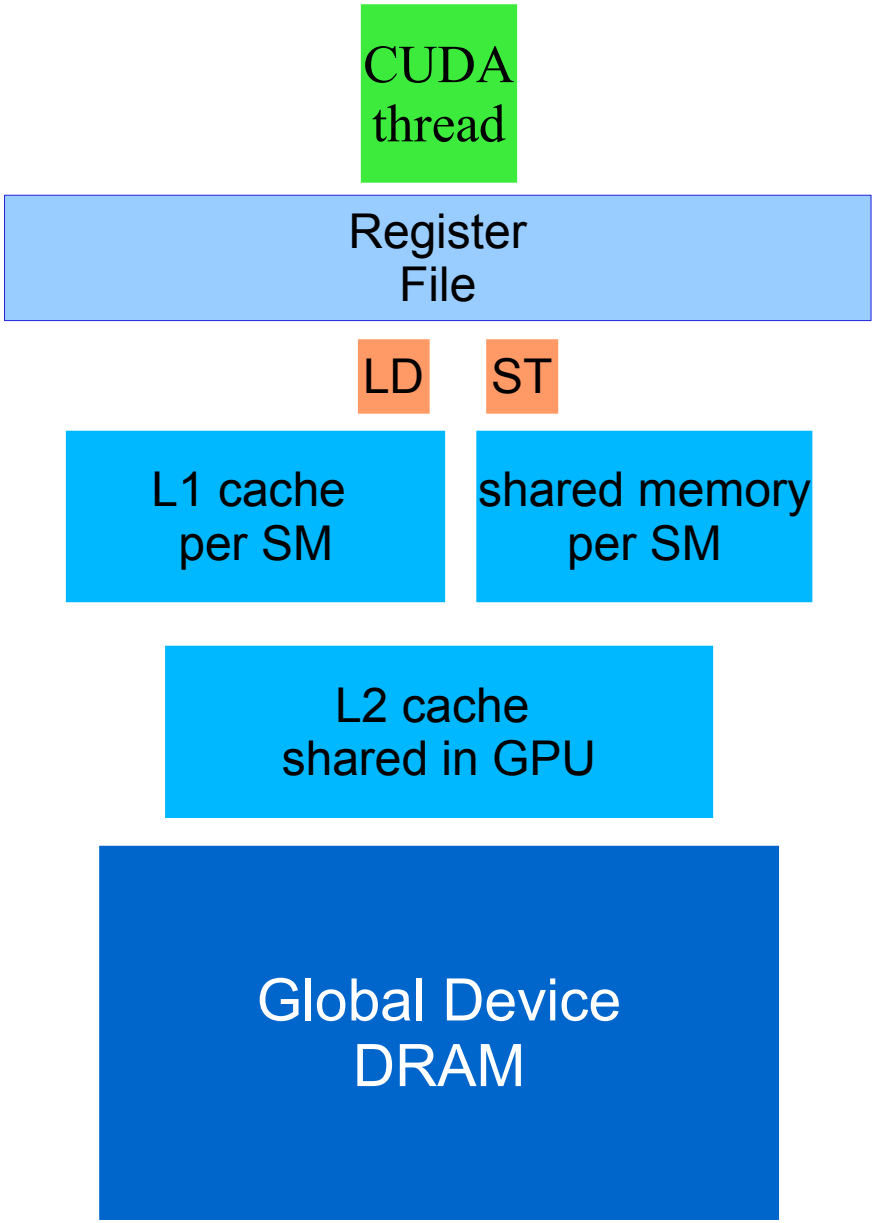


First Level Caches in SM

- Each SM has each own 64KiB to be configured as L1 cache and shared memory
- Shared memory enables voluntary cooperation and on-chip data reuse for threads within the same thread block
- All code benefits from L1

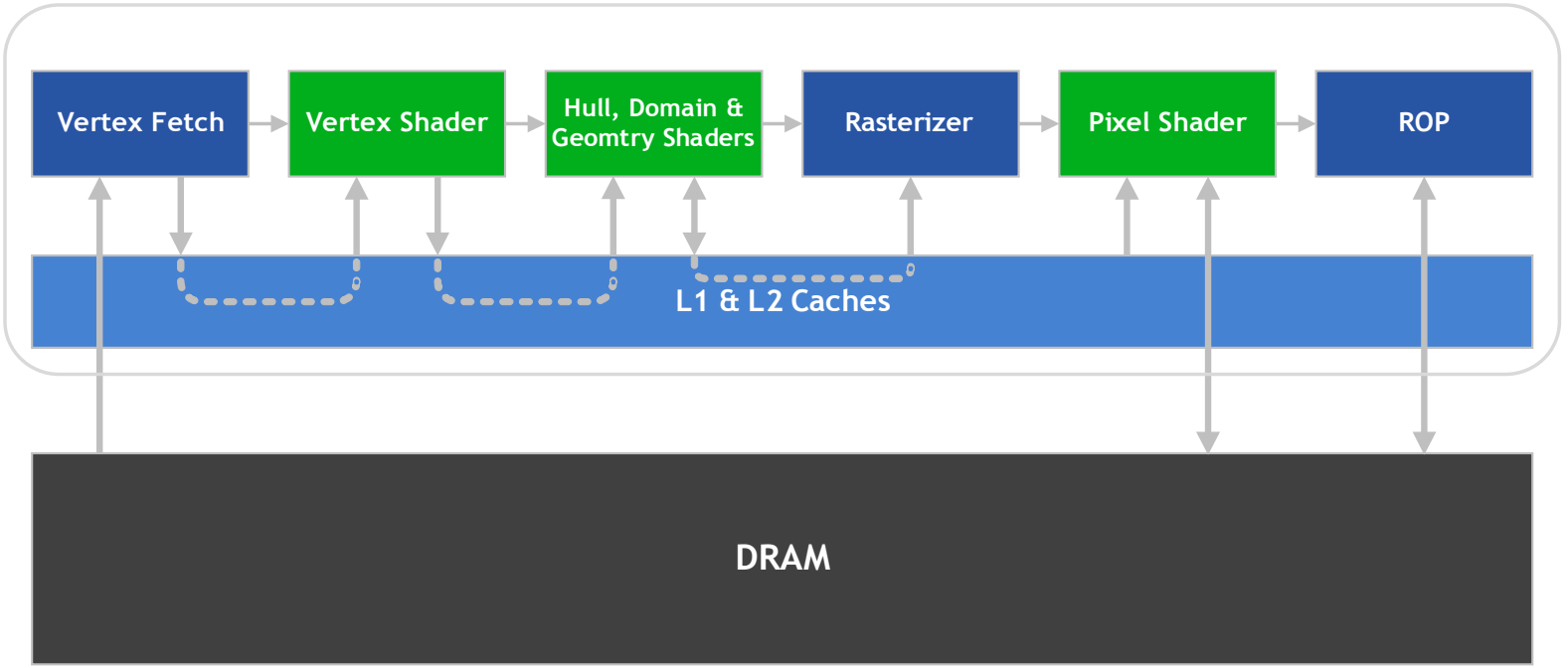
Memory Hierachy in Fermi

- Each CUDA thread can access
  - 32768 X 32bit register file (128KiB)
  - configurable L1 cache / shared memory per SM
    - L1 caches local or global data
    - temp register spills
  - shared 768KB L2 cache across chip (LD, ST, tex)
  - global GPU off-chip memory



Cache Usage: Graphics

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



ECC on Fermi

- **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
  - 3GiB DDR5 reduces to 2.62 GiB
  - 6GiB DDR5 reduces to 5.25 GiB
- **ECC is a must have for many computing applications**
  - by user feedback

Multi-Core and Many-Core Paradigms

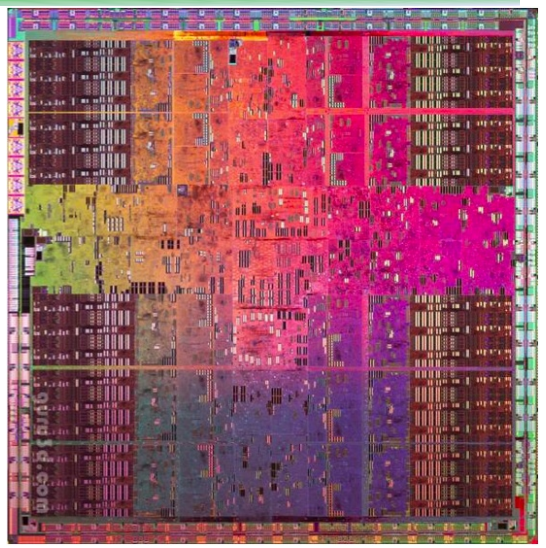
- **Multi-core Design, currently**
  - **1-10 cores per die**
  - **emphasis on speeding up execution of scalar code**
  - **its core is a complex, wide-issue, supercalar instruction pipeline, with multiple execution units,**
  - **support for ILP by out-of-order execution of micro-operations**
  - **heavy branch prediction processing with speculative execution**
  - **h/w multilevel data pre-fetching**
  - **1 to 4 h/w threads per core (SMT)**

Multi-Core and Many-Core

- **Many-core**
  - **10 – 100s cores per die**
  - **individual core performance is not emphasized**
  - **simpler core design**
  - **cores are used together for higher aggregate throughput**
  - **SIMD approach**
    - **Single Instruction applied to Multiple Data**
  - **MP-SIMD approach**
    - **Different instructions can be applied to different sub-sets of data**

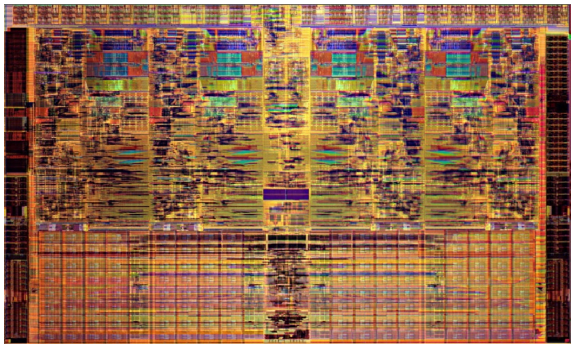
Multi-Core and Many-Core

Specifications	Core i7 960	GTX285
Processing Elements per Chip	4 cores, 4 way SIMD @3.2 GHz	30 cores, 8 way SIMD @1.5 GHz
Resident Strands/Threads (max)	4 cores, 2 threads, 4 way SIMD: 32 strands	30 cores, 32 SIMD vectors, 32 way SIMD: 30720 threads
SP GFLOP/s	102	1080
Memory Bandwidth	25.6 GB/s	159 GB/s
Register File	-	1.875 MB
Local Store	-	480 kB



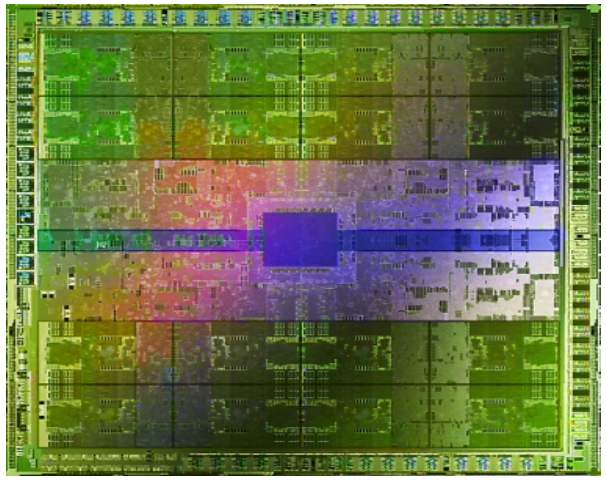
GTX285 (55nm)

Core i7 (45nm)



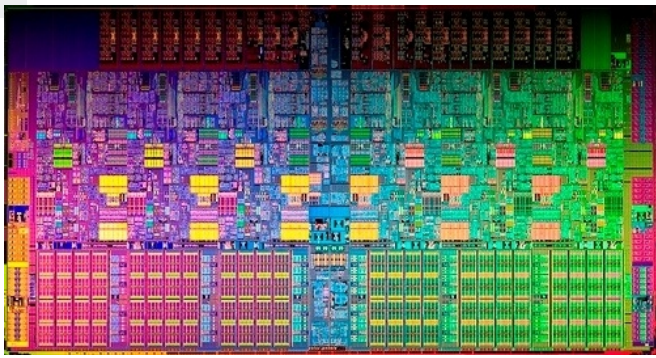
Multi-Core and Many-Core

Specifications	Westmere-EP	Fermi (GF100)
Processing Elements	6 cores, 2-issue 4-way SIMD @3.46 GHz	16 cores, 2-issue, 16 way SIMD @1.54 GHz
Resident Strands / Threads (max)	6 cores, 2 threads, 4 way SIMD: 48 strands	16 cores, 48 SIMD vectors, 32-way SIMD2; 4576 threads
SP GFLOP/s	166	1577
Memory Bandwidth	32 GB/s	192 GB/s
Register File	6 KB	2 MB
Local Store L1 cache	192 KB	1024 KB
L2 cache	1536 KB	0.75 MB
L3 cache	12 MB	–



Fermi (40nm)

Westmere (32nm)



Credits

[1] Bryan Catanzaro, *University of California Berkeley*

[2] Craig M. Wittenbrink, Emmett Kilgariff, Arjun Prabhu, Michael C. Shebanow, *nVidia Research*

[3] David Kirk/NVIDIA and Wen-mei W. Hwu, *UIUC*

[4] UCB ParLab 2010, ParLab2009, *HotChips*

[5] SU CS193G, Spring 2010

[6] UCB CS267, Spring 2011

[7] UIUC ECE 498AL, Spring 2010

[8] nVidia Tech Reports, Documentation

- OpenOffice.org used for art-work