

Parallel Programming

Data-Parallel Programming with CUDA

Professor Yi-Ping You (游逸平)

Department of Computer Science

<http://www.cs.nctu.edu.tw/~ypyou/>



Acknowledgement

- This set of slides is adapted from lectures by
 - ✚ Prof. Wen-mei Hwu (University of Illinois, Urbana-Champaign) and Dr. David Kirk (NVIDIA)
 - ✚ Dr. Erich Elsen (Consulting Assistant Professor at Stanford University)
 - ✚ Prof. Patrick Cozzi (University of Pennsylvania)
 - ✚ Prof. Henk Corporaal and Dr. Bart Mesman (Technische Universiteit Eindhoven)



Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ◆ Built-in Data Types and Functions
 - ◆ Thread Model
 - ◆ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization

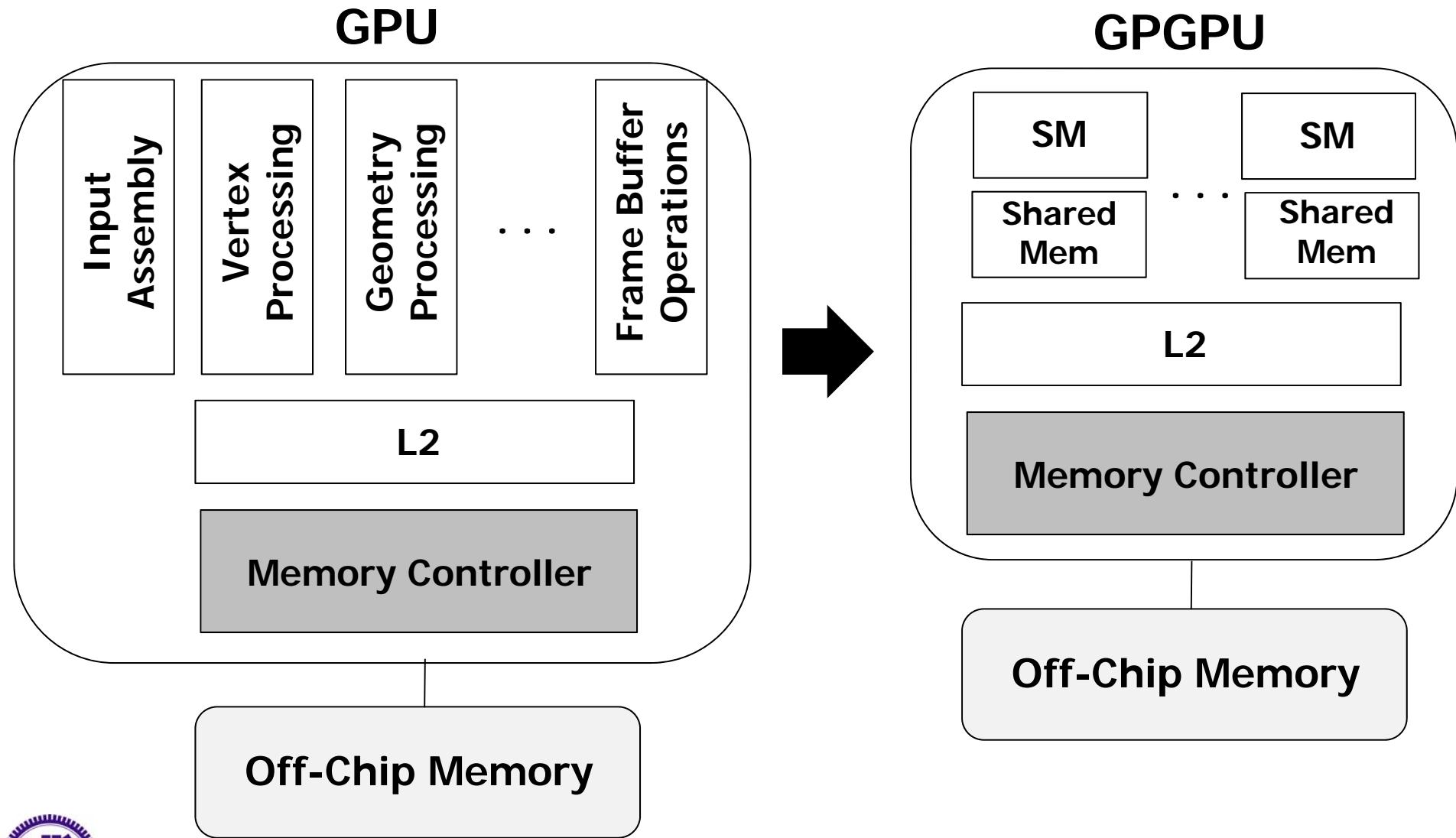


GPU Computing History

- 2001/2002 - researchers see GPU as data-parallel coprocessor
 - ❖ The *GPGPU* field is born
- 2007 - NVIDIA releases CUDA
 - ❖ *CUDA* - Compute Uniform Device Architecture
 - ❖ GPGPU shifts to *GPU Computing*
- 2008 - Khronos releases *OpenCL* specification

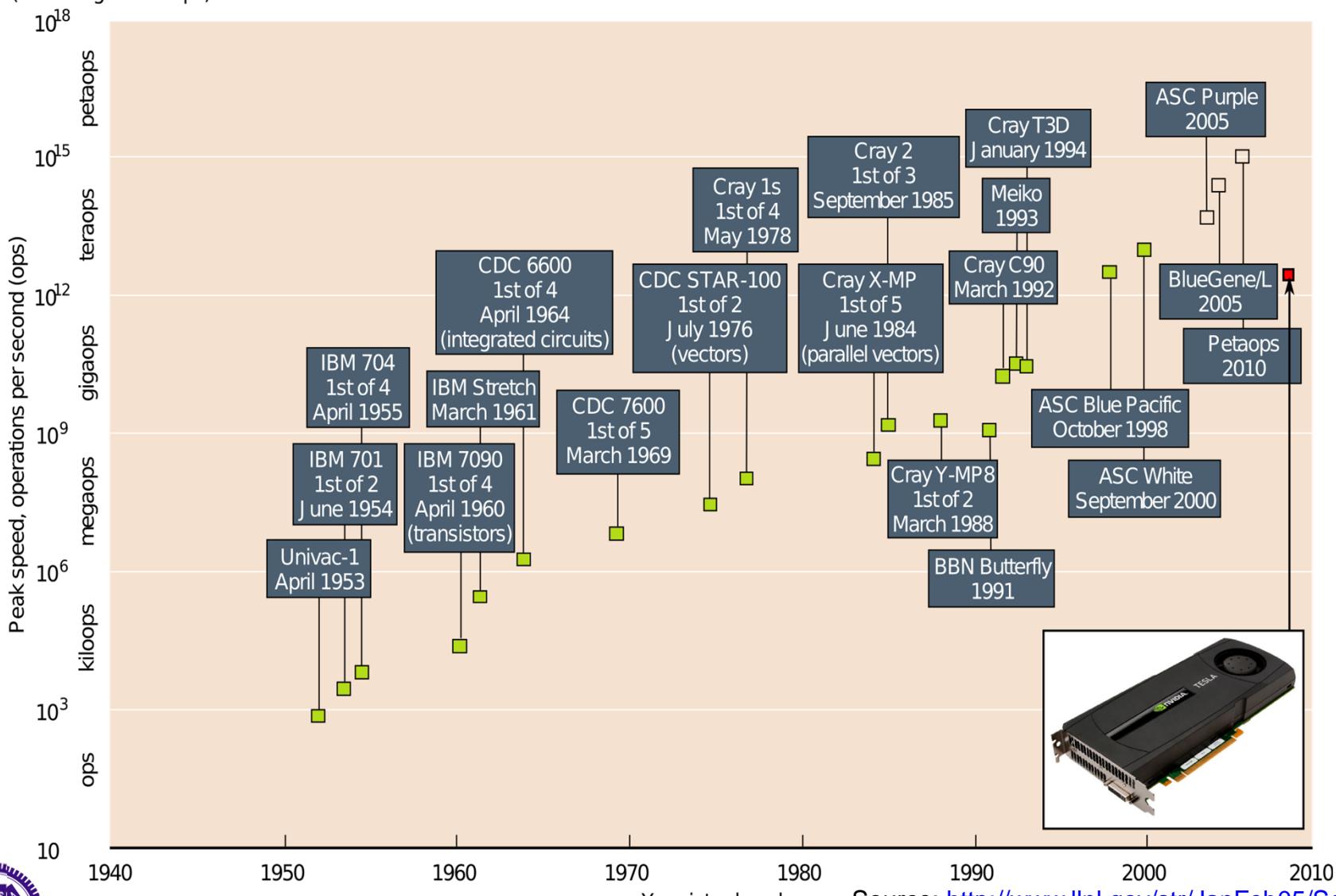


From GPU to GPGPU



Single-Chip GPU vs Fastest Supercomputers

(Next range is exaops)



Single-Chip GPU vs Supercomputers



Nvidia K40 (2013)
4.29 TFlop/s



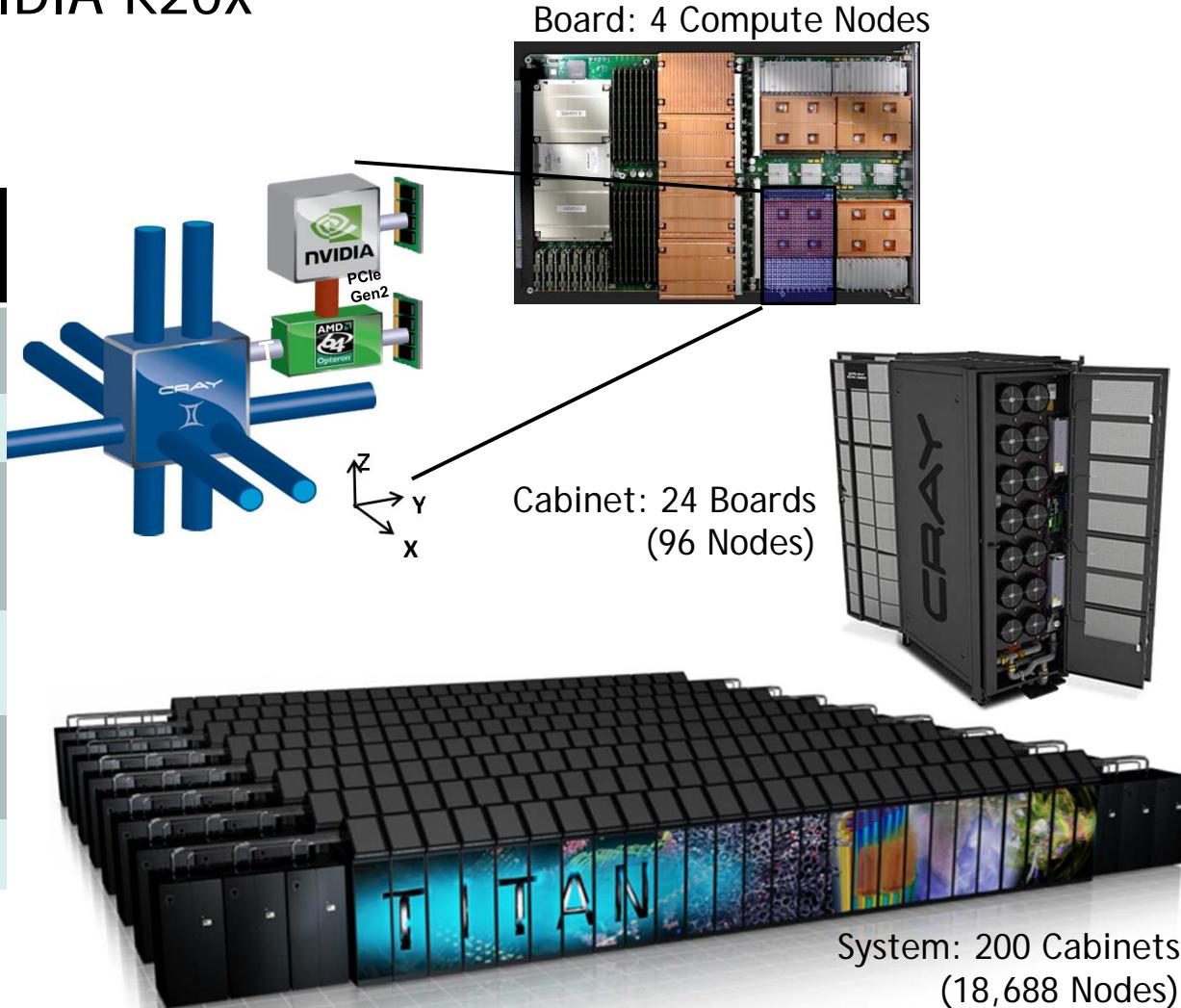
IBM BlueGene/L (2007)
(with 131,072 CPU cores)
367 TFlop/s

Titan (#2, 06/2015)



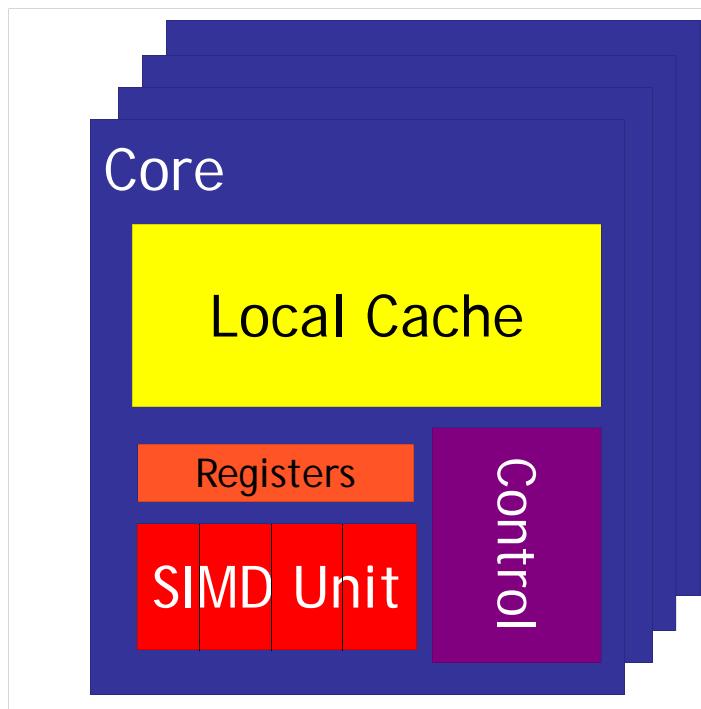
- Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x
- 560,640 cores

XK7 Compute Node Characteristics
AMD Series 6200 (Interlagos)
NVIDIA Kepler
Host Memory 32GB 1600 MT/s DDR3
NVIDIA Tesla X2090 Memory 6GB GDDR5 capacity
Gemini High Speed Interconnect
Keplers in final installation

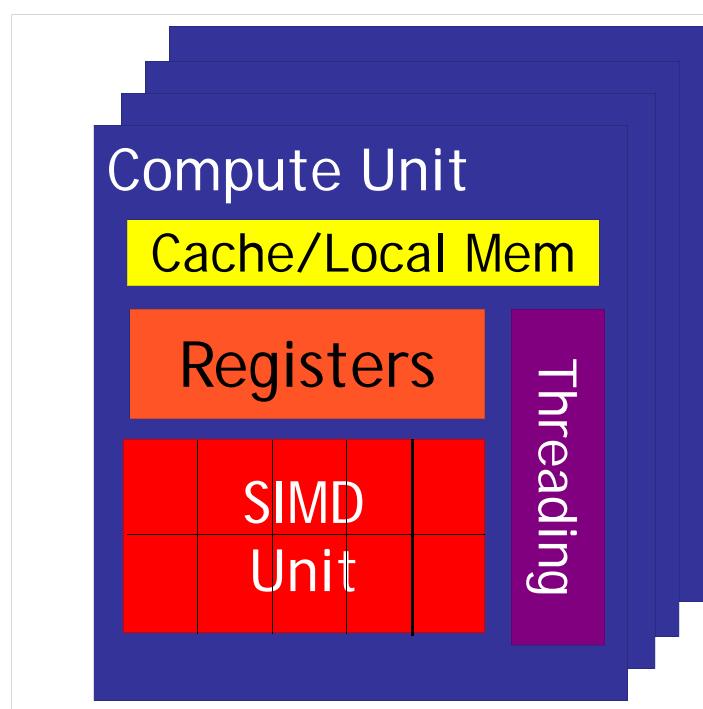


CPU and GPU have very different design philosophy

CPU
Latency Oriented Cores

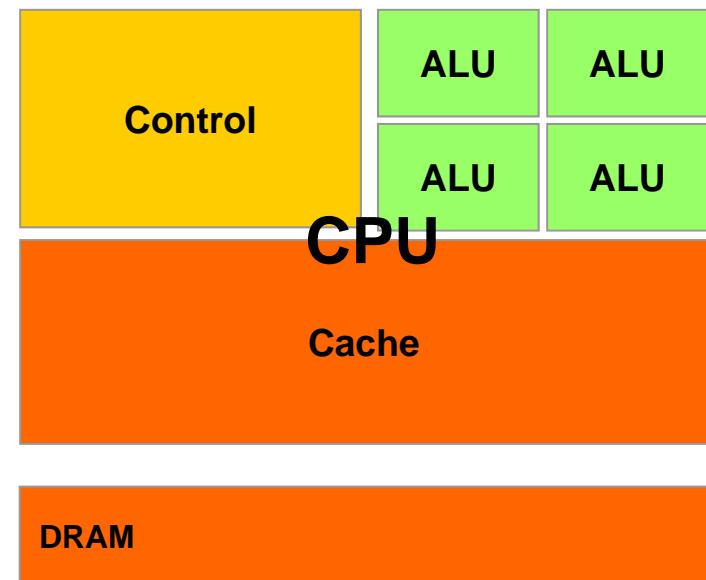


GPU
Throughput Oriented Cores



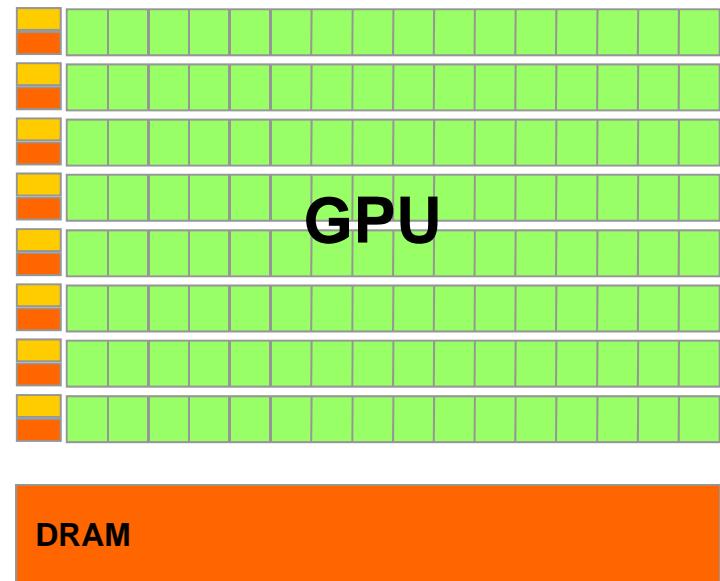
CPUs: Latency Oriented Design

- Large caches
 - ❖ Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
 - ❖ Branch prediction for reduced branch latency
 - ❖ Data forwarding for reduced data latency
- Powerful ALU
 - ❖ Reduced operation latency



GPUs: Throughput Oriented Design

- Small caches
 - ❖ To boost memory throughput
- Simple control
 - ❖ No branch prediction
 - ❖ No data forwarding
- Energy efficient ALUs
 - ❖ Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies



Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
 - ⊕ CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
 - ⊕ GPUs can be 10+X faster than CPUs for parallel code



Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ❖ Built-in Data Types and Functions
 - ❖ Thread Model
 - ❖ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



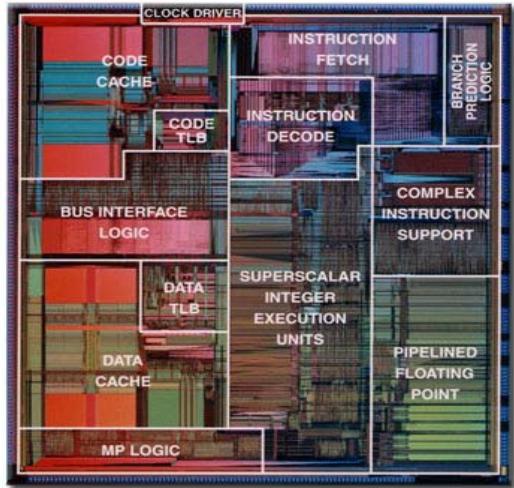
GPU Has 10x Compute Density

- Given the same chip area, the achievable performance of GPU is 10x higher than that of CPU.

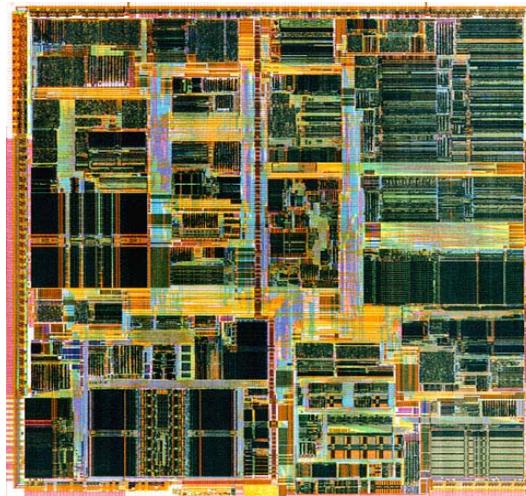


Evolution of Intel Pentium

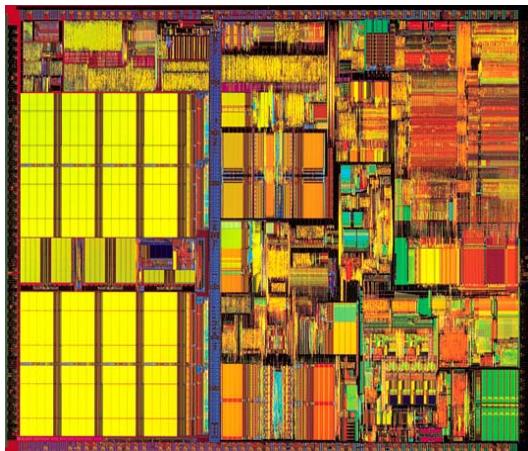
Pentium I



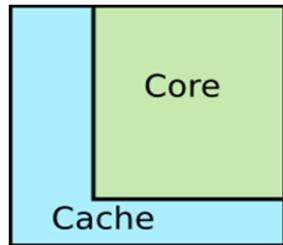
Pentium II



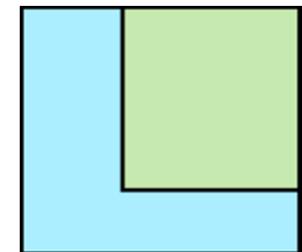
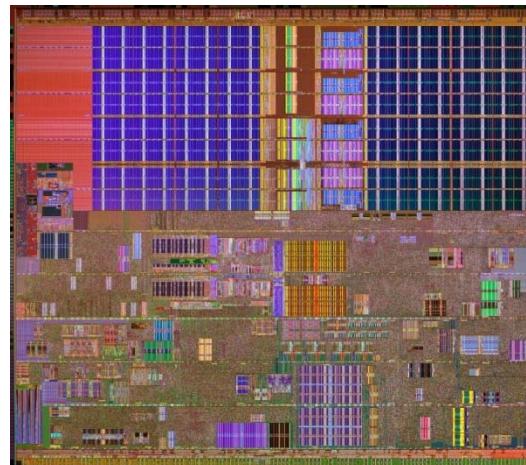
Pentium III



Chip area
breakdown

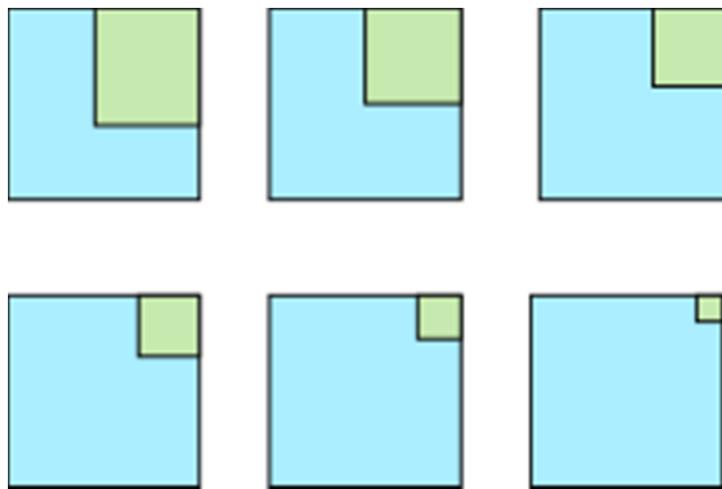


Pentium IV



Extrapolation of Single Core CPU

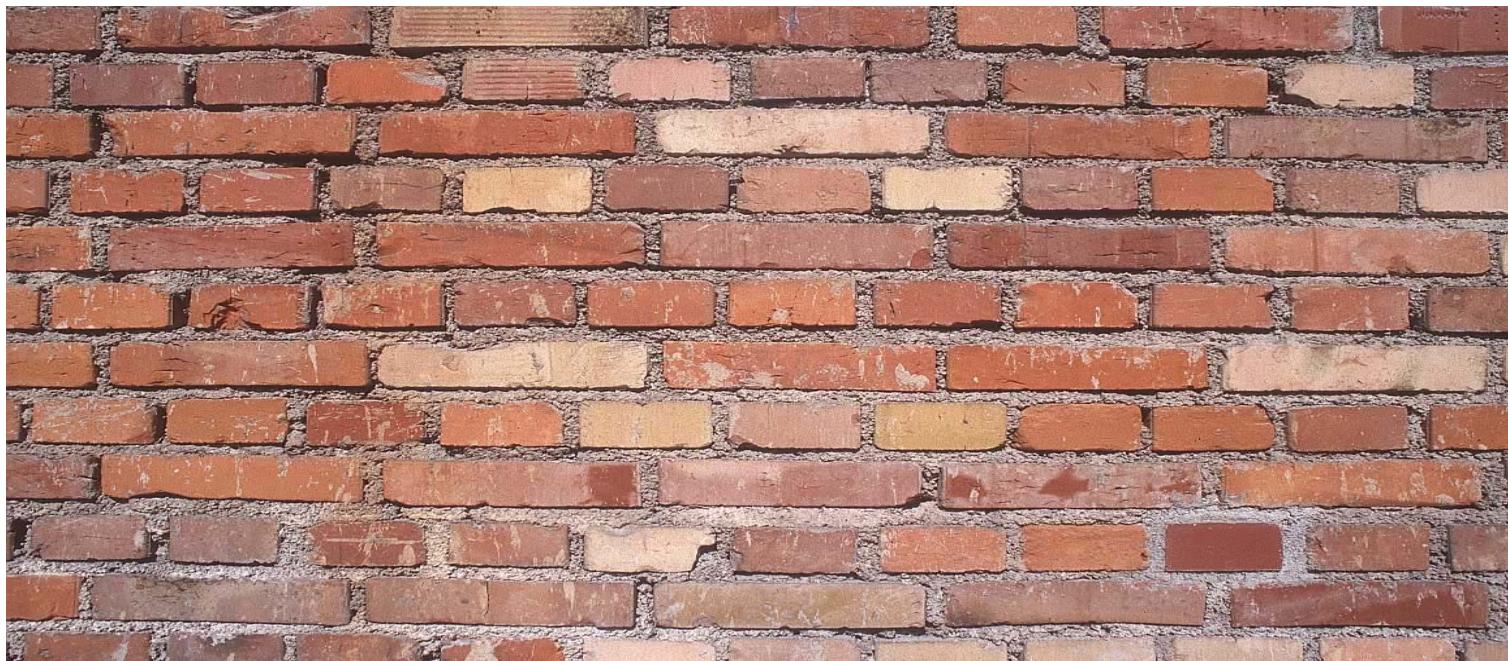
- If we extrapolate the trend, in a few generations, Pentium will look like:



- Of course, we know it did not happen.
- Q: What happened instead? Why?

The Brick Wall -- UC Berkeley's View

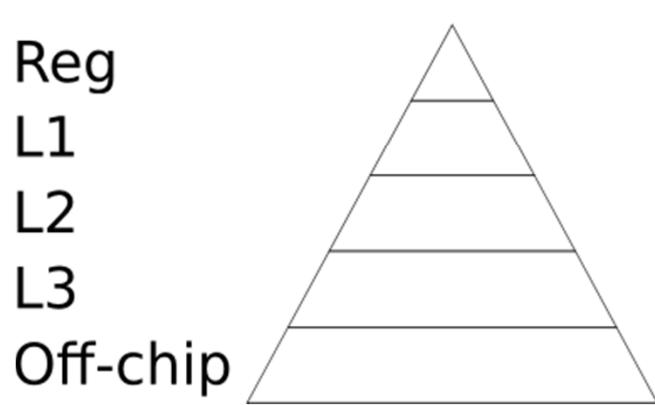
- Memory Wall: Memory slow, multiplies fast
 - ILP Wall: diminishing returns on more ILP HW
 - Power Wall: power expensive, transistors free
- Power Wall + Memory Wall + ILP Wall = Brick Wall



David Patterson, "Computer Architecture is Back - The Berkeley View of the Parallel Computing Research Landscape", Stanford EE Computer Systems Colloquium, Jan 2007,



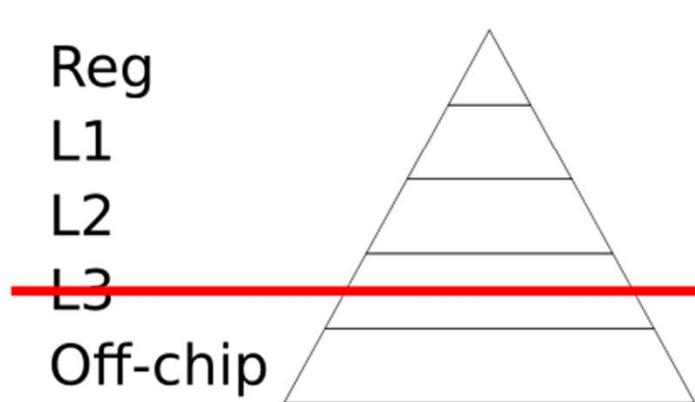
How to Break the Brick Wall?



	Size (Byte)	Energy (pJ)	Delay (cycles)	Bandwidth (GB/s)
Reg	1K	10	1	1000
L1	32K	20	5	100
L2	256K	100	10	100
L3	8M	200	50	100
Off-chip	4G	2000	100	10

Hint: how to exploit the parallelism inside the application?

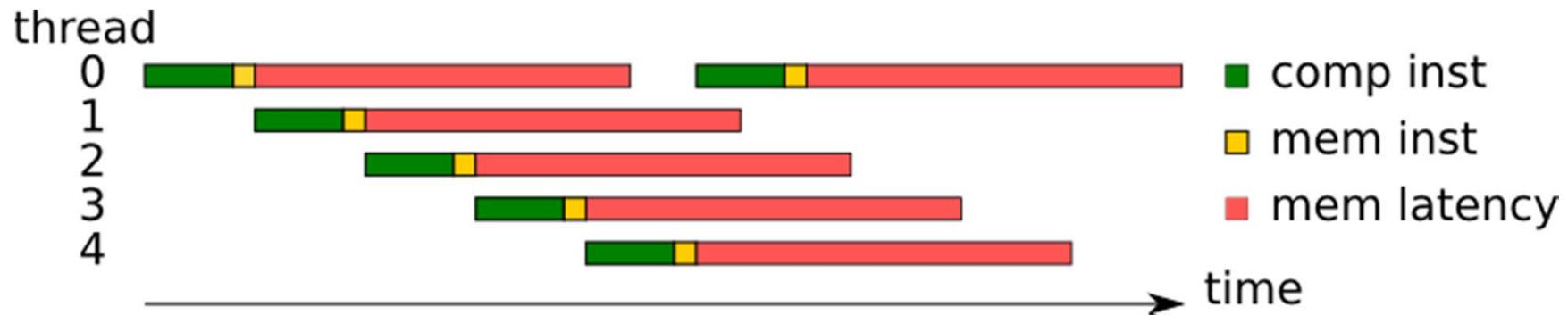
Step 1: Trade Latency with Throughput



	Size (Byte)	Energy (pJ)	Delay (cycles)	Bandwidth (GB/s)
Reg	1K	16K	10~20	1~2~3
L1	32K	20	5	100
L2	256K	100	10	100
L3	8M	200	50	100
Off-chip	4G	2000	100	10

Hind the memory latency through fine-grained interleaved threading.

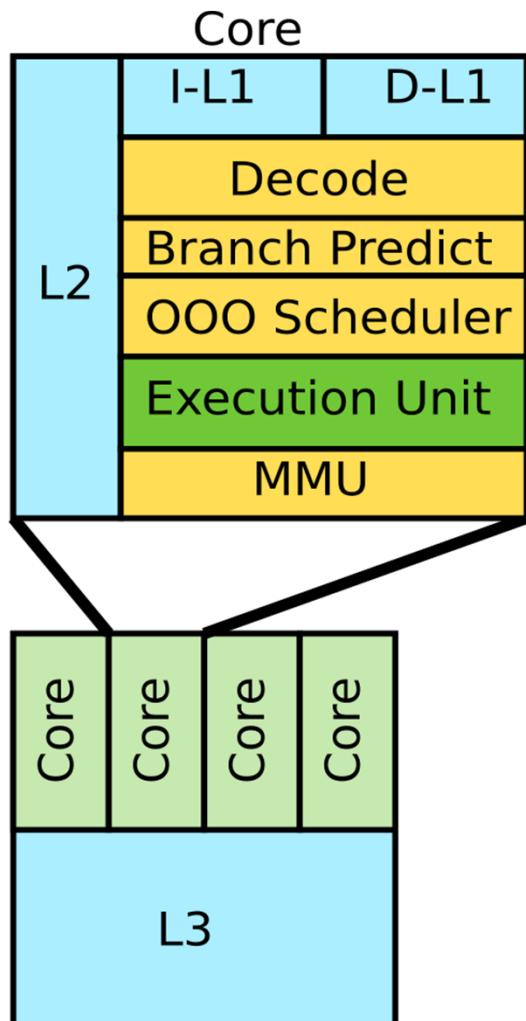
Interleaved Multithreading



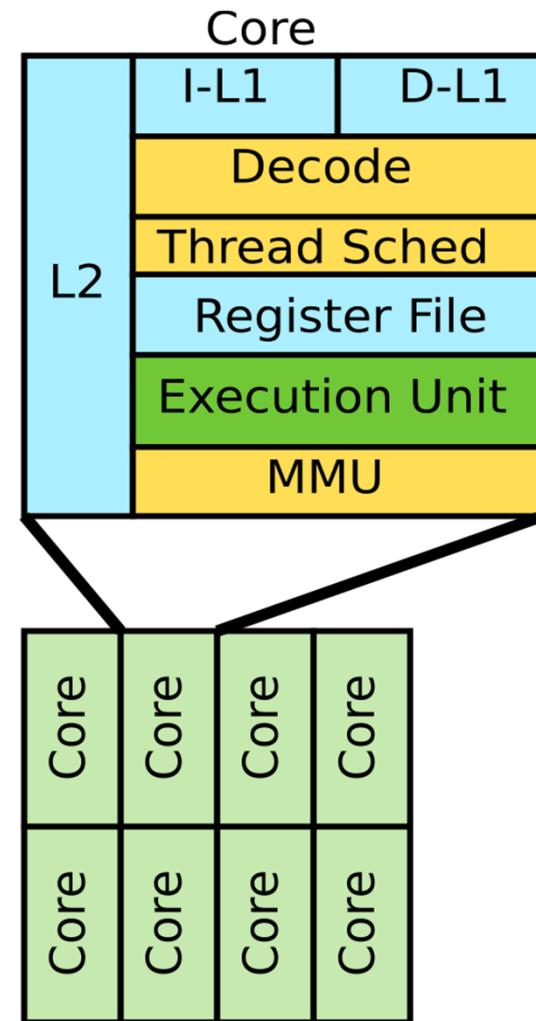
- Fine-grained interleaved multi-threading:
 - ❖ Pros: remove branch predictor, 000 scheduler, large cache
 - ❖ Cons: register pressure, thread scheduler, require huge parallelism, etc.

Fine-Grained Interleaved Threading

Without fine-grained interleaved threading

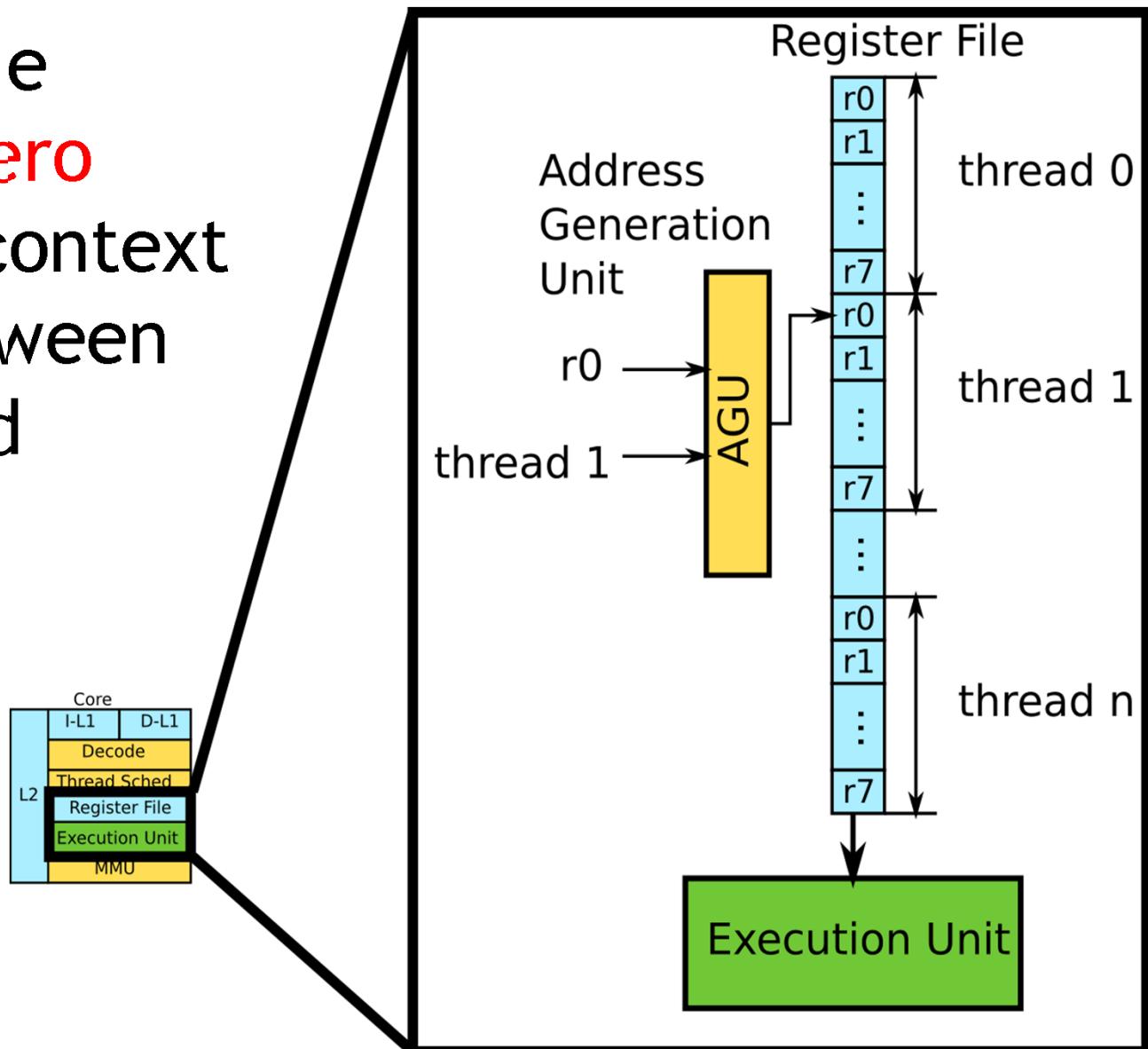


With fine-grained interleaved threading



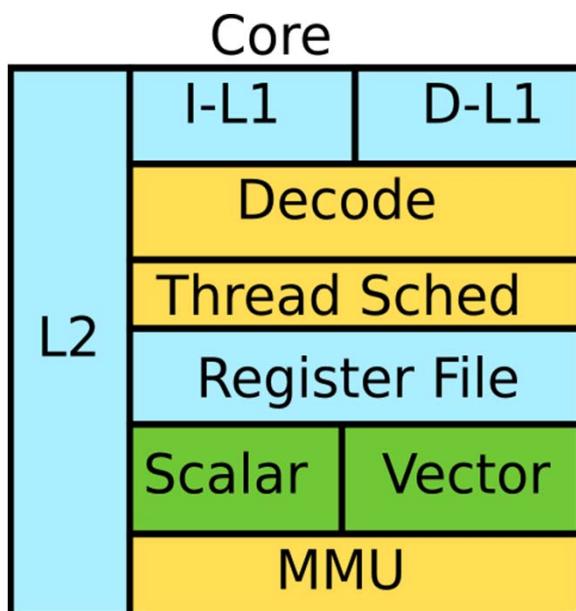
Hardware Support

- Register file supports **zero overhead context switch between interleaved threads**



Can We Make Further Improvement?

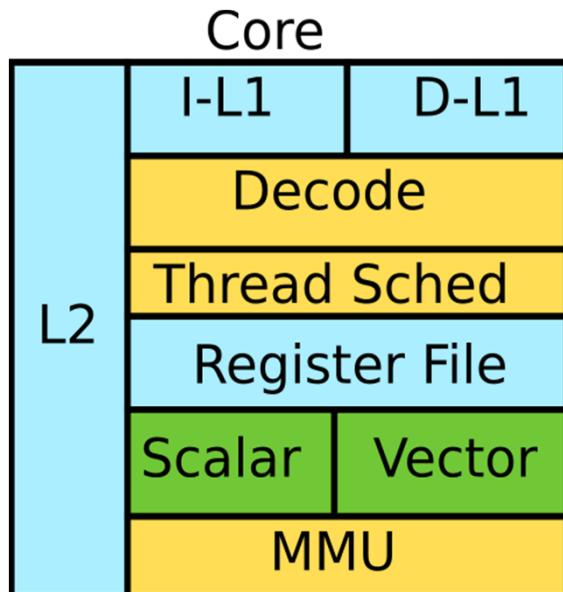
- Reducing large cache gives 2x computational density
- Q: Can we make further improvements?
 - ✚ Hint: We have only utilized thread level parallelism (TLP) so far



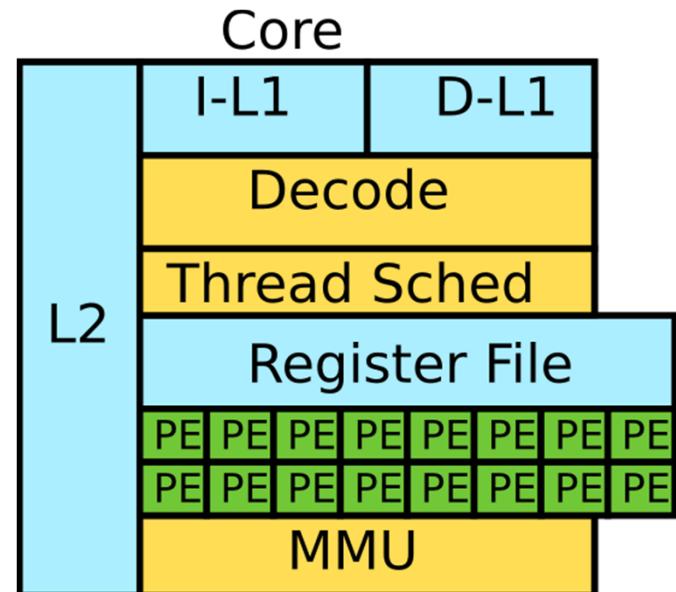
Step 2: Single Instruction Multiple Data

- GPU uses wide SIMD: 8/16/24/32/64/... processing elements (PEs)
 - CPU uses short SIMD: usually has vector width of 4

SSE has 4 data lanes

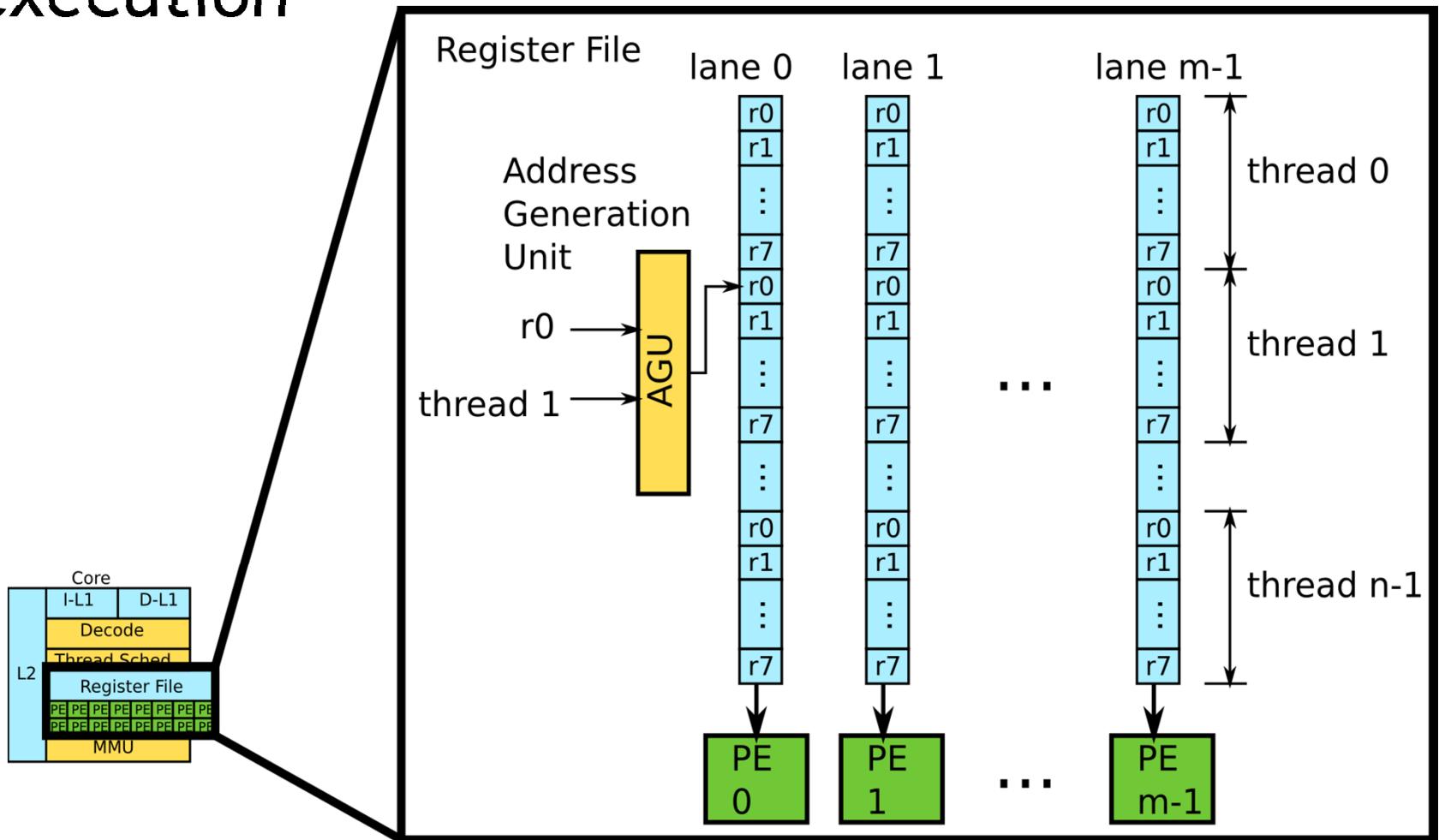


GPU has 8/16/24/... data lanes



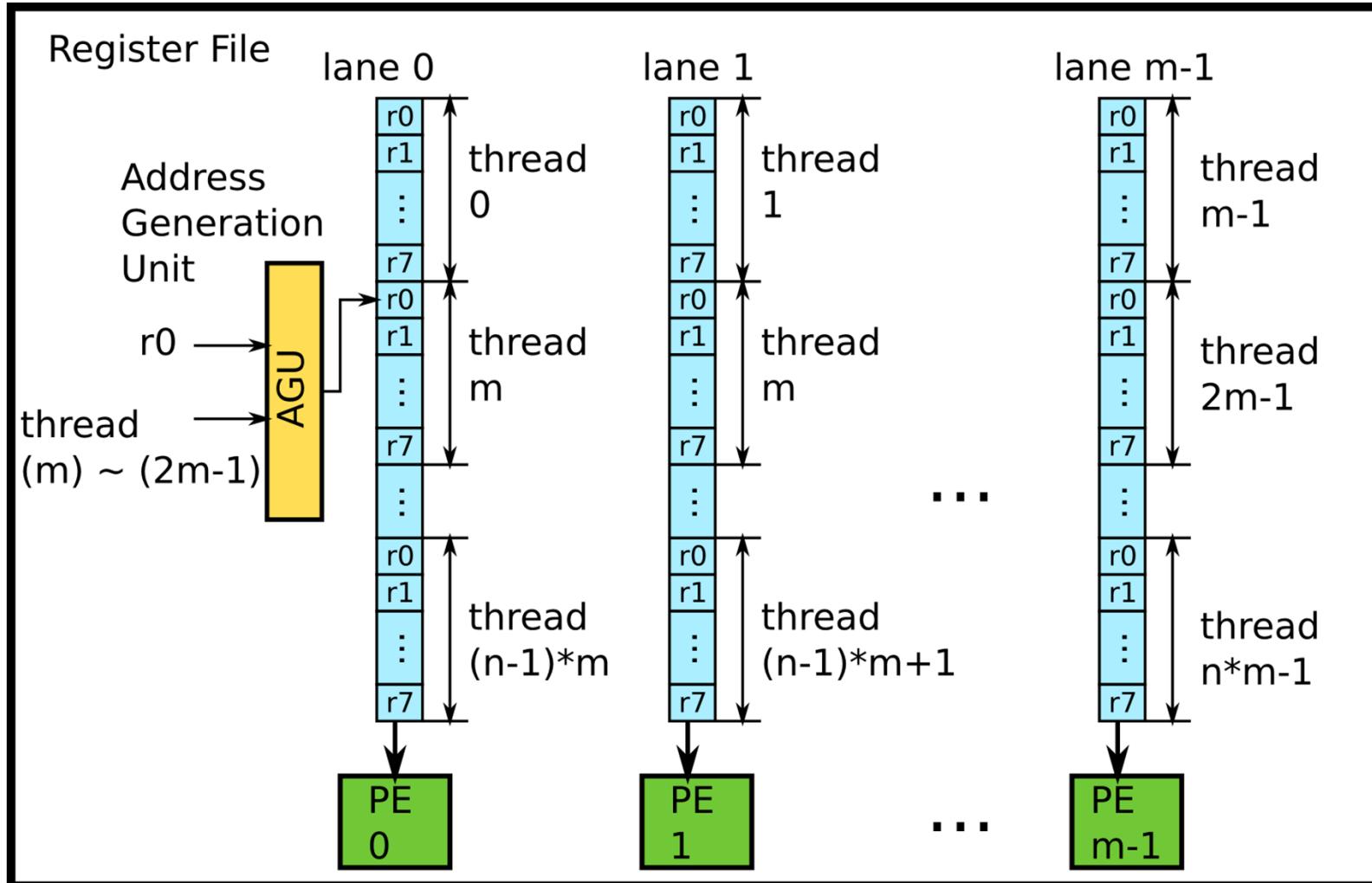
Hardware Support

- Supporting interleaved threading + SIMD execution



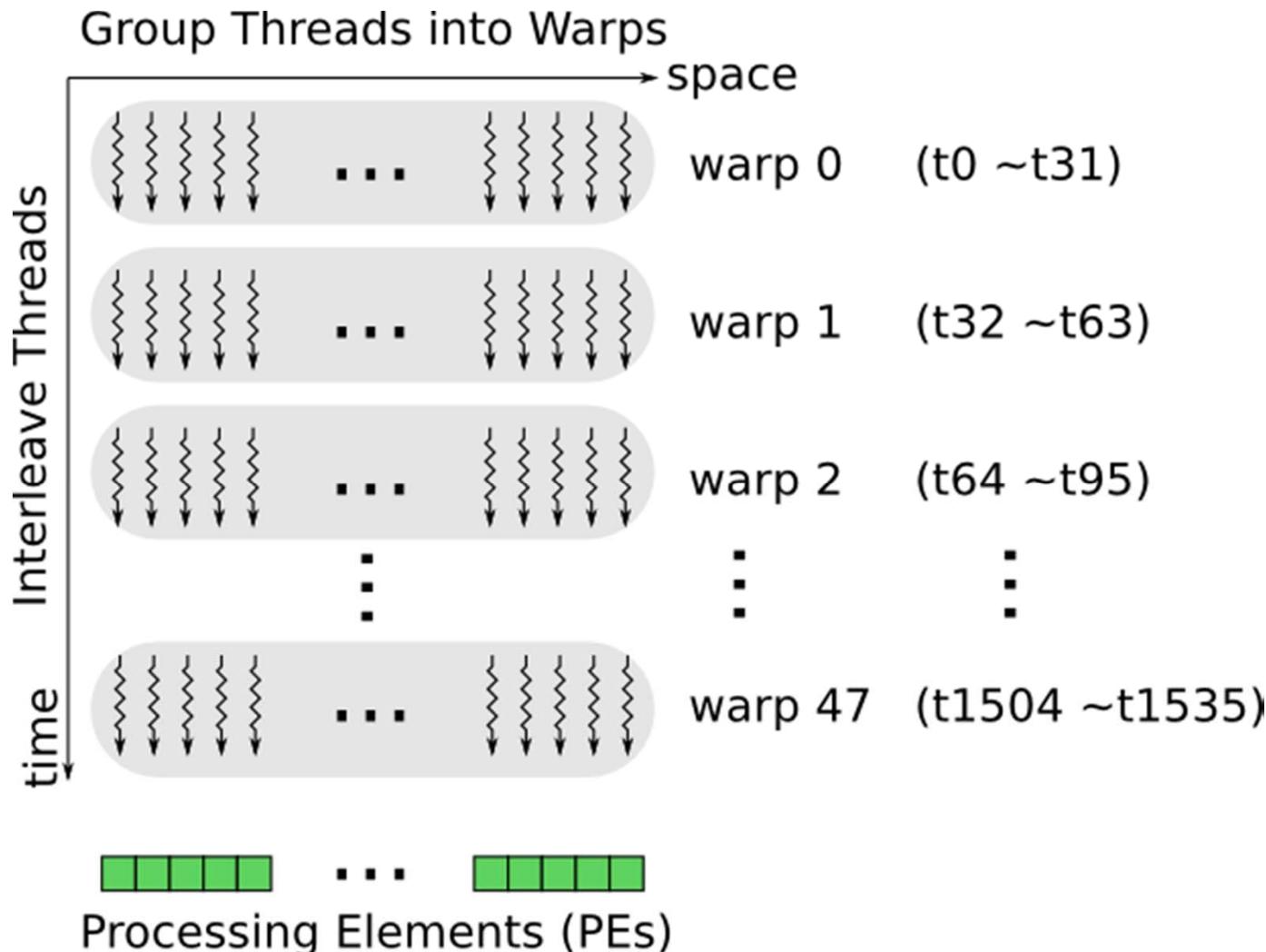
Single Instruction Multiple Thread (SIMT)

Hide vector width using scalar threads



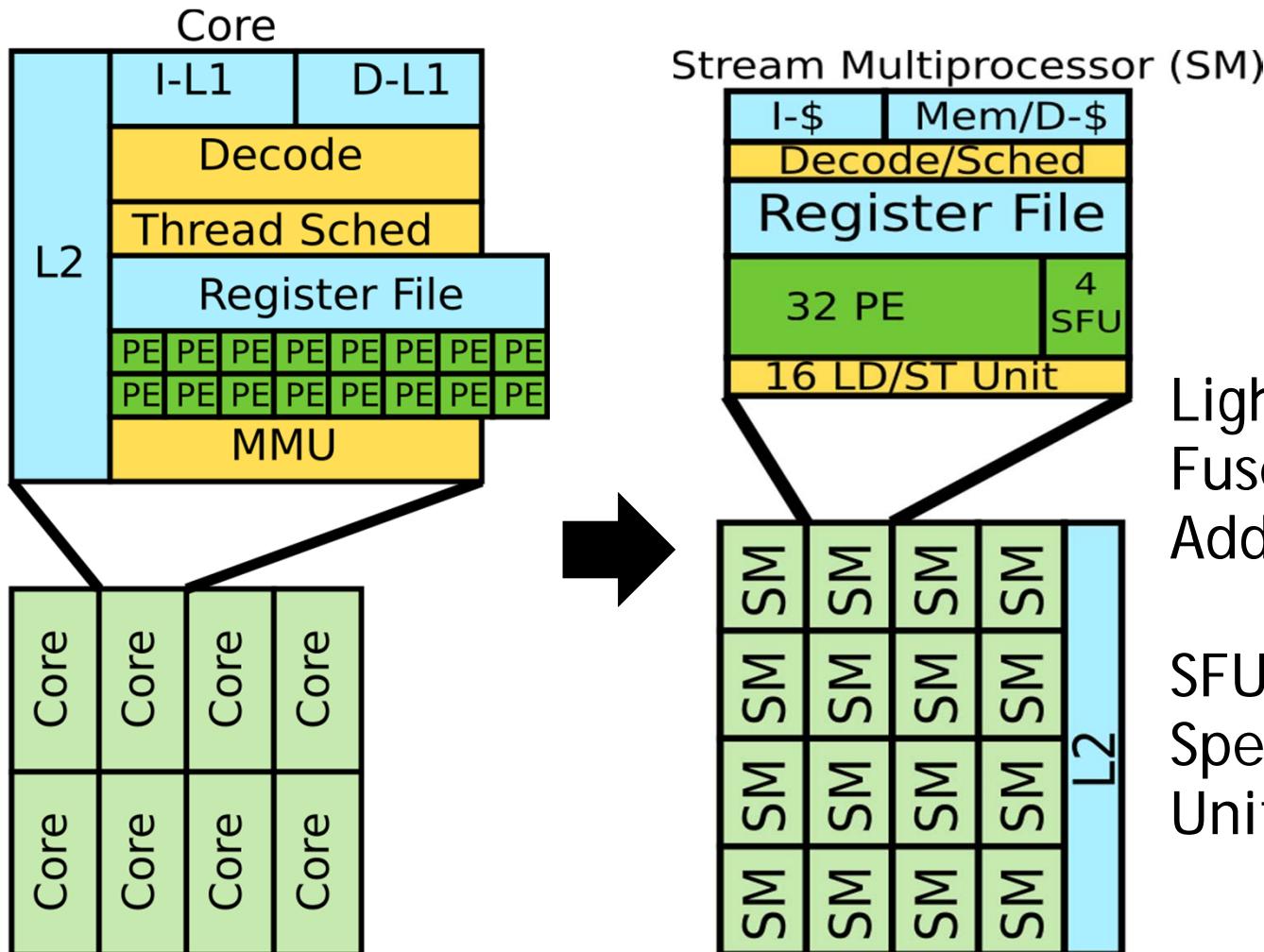
Example of SIMT Execution

- Assume 32 threads are grouped into one warp



Step 3: Simple Core

The Stream Multiprocessor (SM) is a light weight core compared to IA core

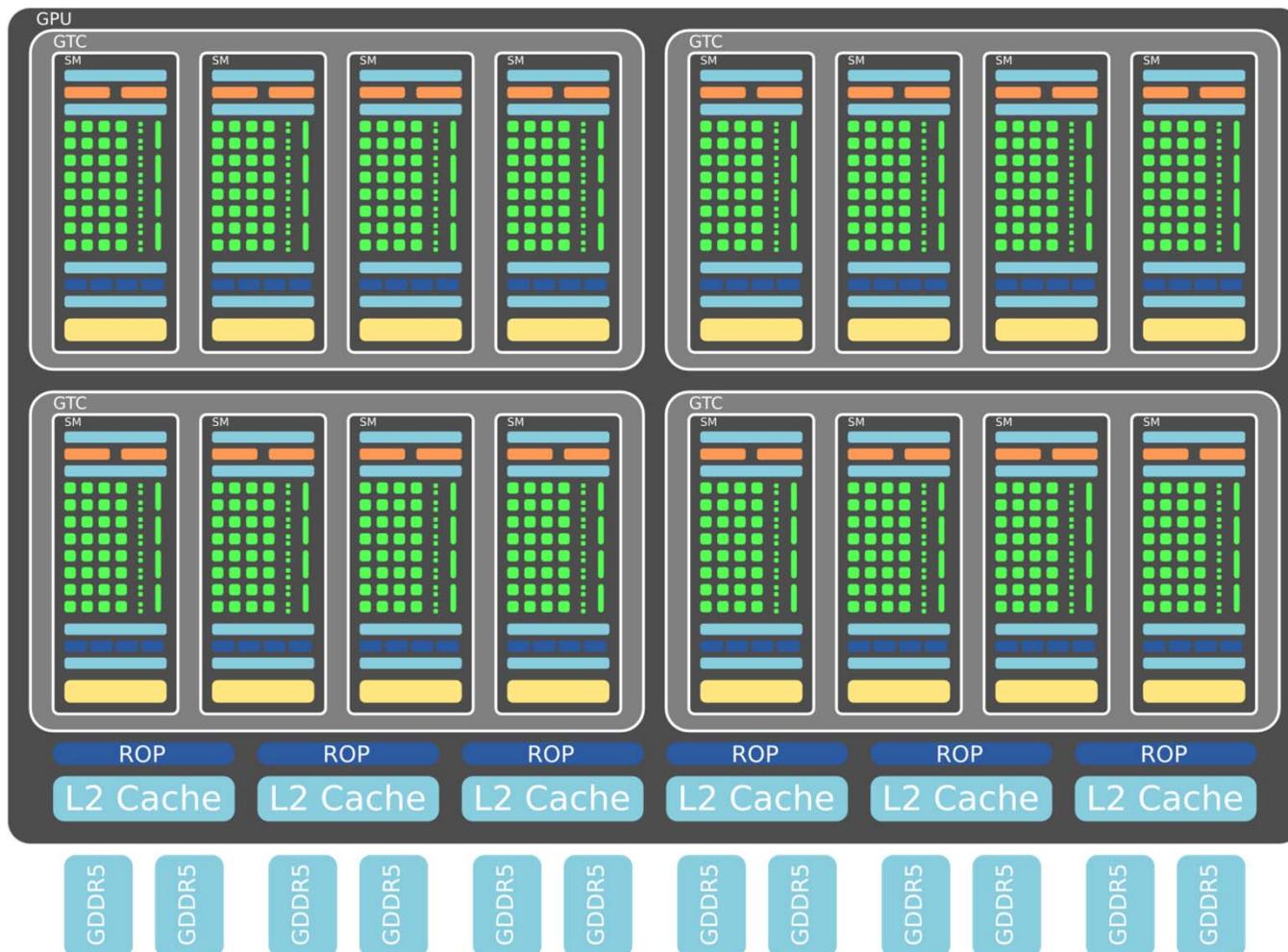


Light weight PE:
Fused Multiply
Add (FMA)

SFU:
Special Function
Unit

NVIDIA Fermi

- 512 Processing Elements (PEs)

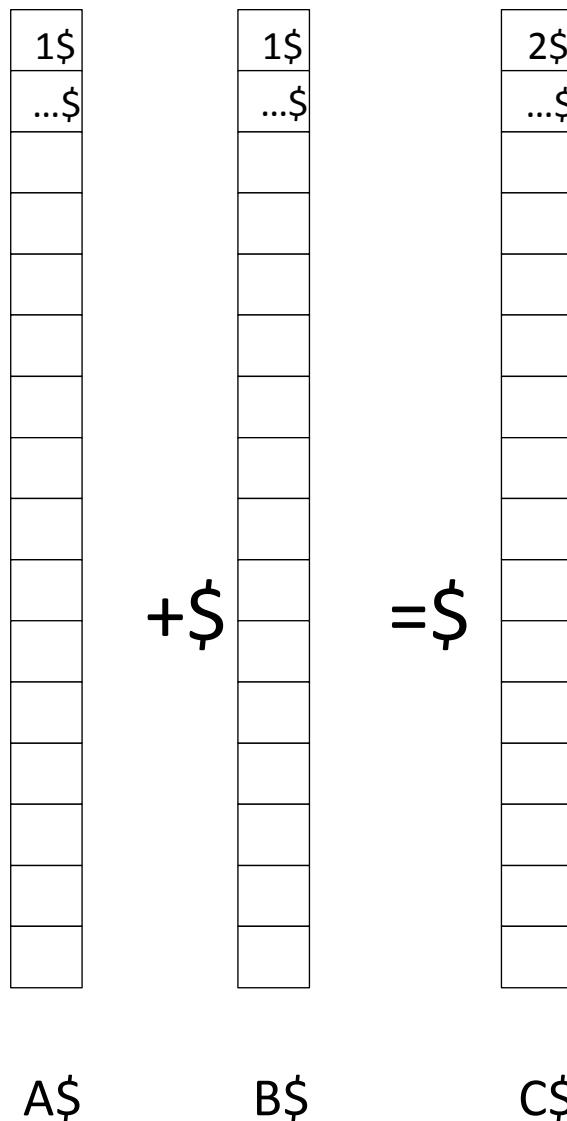


Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ❖ Built-in Data Types and Functions
 - ❖ Thread Model
 - ❖ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



Threading Model: Serial Code

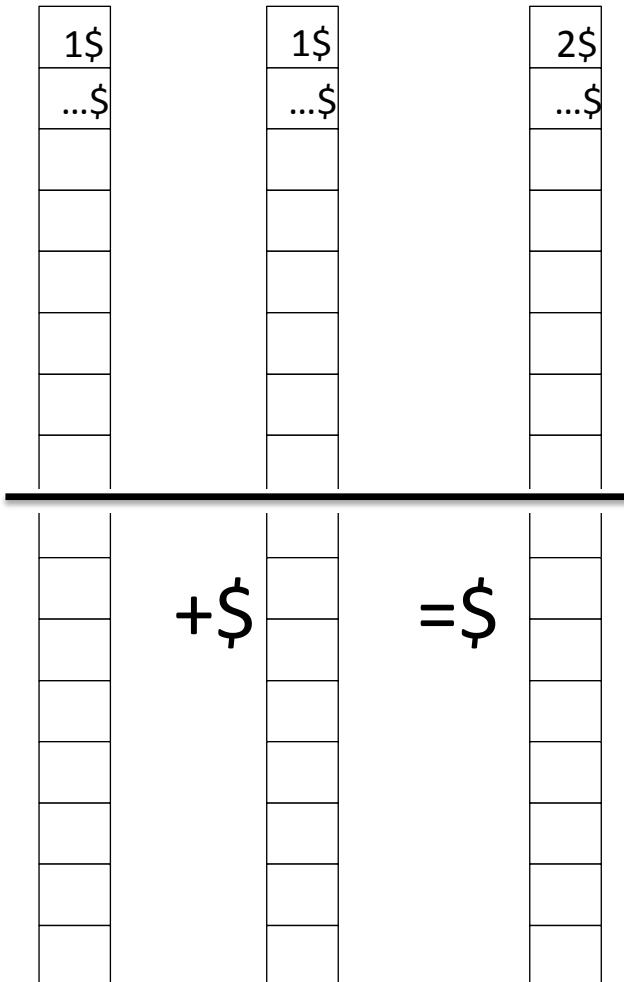


Threads

```
for (int i = 0; i < 1000; ++i)!  
    c[i] = a[i] + b[i];$
```



Threading Model: Dual-core Parallelism (2 Threads)



Thread \$1\$

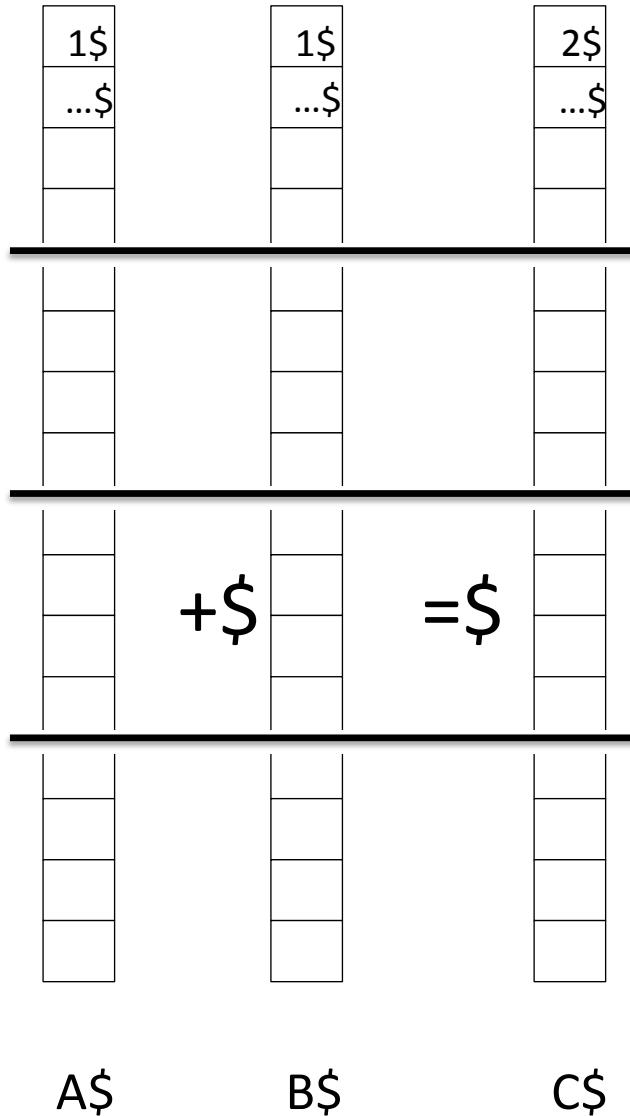
```
for (int i = 0; i < 500; ++i)!  
    c[i] = a[i] + b[i];$
```

Thread \$2\$

```
for (int i = 500; i < 1000; ++i)!  
    c[i] = a[i] + b[i];$
```



Threading Model: Quad-core Parallelism (4 Threads)



Thread\$0\$

```
for (int i = 0; i < 250; ++i)!  
    c[i] = a[i] + b[i];$
```

Thread\$1\$

```
for (int i = 250; i < 500; ++i)!  
    c[i] = a[i] + b[i];$
```

Thread\$2\$

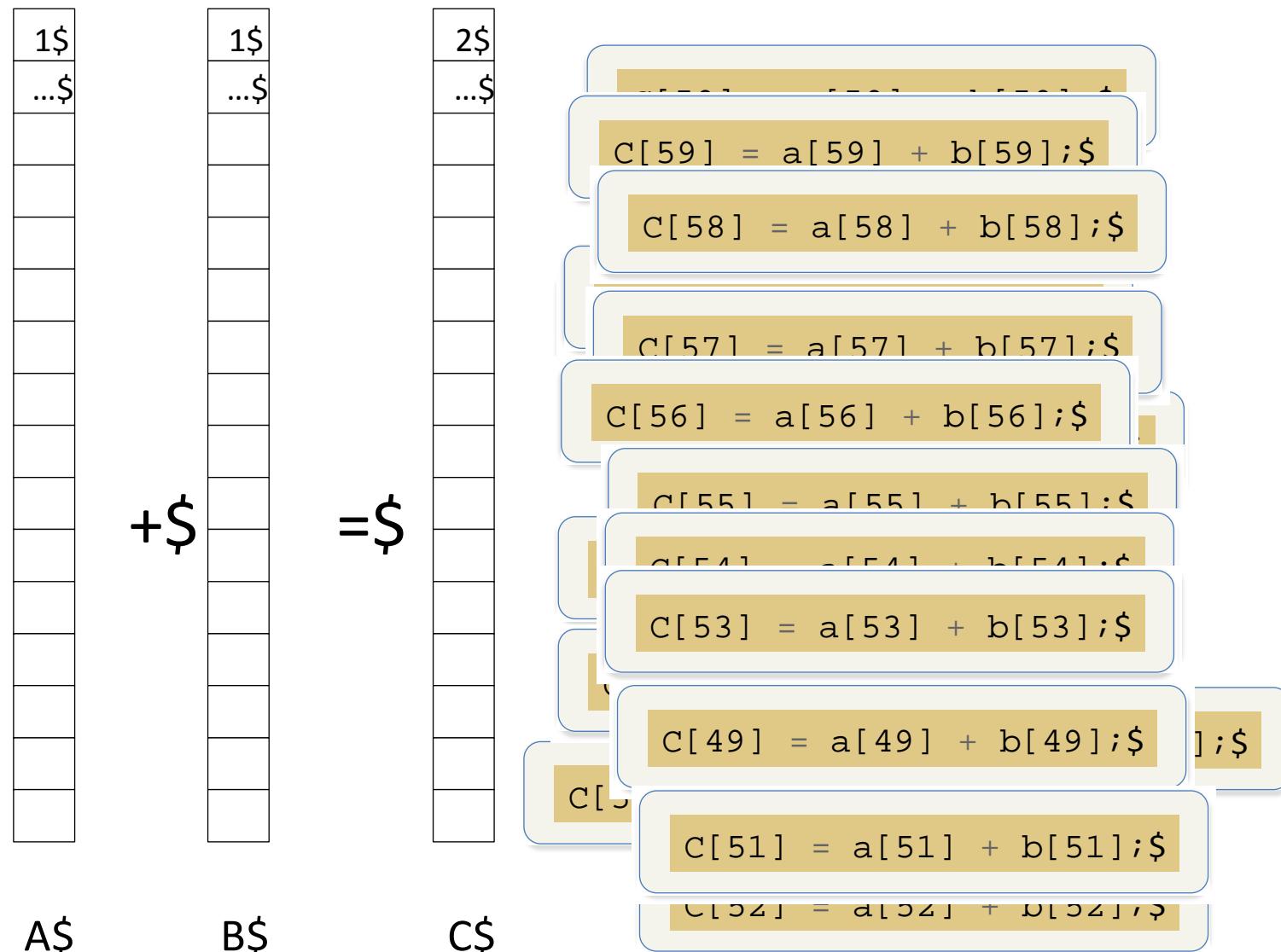
```
for (int i = 500; i < 750; ++i)!  
    c[i] = a[i] + b[i];$
```

Thread\$3\$

```
for (int i = 750; i < 1000; ++i)!  
    c[i] = a[i] + b[i];$
```



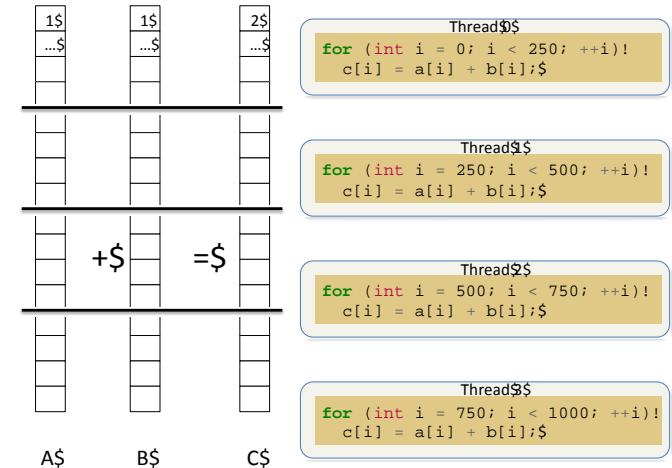
Threading Model: GPU Parallelism (N Threads)



Threading Model

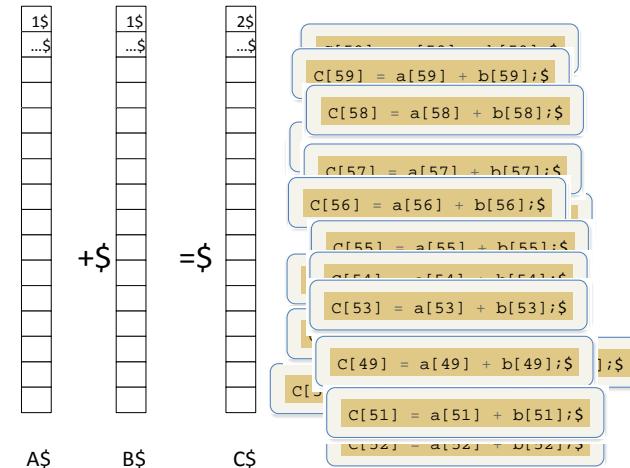
CPU

- Number of Threads
- ~ Physical Cores



GPU

- Number of Threads
- ~ Amount of Data
- Single instruction, multiple threads (SIMT)

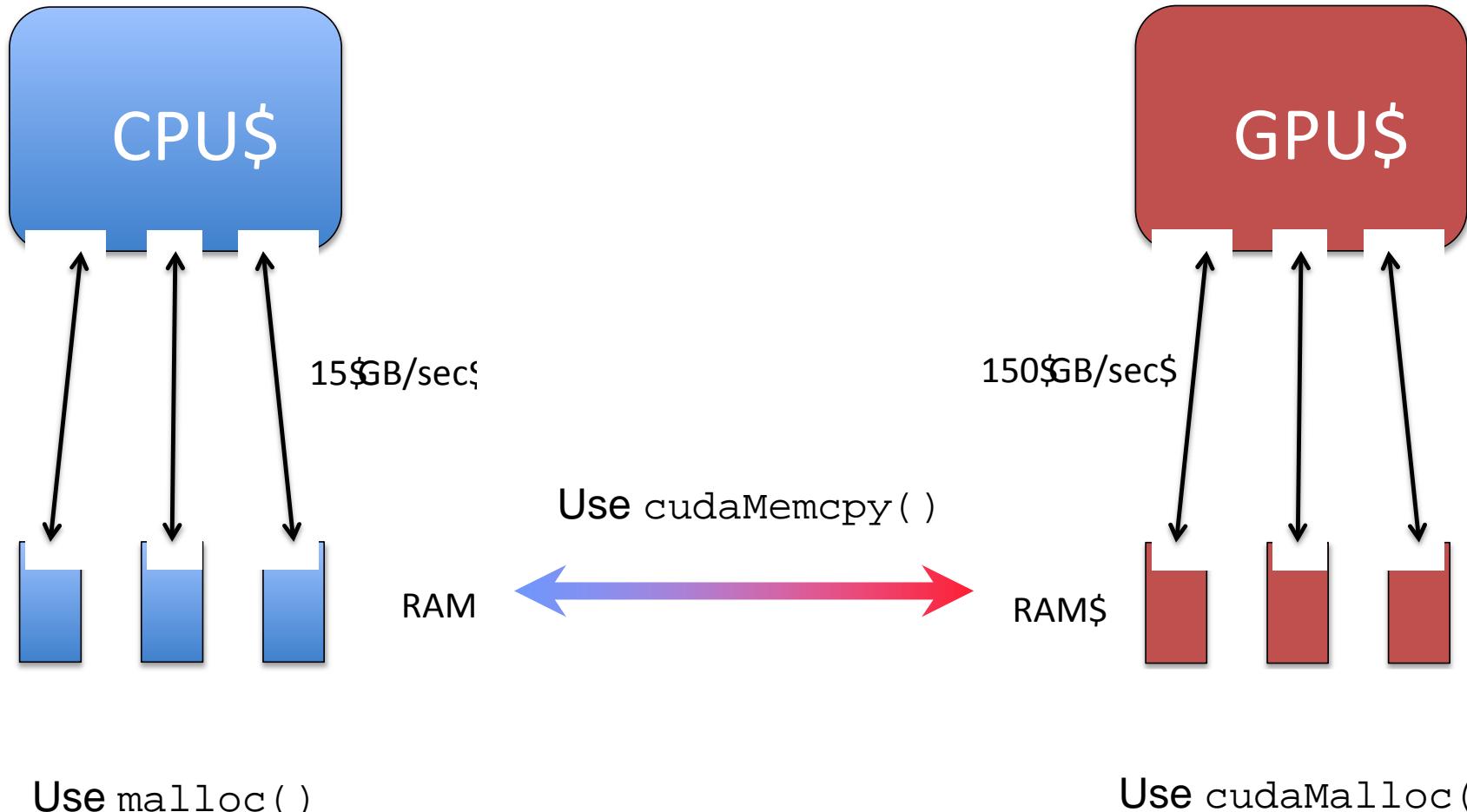


Outline

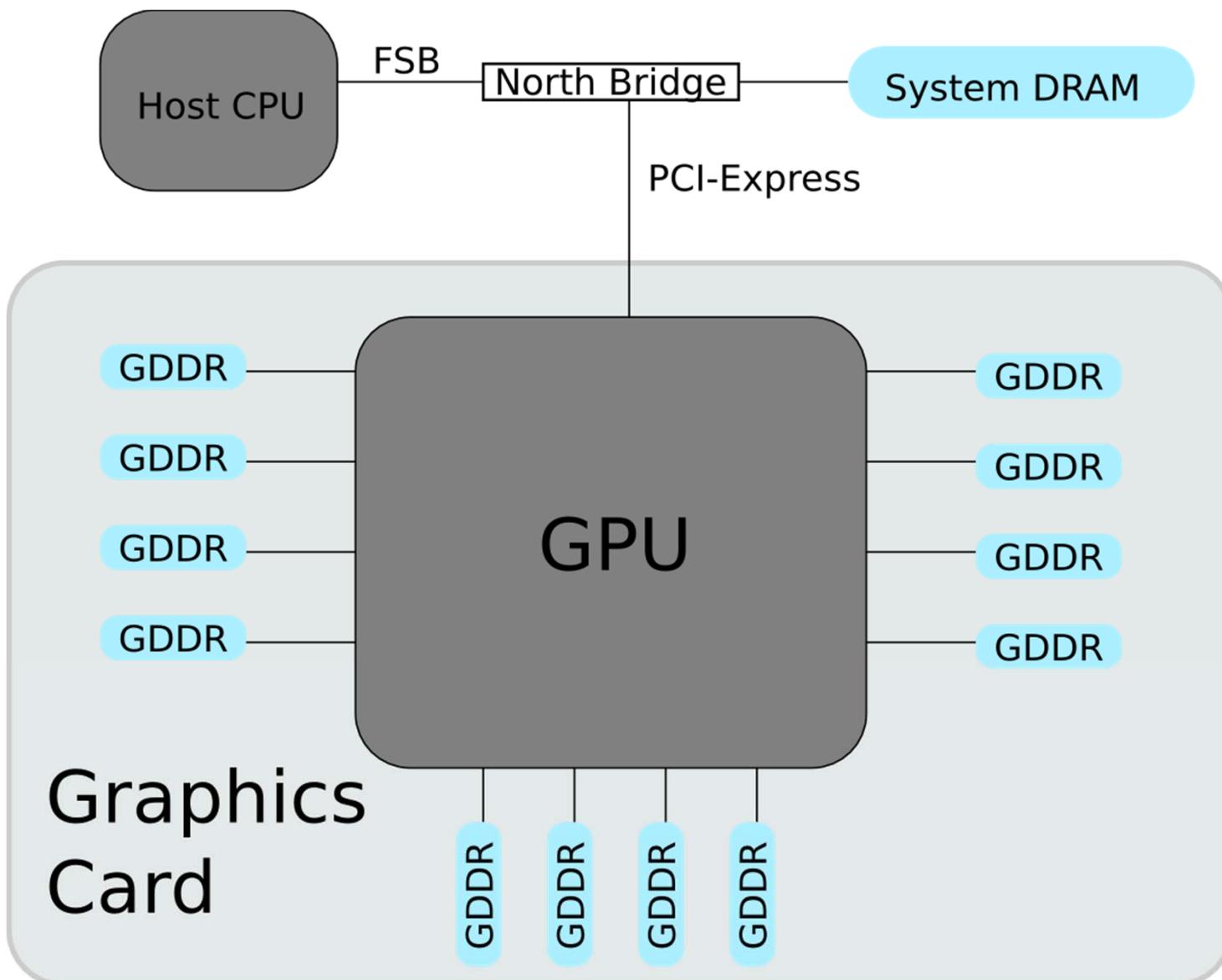
- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ◆ Built-in Data Types and Functions
 - ◆ Thread Model
 - ◆ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



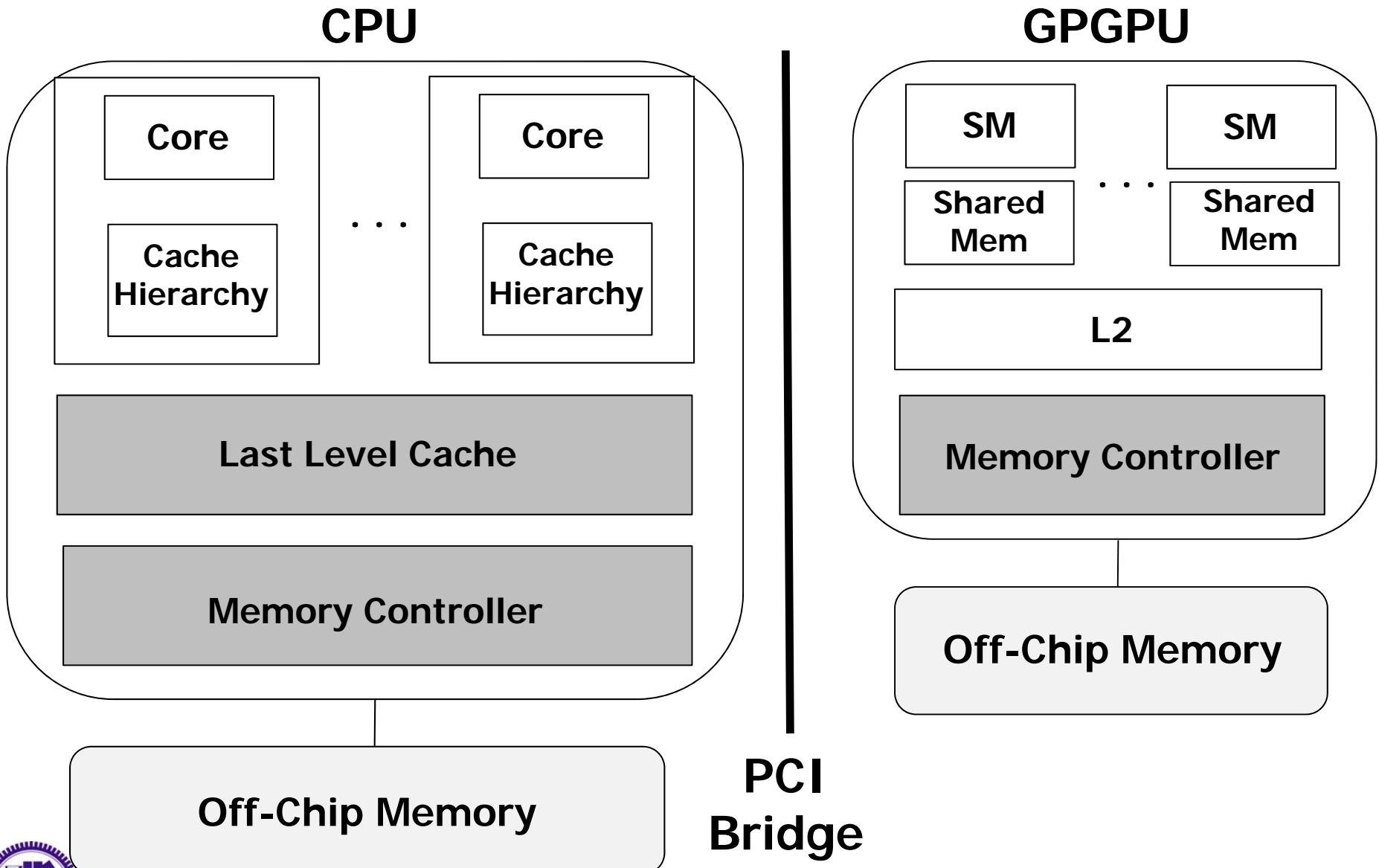
Memory Separation



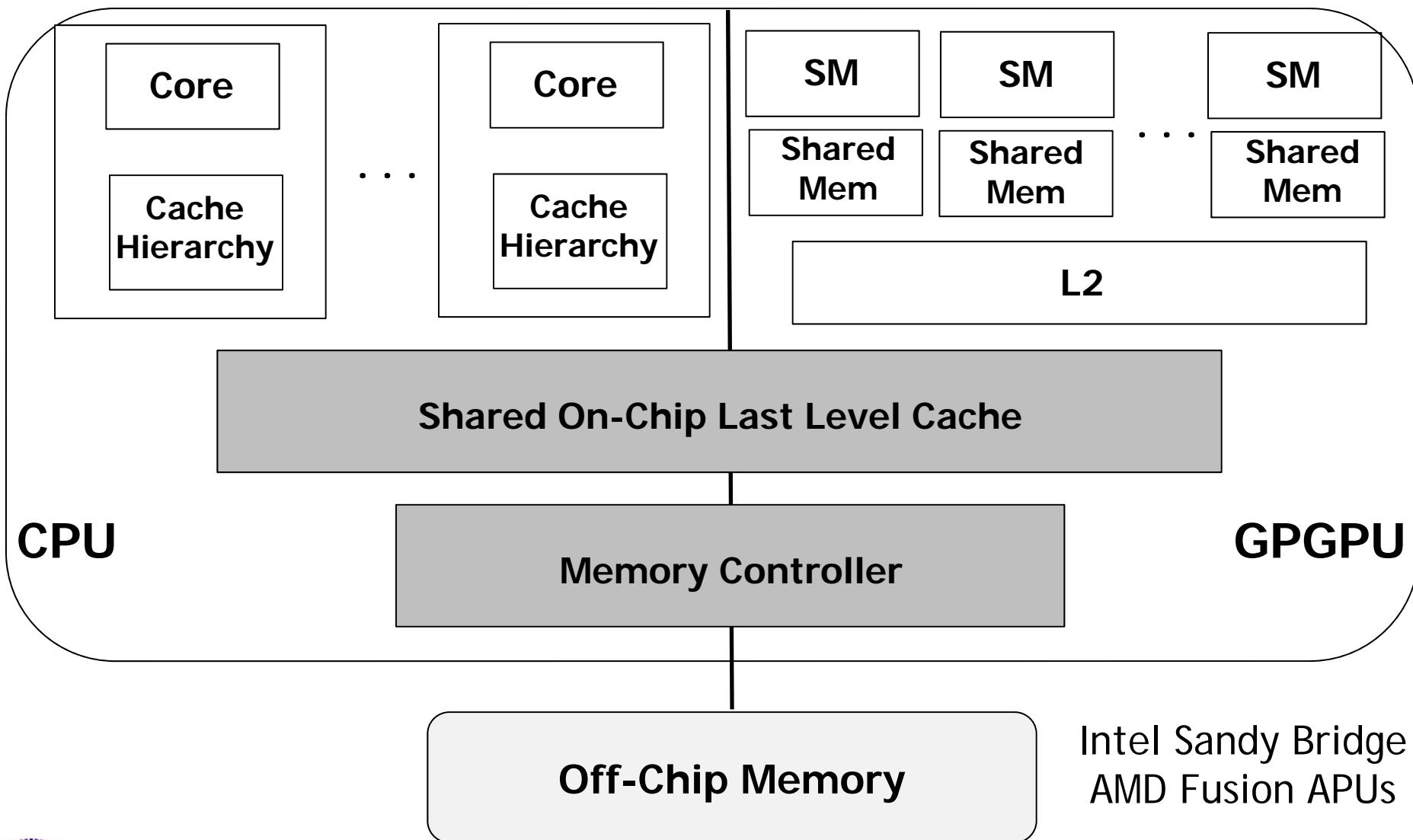
System Architecture



Discrete GPUs



Integrated CPUs-GPUs



Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- **Introduction to CUDA**
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ❖ Built-in Data Types and Functions
 - ❖ Thread Model
 - ❖ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



CUDA

- Compute Unified Device Architecture
- In November 2006, NVIDIA introduced CUDA™, a general-purpose **parallel computing platform and programming model**
- Leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU



GPU as a General-Purpose Computing Platform

- Speedups are impressive and ever increasing!



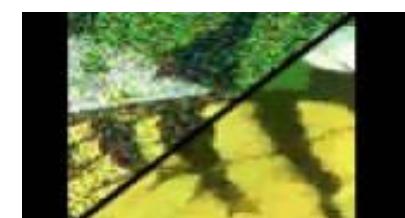
Genetic Algorithm
2600 X



Real Time Elimination
of Undersampling Artifacts
2300 X



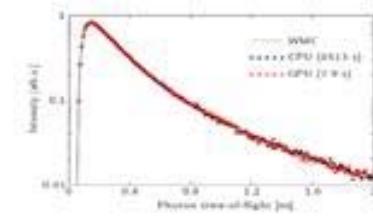
Lattice-Boltzmann Method
for Numerical Fluid Mechanics
1840 X



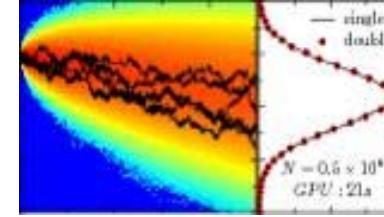
Total Variation Modeling
1000 X



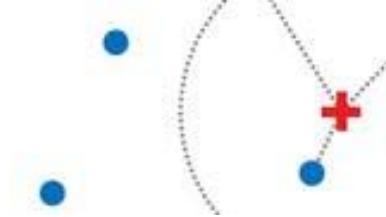
Fast Total Variation for
Computer Vision
1000 X



Monte Carlo Simulation
Of Photon Migration
1000 X



Stochastic Differential
Equations
675 X



K-Nearest Neighbor
Search
470 X

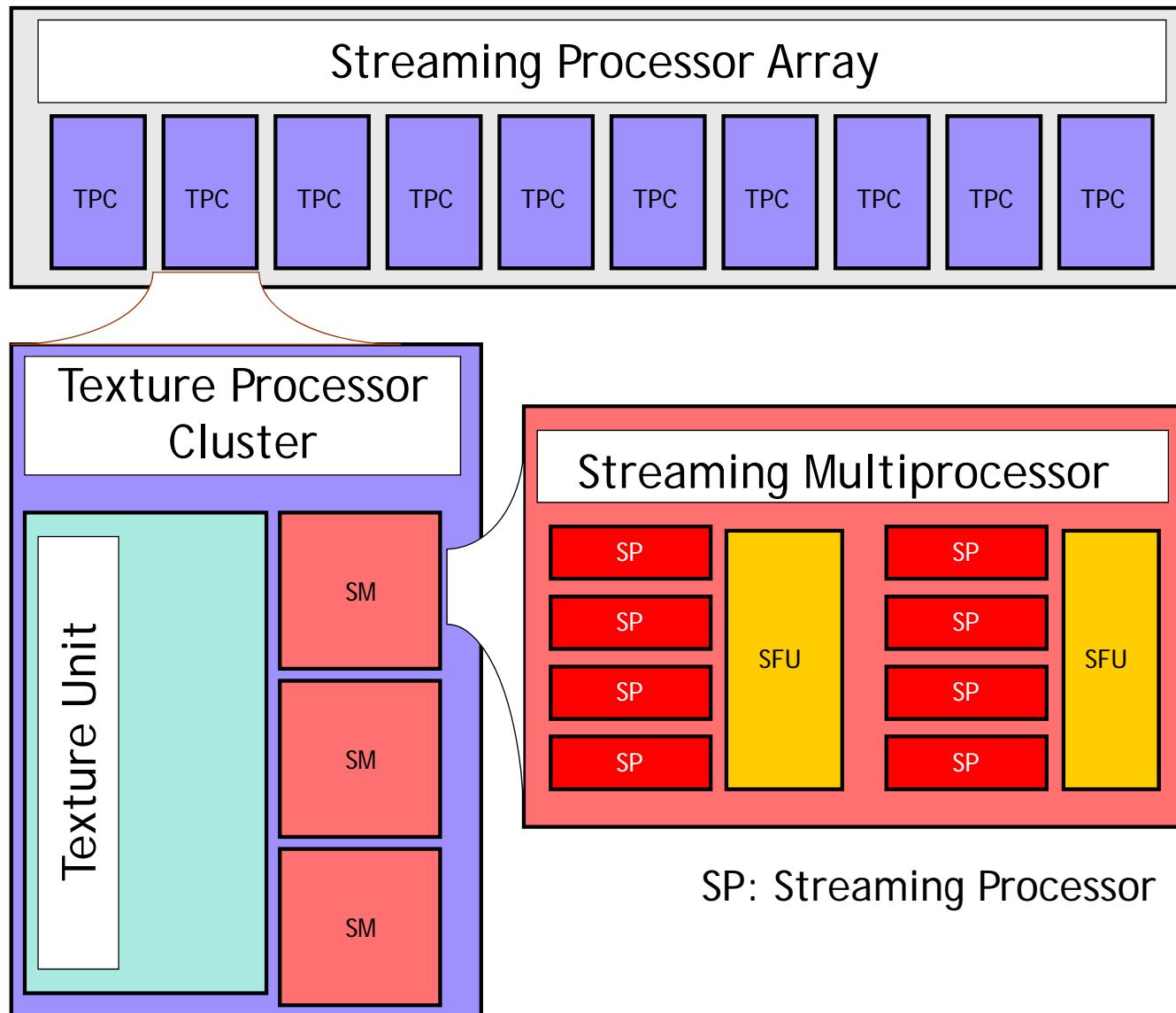
Source: CUDA Zone at www.nvidia.com/cuda/

Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ◆ Built-in Data Types and Functions
 - ◆ Thread Model
 - ◆ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization

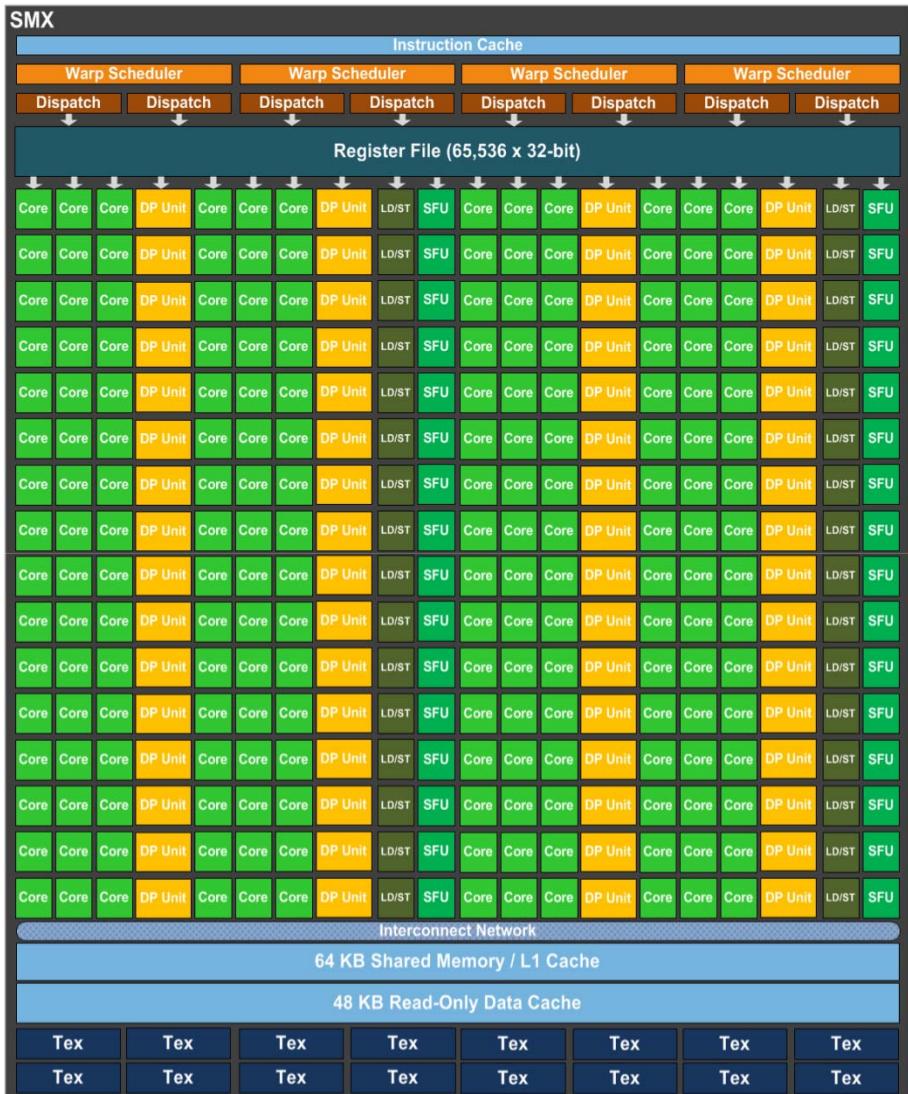


CUDA: Parallel Computing Platform



Kepler GK110

- New streaming multiprocessor (now called SMX)
 - ✚ Contains 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



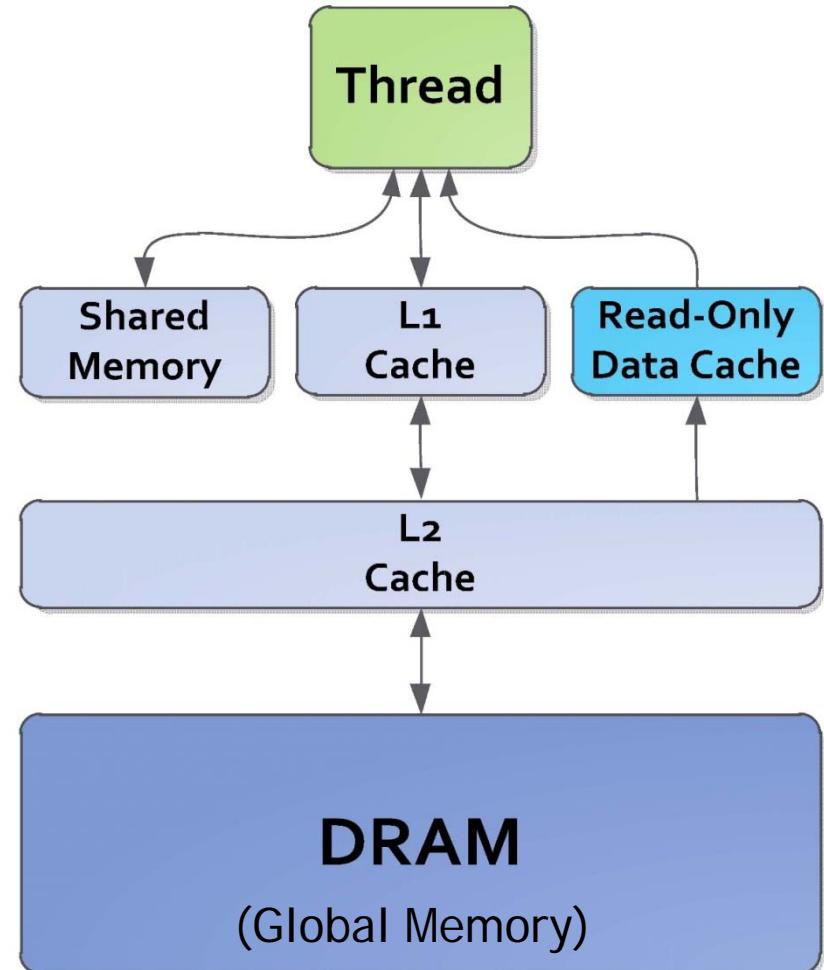
Source: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>



Kepler Memory Hierarchy

- Shared memory/L1 cache split
- Each SMX has 64 KB on-chip memory, that can be configured as:
 - ⊕ 48 KB of shared memory with 16 KB of L1 cache,
 - ⊕ 16 KB of shared memory with 48 KB of L1 cache, or
 - ⊕ (new) a 32KB / 32KB split between shared memory and L1 cache.

Kepler Memory Hierarchy



Source: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>

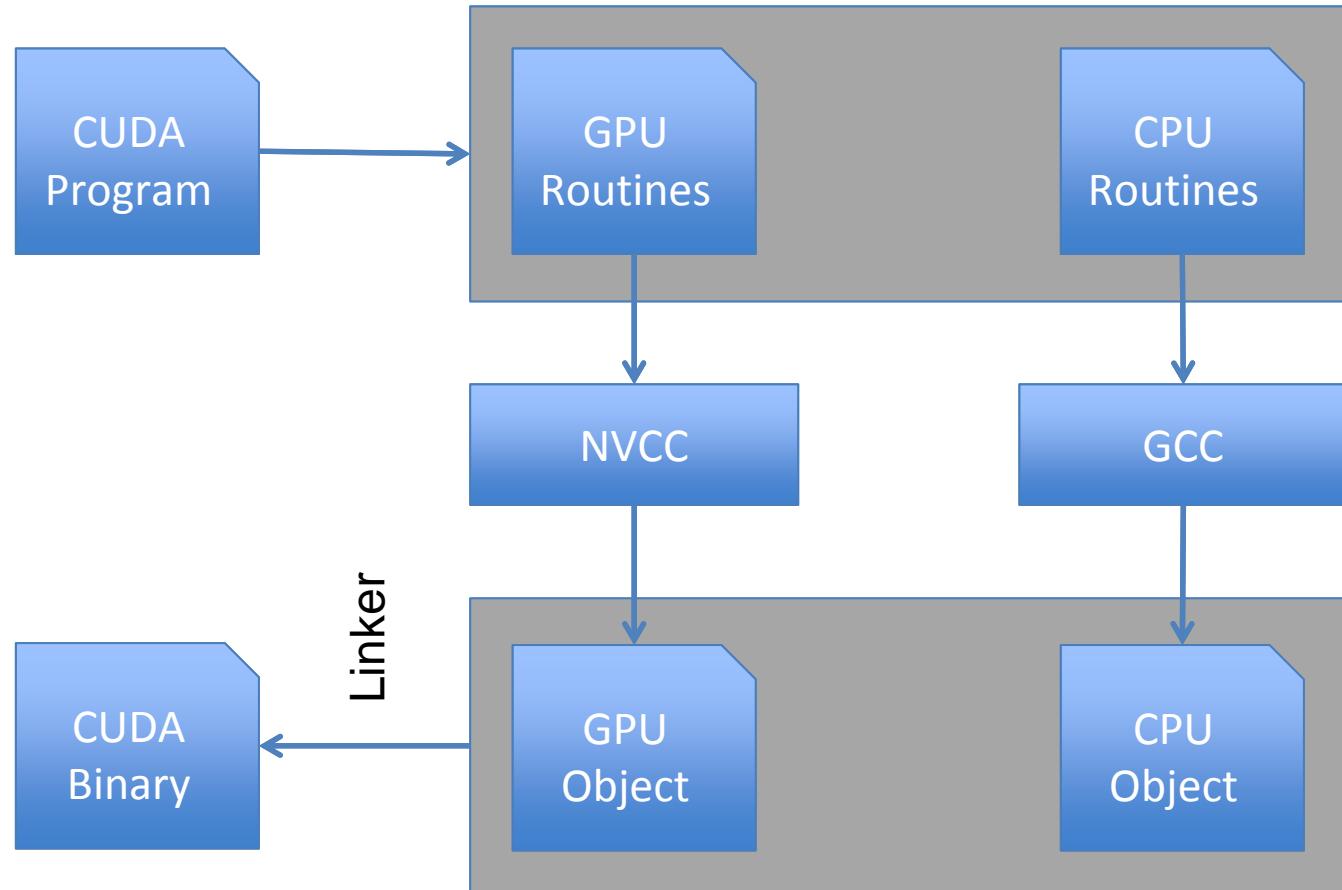


Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ◆ Built-in Data Types and Functions
 - ◆ Thread Model
 - ◆ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



Heterogeneous Programming: CPU/GPU Code



CUDA Program: An Example

mycode.cu

```
int main_data;  
__shared__ int sdata;  
  
Main() { }  
__host__ hfunc () {  
    int hdata;  
    gfunc<<<gs,bs>>>();  
}  
  
__global__ gfunc() {  
    int gdata;  
    dfunct();  
}  
  
__device__ dfunc() {  
    int ddata;  
}
```

Host Only
Interface
Device Only

Compiled by native compiler: gcc, icc, cc

```
int main_data;
```

```
Main() {}  
__host__ hfunc () {  
    int hdata;  
    gfunc<<<gs,bz>>>();  
}
```

Compiled by nvcc compiler

```
__shared__ sdata;
```

```
__global__ gfunc() {  
    int gdata;  
    dfunc();  
}
```

```
__device__ dfunc() {  
    int ddata;  
}
```



CUDA: Minimal Extensions to C + API

■ Declaration specifiers

- ⊕ global, device, shared.

local, constant

```
__device__ float filter[N];
```

■ Keywords

- ⊕ threadIdx, blockIdx

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

■ Runtime API

- ⊕ Memory, symbol, execution management

```
int main() {
    // Allocate GPU memory
    void *myimage = cudaMalloc(...);

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



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



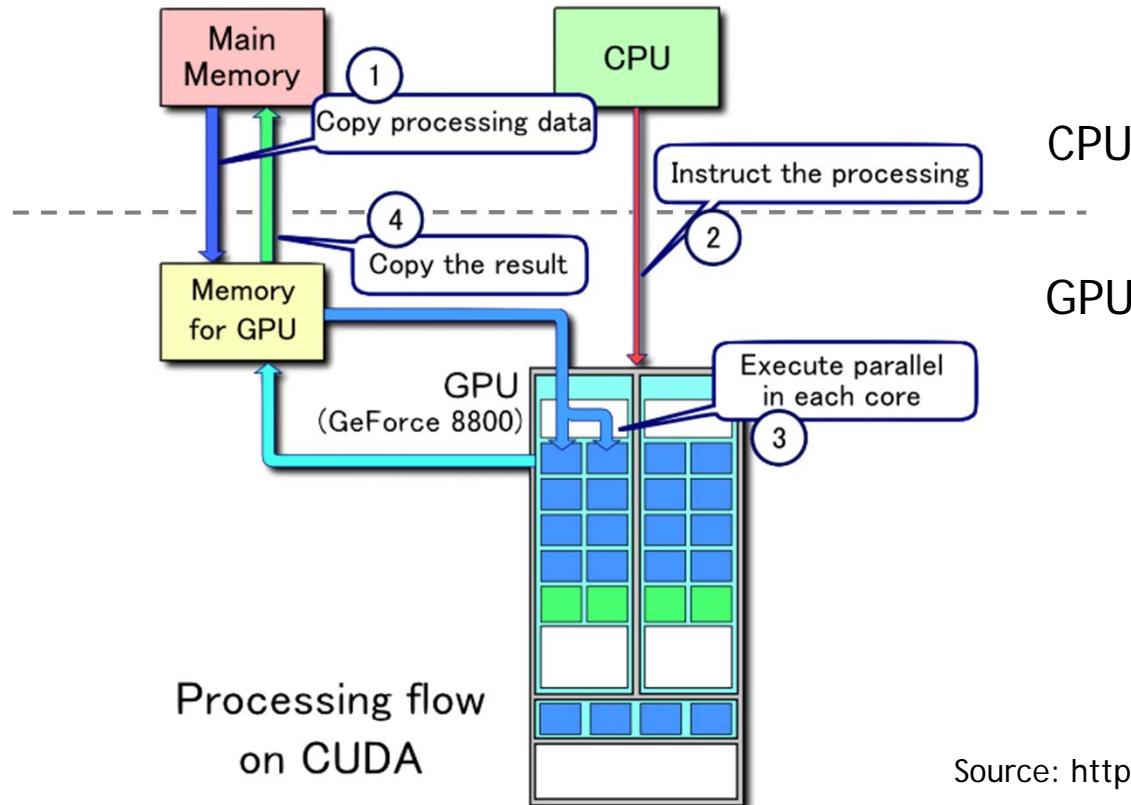
What Programmer Expresses in CUDA

- Computation partitioning (where does computation occur?)
 - ⊕ Declarations on functions `__host__`, `__global__`, `__device__`
 - ⊕ Mapping of thread programs to device:
`compute <<<gs, bs>>>(<args>)`
- Data partitioning (where does data reside, who may access it and how?)
 - ⊕ Declarations on data `__shared__`, `__device__`, `__constant__`, ...
- Data management and orchestration
 - ⊕ Copying to/from host: *e.g.*,
`cudaMemcpy(h_obj, d_obj, cudaMemcpyDeviceToHost)`
- Concurrency management
 - ⊕ *E.g.* `__syncthreads()`



Processing Flow on CUDA

1. Copy data from main mem to GPU mem
2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy the result from GPU mem to main mem



Source: <http://en.wikipedia.org/wiki/CUDA>

CUDA: A Scalable Programming Model

- A low learning curve for programmers familiar with standard programming languages such as C
- Three key abstractions
 - 1) A hierarchy of thread groups
 - 2) Shared memories
 - 3) Barrier synchronization



CUDA Terminology

- ***Host*** - typically the CPU
 - ◆ Code written in ANSI C
 - ◆ Written in *extended* ANSI C when interacting with devices
- ***Device*** - typically the GPU (data-parallel)
 - ◆ Code written in *extended* ANSI C
- Host and device have separate memories
- CUDA Program
 - ◆ Contains both host and device code



CUDA Terminology (Cont'd)

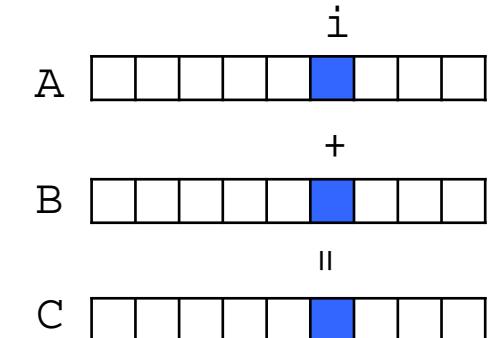
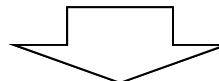
- *Kernel* - data-parallel function
 - ◆ Invoking a kernel creates lightweight threads on the device
 - ◆ Threads are generated and scheduled with hardware
- Similar to a *shader* in OpenGL



CUDA Kernels

- Executed N times in parallel by N different *CUDA threads*

```
for (int i = 0; i < N; i++)  
    C[i] = A[i] + B[i];
```



*Declaration
Specifier*

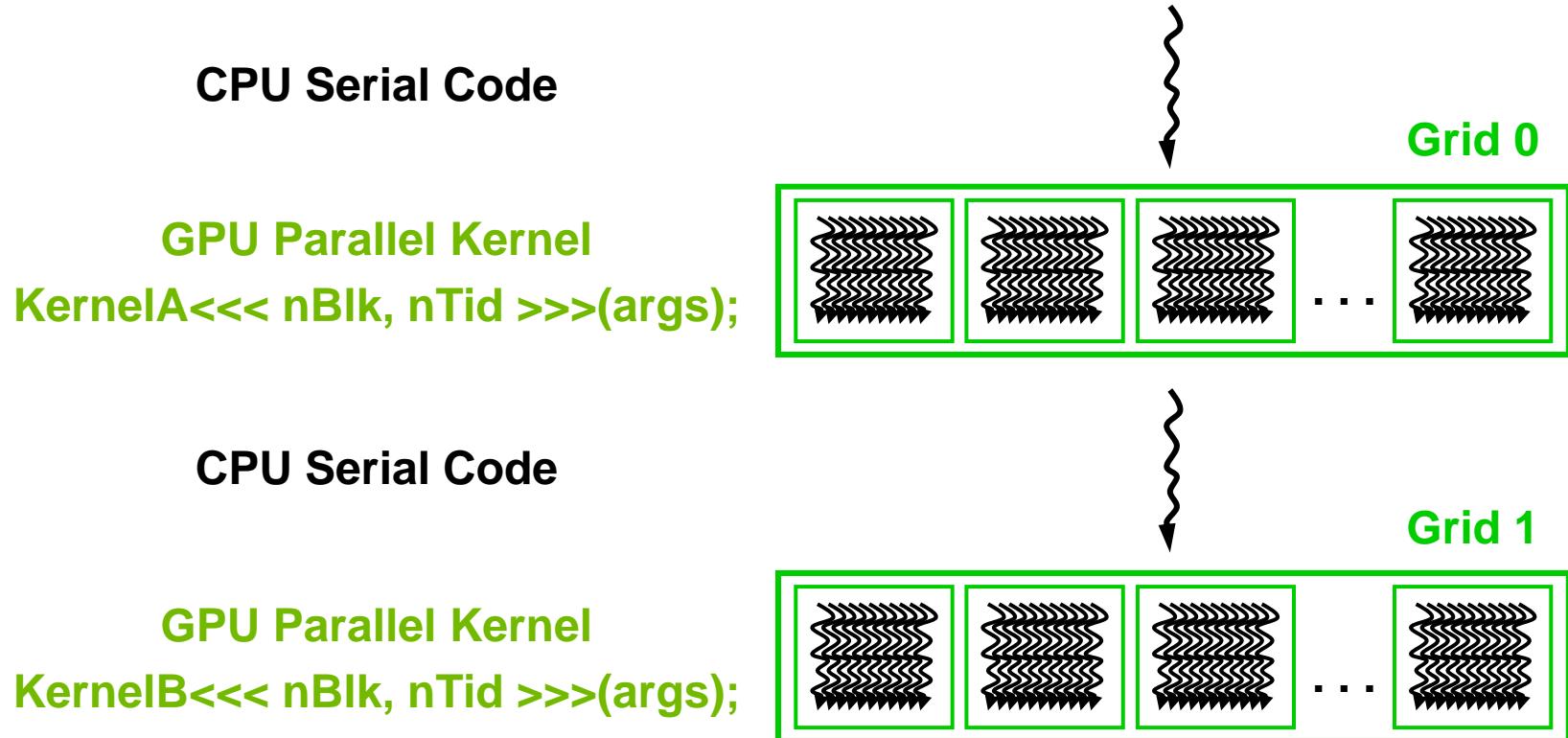
```
// Kernel definition  
__global__ void VecAdd(float* A, float* B, float* C)  
{  
    int i = threadIdx.x;  
    C[i] = A[i] + B[i];  
}  
  
int main()  
{  
    ...  
    // Kernel invocation with N threads  
    VecAdd<<<1, N>>>(A, B, C);  
}
```

Thread ID

*Execution
Configuration*



CUDA Program Execution



Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ❖ Built-in Data Types and Functions
 - ❖ Thread Model
 - ❖ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



Functional Declarations

	Executed on the:	Only callable from the:
<code>__global__ void KernelFunc()</code>	device	host
<code>__device__ float DeviceFunc()</code>	device	device
<code>__host__ float HostFunc()</code>	host	host

- `__global__`
 - Must return `void`
- `__device__`
 - Inlined by default



Functional Declarations

- What do these do?

~~■ __global__ __host__ void func()~~

~~■ __device__ __host__ void func()~~



Functional Declarations

■ What do these do?

- **`__global__ __host__ void func()`**
- **`__device__ __host__ void func()`**

```
__host__ __device__ func()
{
    #if __CUDA_ARCH__ == 100
        // Device code path for compute capability 1.0
    #elif __CUDA_ARCH__ == 200
        // Device code path for compute capability 2.0
    #elif !defined(__CUDA_ARCH__)
        // Host code path
    #endif
}
```



Functional Declarations

- Global and device functions
 - No recursion (except Fermi)
 - No static variables
 - No malloc()
 - Careful with function calls through pointers



Vector Types

- `char[1-4], uchar[1-4]`
- `short[1-4], ushort[1-4]`
- `int[1-4], uint[1-4]`
- `long[1-4], ulong[1-4]`
- `longlong[1-4], ulonglong[1-4]`
- `float[1-4]`
- `double1, double2`



Vector Types

- Available in host and device code
- Construct with `make_<type name>`

```
int2 i2 = make_int2(1, 2);  
float4 f4 = make_float4(  
    1.0f, 2.0f, 3.0f, 4.0f);
```



Vector Types

- Access with .x, .y, .z, and .w

```
int2 i2 = make_int2(1, 2);  
int x = i2.x;  
int y = i2.y;
```

- No .r, .g, .b, .a, etc. like GLSL



Math Functions

- **double** and **float** overloads
 - ⊕ No vector overloads
- On the host, functions use the C runtime implementation if available



Math Functions

■ Partial list:

- ◆ `sqrt, rsqrt`
- ◆ `exp, log`
- ◆ `sin, cos, tan, sincos`
- ◆ `asin, acos, atan2`
- ◆ `trunc, ceil, floor`



Math Functions

■ *Intrinsic* function

- ⊕ Device only
- ⊕ Faster, but less accurate
- ⊕ Prefixed with _
- ⊕ _exp, _log , _sin , _pow , ...



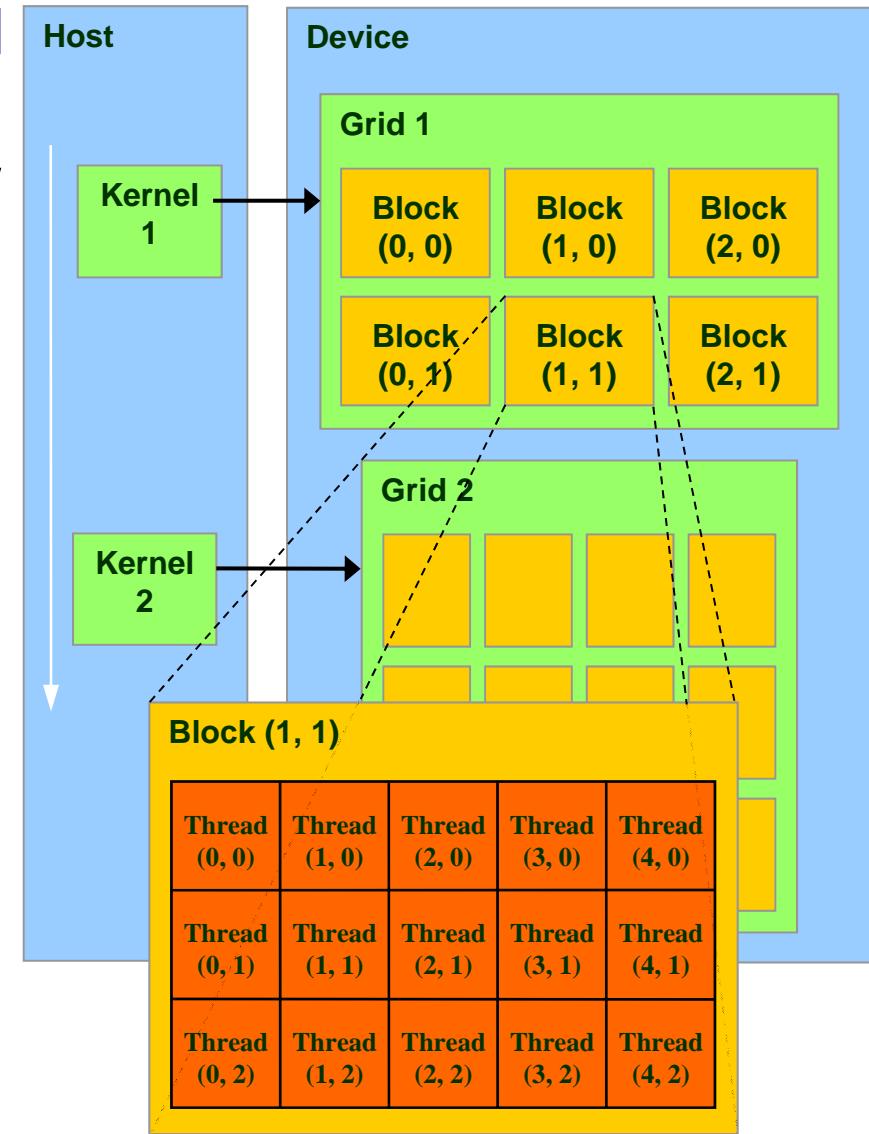
Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ◆ Built-in Data Types and Functions
 - ◆ **Thread Model**
 - ◆ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



Thread Hierarchies

- A kernel is executed as a grid of thread blocks
 - All threads share data memory space
- A **thread block** is a batch of threads that can cooperate with each other by:
 - Synchronizing their execution
 - For hazard-free shared memory accesses
 - Efficiently sharing data through a low latency **shared memory**
- Two threads from two different blocks cannot cooperate



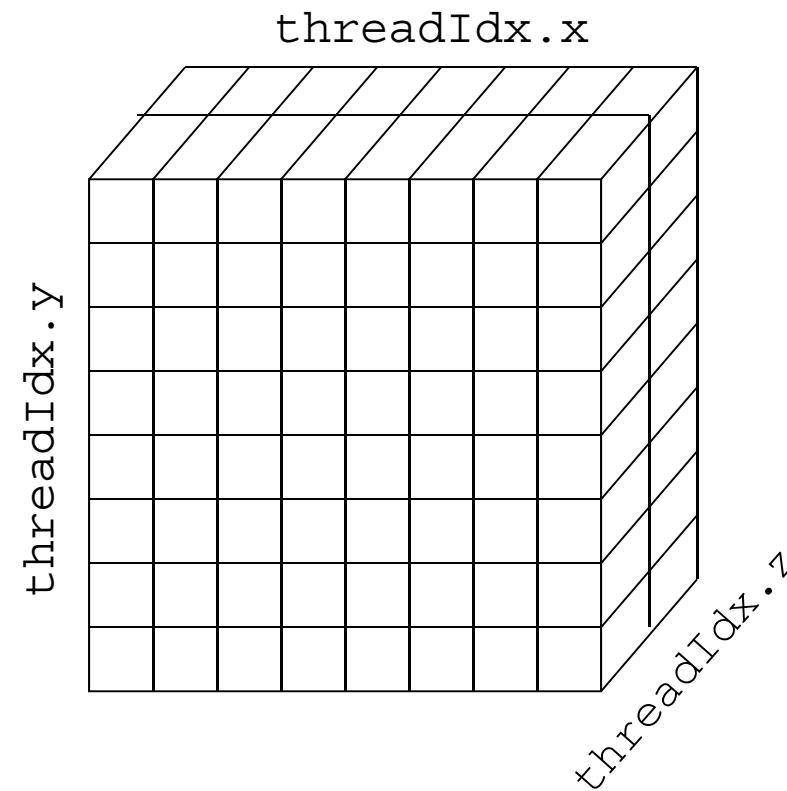
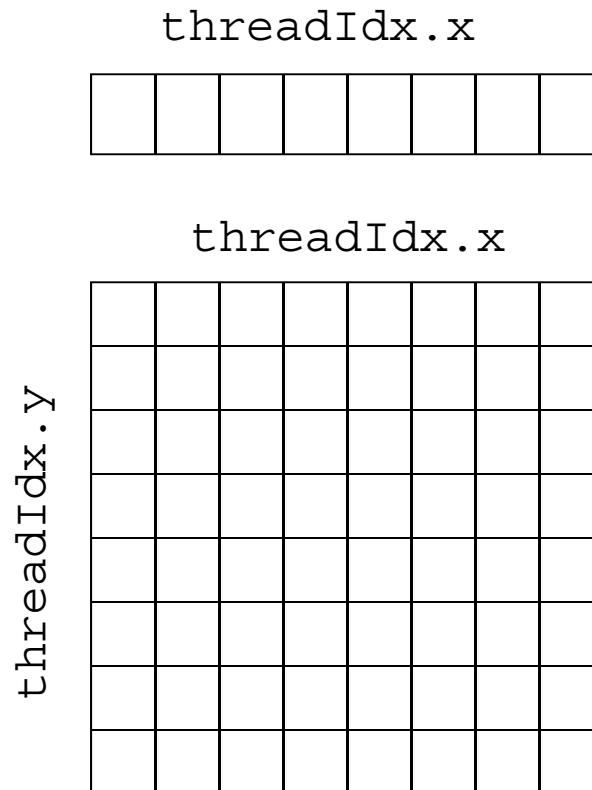
Thread Hierarchies

- *Grid* - one or more thread blocks
 - ◆ 1D or 2D
- *Block* - array of threads
 - ◆ 1D, 2D, or 3D
 - ◆ Each block in a grid has the same number of threads
 - ◆ Each thread in a block can
 - ◆ Synchronize
 - ◆ Access shared memory

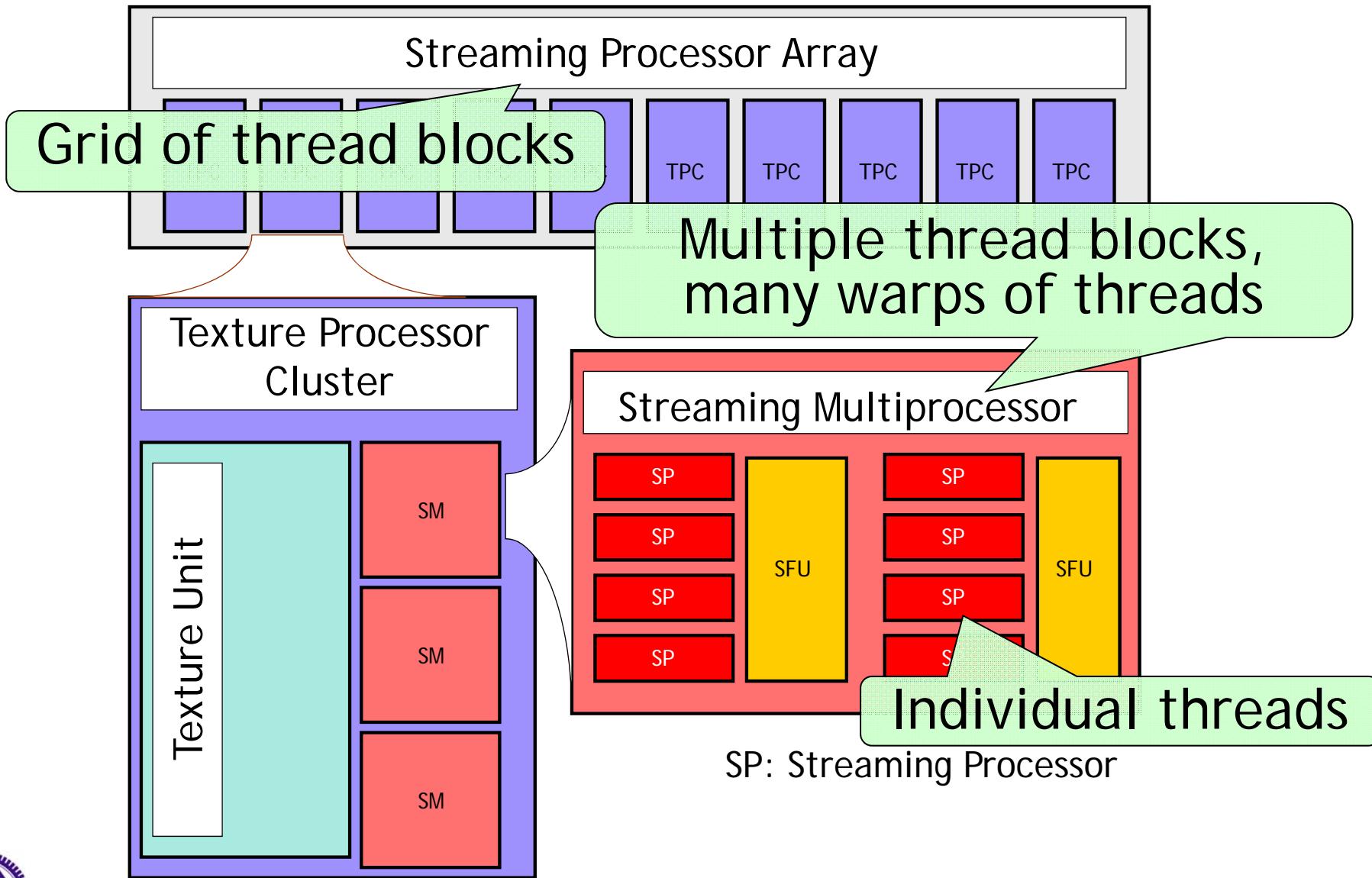


Thread Hierarchies

- **Block** - 1D, 2D, or 3D
 - ✚ Example: Index into vector, matrix, volume



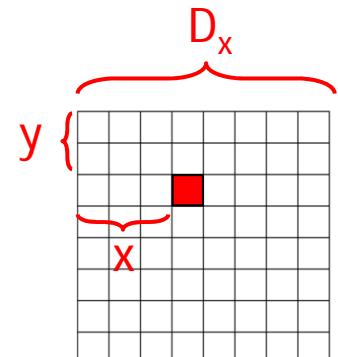
CUDA: Parallel Computing Platform



Thread ID

- *Thread ID*: Scalar thread identifier
- Thread Index: `threadIdx`

- 1D: Thread ID == Thread Index
- 2D with size (D_x, D_y)
 - ⊕ Thread ID of index $(x, y) == x + y D_x$
- 3D with size (D_x, D_y, D_z)
 - ⊕ Thread ID of index $(x, y, z) == x + y D_x + z D_x D_y$

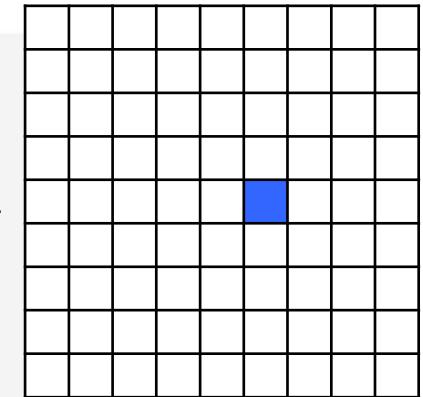


Thread ID

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        C[i][j] = A[i][j] + B[i][j];
```

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int j = threadIdx.x;
    int i = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<numBlocks, threadsPerBlock>>(A, B, C);
}
```



2D Index

1 Thread Block

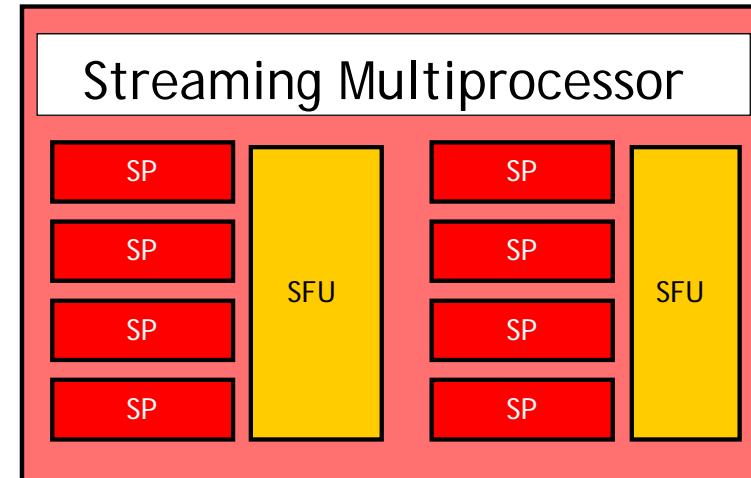
2D Block



Thread Hierarchies

Thread Block

- ⊕ Group of threads
 - ◆ G80 and GT200: Up to 512 threads
 - ◆ Fermi and Kepler: Up to 1024 threads
- ⊕ Reside on same SM
- ⊕ Share memory of that core



Block ID

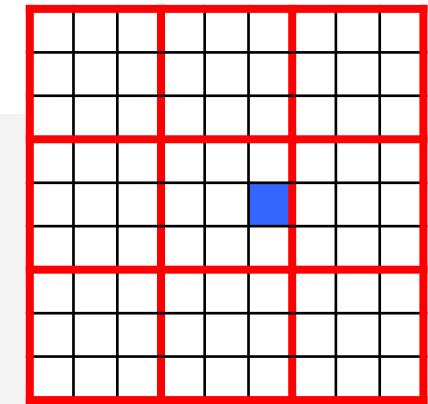
- Block Index: `blockIdx`
- Dimension: `blockDim`
 - ❖ 1D or 2D



Thread Hierarchies

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```



16x16
Threads per block

2D Thread Block



Thread Hierarchies

■ Example: N = 32

- ⊕ 16x16 threads per block (independent of N)
 - ◆ `threadIdx` ([0, 15], [0, 15])
- ⊕ 2x2 thread blocks in grid
 - ◆ `blockIdx` ([0, 1], [0, 1])
 - ◆ `blockDim` = 16

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

■ $i = [0, 1] * 16 + [0, 15]$



Thread Hierarchies

- Blocks execute independently
 - ✚ Each thread block is assigned to an SM to execute
 - ✚ In any order:
 - ◆ parallel (interleaved or not) or
 - ◆ series
 - ✚ Scheduled in any order by any number of cores
 - ◆ Allows code to scale with core count



Review: Thread Hierarchies

```
int threadID = blockIdx.x *  
    blockDim.x + threadIdx.x;
```

```
float x = input[threadID];  
float y = func(x);  
output[threadID] = y;
```

Use grid and block position to
compute a thread id



Review: Thread Hierarchies

```
int threadID = blockIdx.x *  
blockDim.x + threadIdx.x;
```

```
float x = input[threadID];
```

```
float y = func(x);
```

```
output[threadID] = y;
```

Use thread id to read from input



Review: Thread Hierarchies

```
int threadID = blockIdx.x *  
    blockDim.x + threadIdx.x;  
float x = input[threadID];
```

```
float y = func(x);
```

```
output[threadID] = y;
```

Run function on input: data-parallel!



Review: Thread Hierarchies

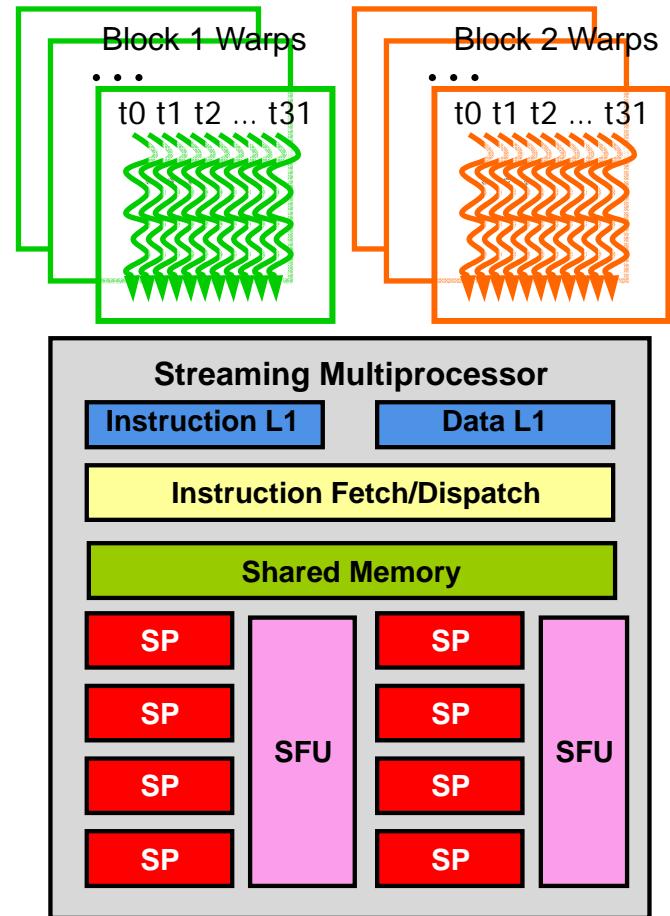
```
int threadID = blockIdx.x *  
    blockDim.x + threadIdx.x;  
float x = input[threadID];  
float y = func(x);  
output[threadID] = y;
```

Use thread id to output result



Scheduling Threads

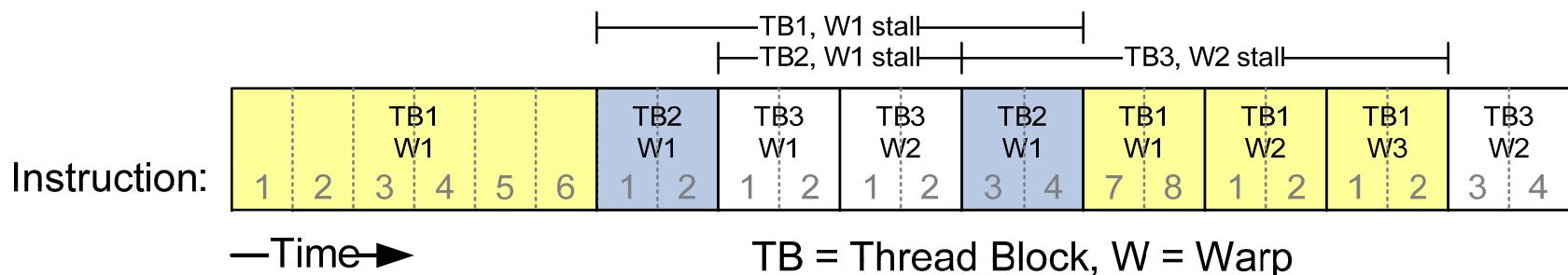
- Each Thread Block is divided in 32-thread **Warp**s
 - ◆ This is an implementation decision, not part of the CUDA programming model
- Warps are the basic scheduling units in SM
- Example (draws on figure at right):
 - ◆ Assume 2 Blocks are processed by an SM and each Block has 512 threads, how many Warps are managed by the SM?
 - ◆ Each Block is divided into $512/32 = 16$ Warps
 - ◆ There are $16 * 2 = 32$ Warps
 - ◆ At any point in time, only **one** of the 32 Warps will be selected for instruction fetch and execution.



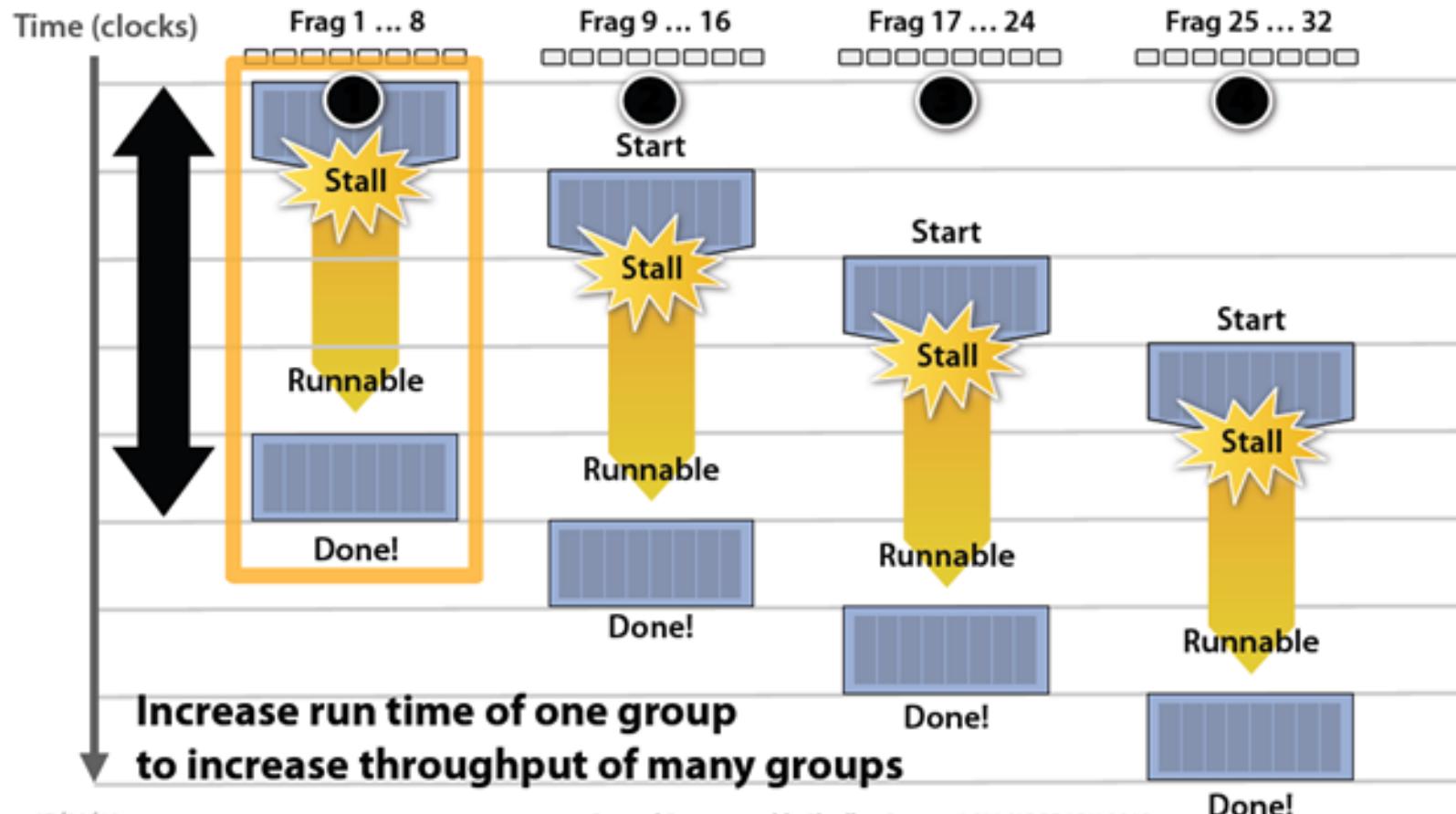
Fermi+ GPUs: each SM has two warp schedulers

SM Warp Scheduling

- SM implements zero-overhead warp scheduling
 - ◆ At any time, only one of the warps is executed by SM
 - ◆ Warps whose next instruction has its operands ready for consumption are eligible for execution
 - ◆ Eligible warps are selected for execution on a prioritized scheduling policy
 - ◆ All threads in a warp execute the same instruction when selected



SM Warp Scheduling



07/29/10

Beyond Programmable Shading Course, ACM SIGGRAPH 2010

Done!

Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf



Scheduling Threads

- What happens if branches in a warp diverge?

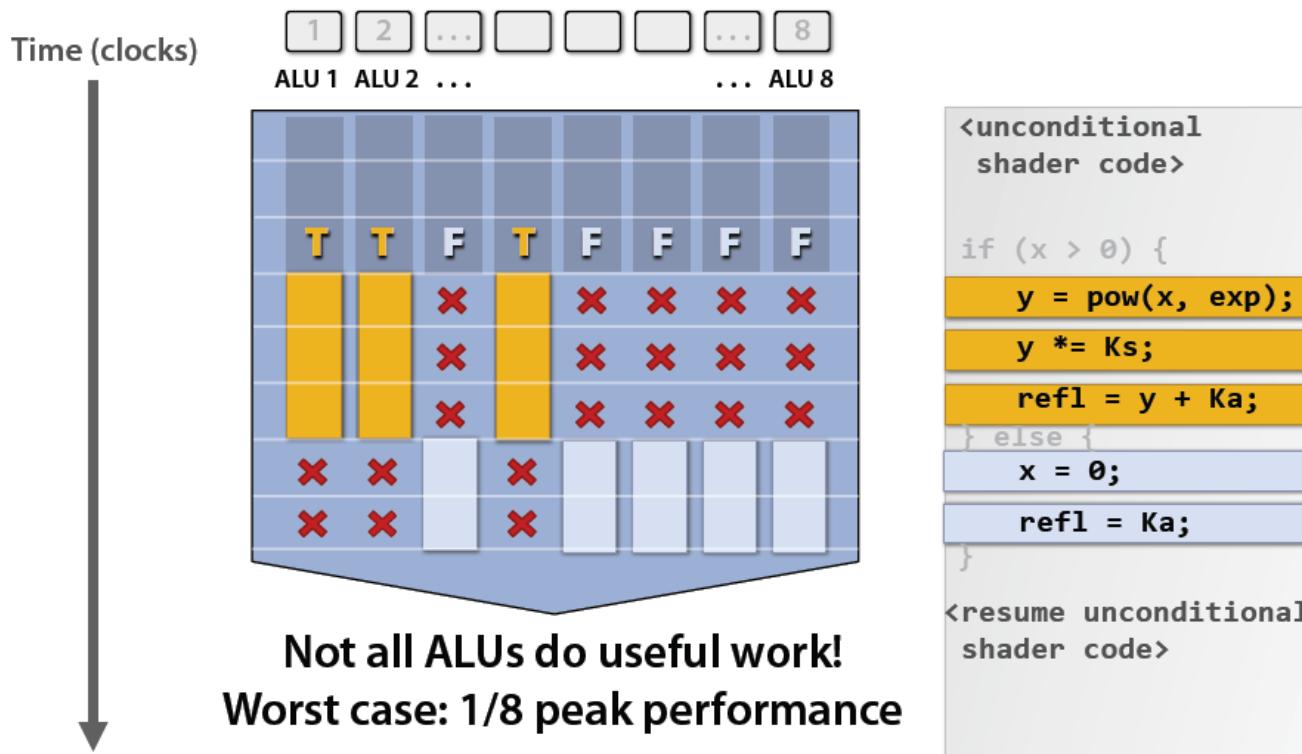


Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf



Outline

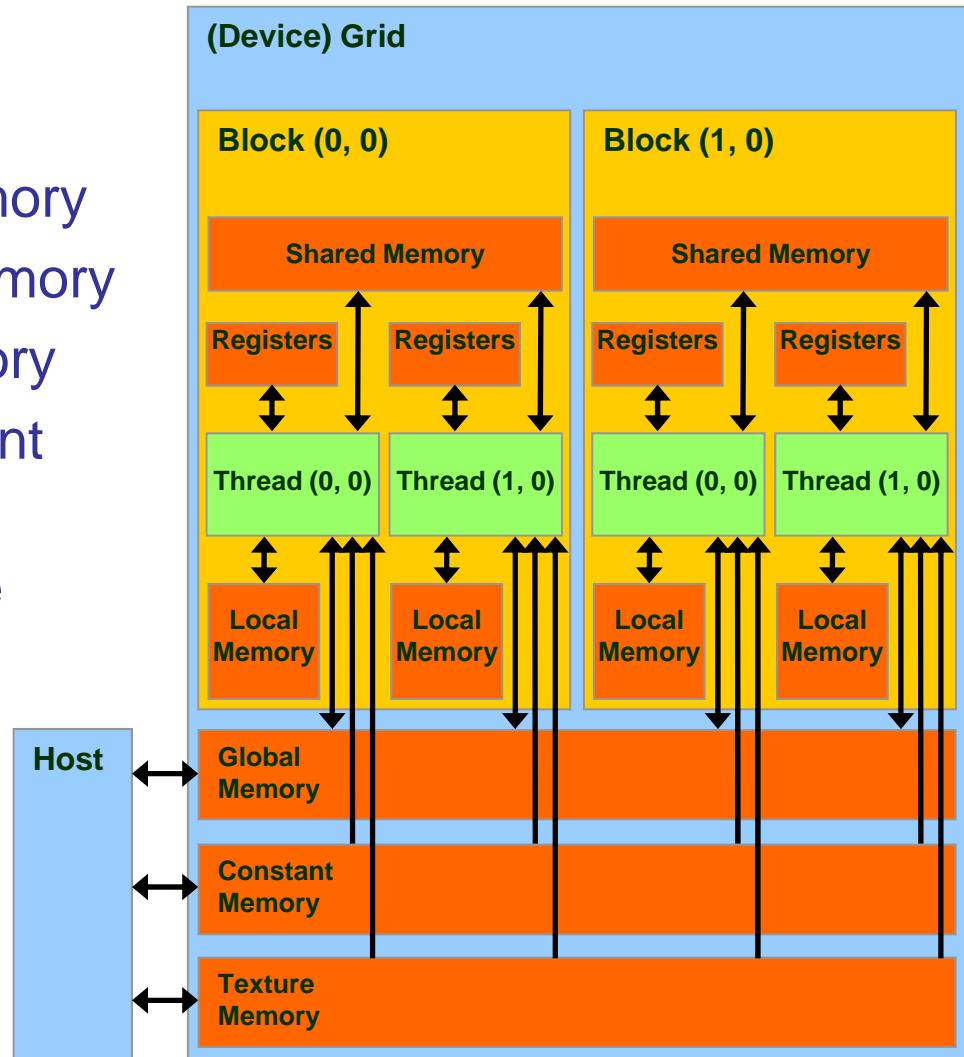
- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ❖ Built-in Data Types and Functions
 - ❖ Thread Model
 - ❖ **Memory Model**
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



CUDA Memory Model

Each thread can:

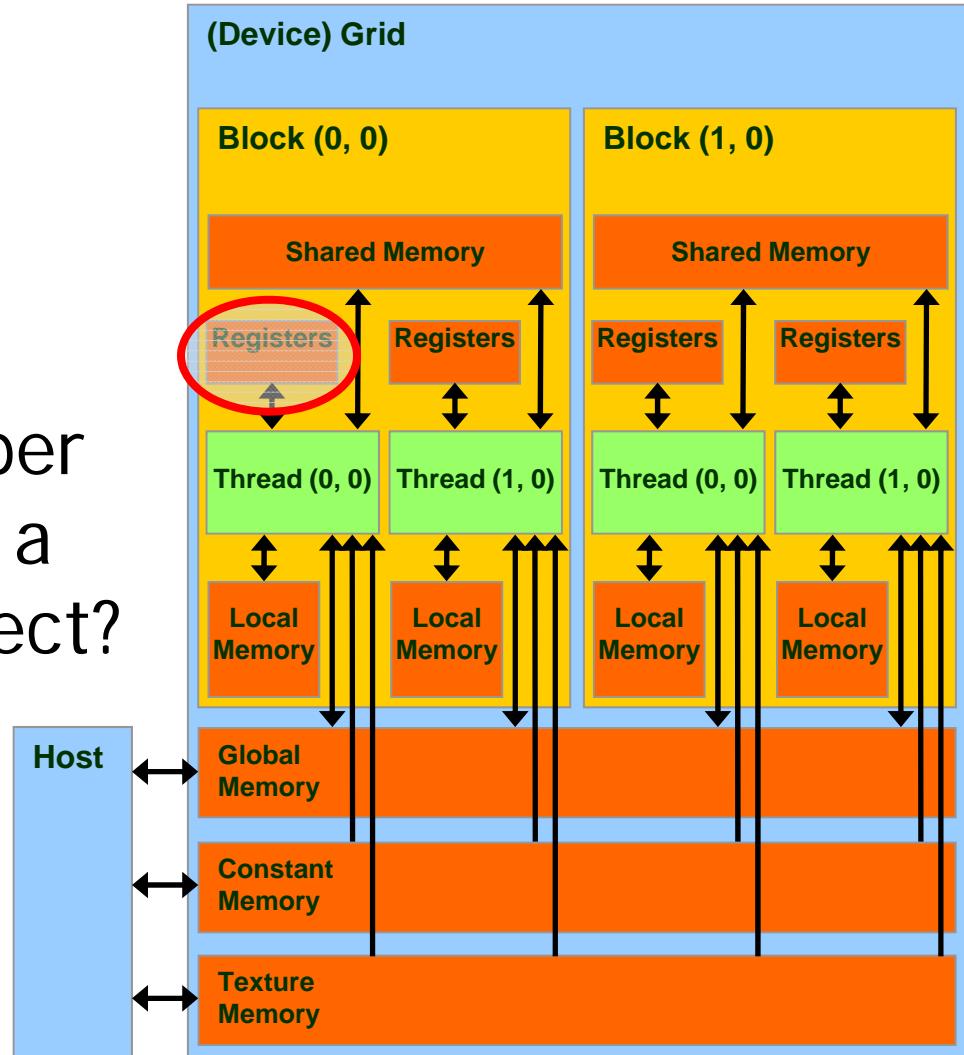
- ⊕ R/W per-thread registers
- ⊕ R/W per-thread local memory
- ⊕ R/W per-block shared memory
- ⊕ R/W per-grid global memory
- ⊕ Read only per-grid constant memory
- ⊕ Read only per-grid texture memory



Memory Model

Registers

- Per thread
- Fast, on-chip, read/write access
- Increasing the number of registers used by a kernel has what affect?



Memory Model

■ Registers - G80

- ◆ Per SM
 - ◆ Up to 768 threads to interleavingly execute
 - ◆ 8K registers
- ◆ How many registers per thread?



Memory Model

■ Registers - G80

- ◆ $8K / 768 = 10$ registers per thread
- ◆ Exceeding limit reduces threads by the block
- ◆ Example: Each thread uses 11 registers, and each block has 256 threads
 - ◆ How many threads can an SM host?
 - ◆ How many warps can an SM host?
 - ◆ What does having less warps mean?

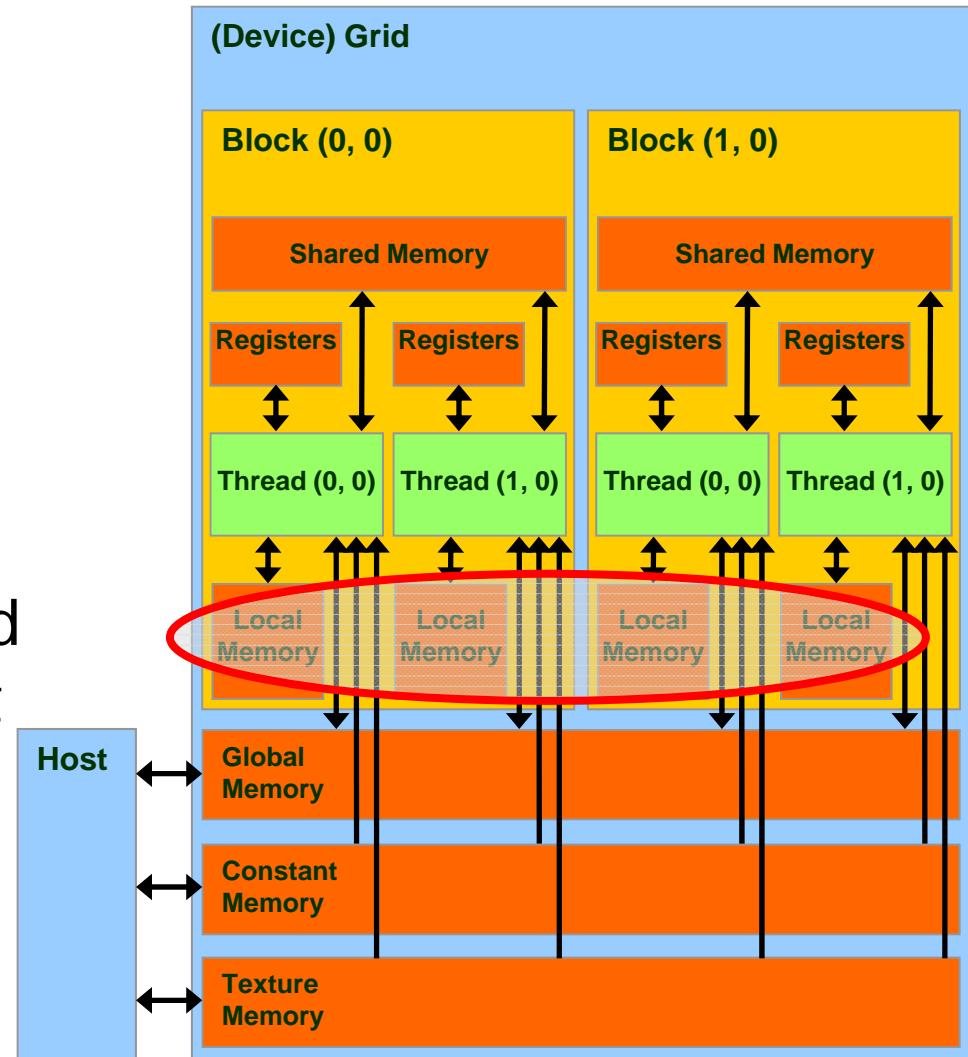
http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls



Memory Model

Local Memory

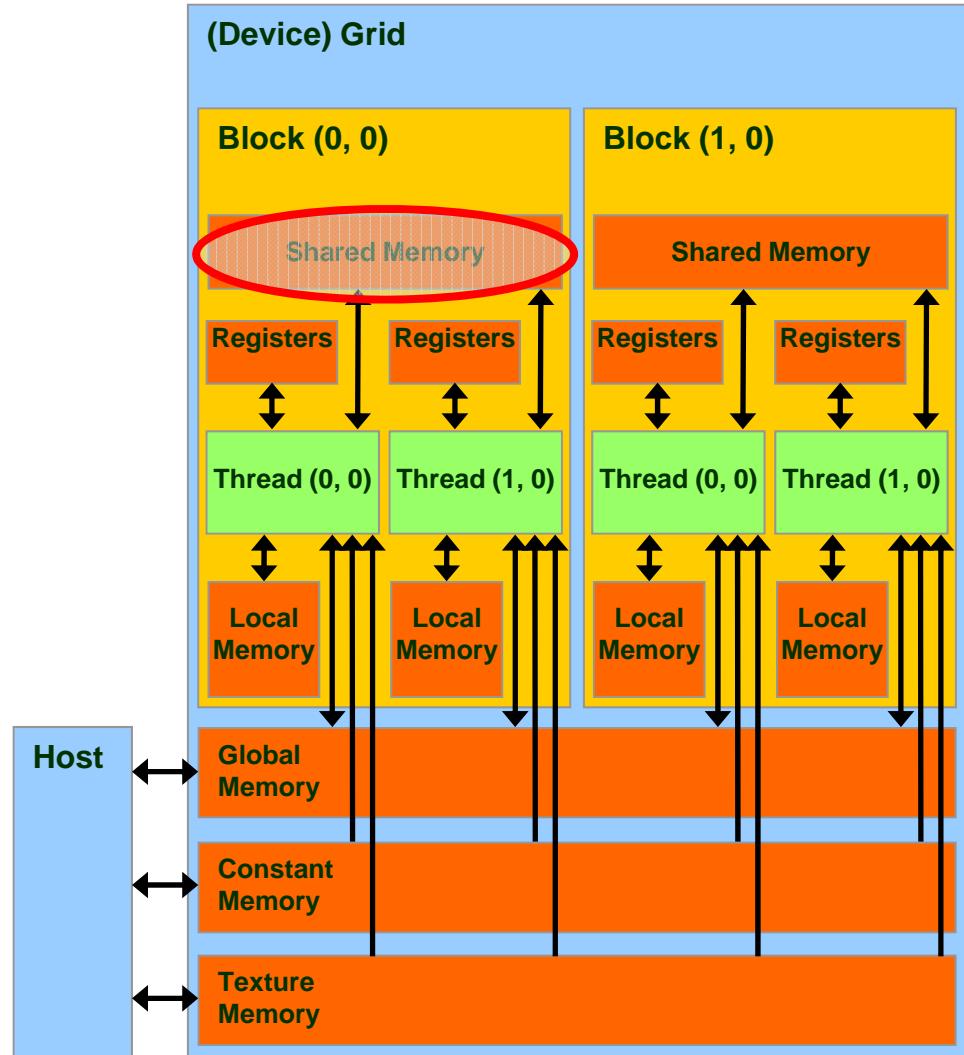
- Stored in global memory
 - Copy per thread
- Used for automatic arrays
 - Unless all accessed with only constant indices



Memory Model

■ Shared Memory

- ❖ Per block
- ❖ Fast, on-chip, read/write access
- ❖ Full speed random access



Memory Model

- Shared Memory - G80
 - ⊕ Per SM
 - ◆ Up to 8 blocks
 - ◆ 16 KB
 - ⊕ How many KB per block



Memory Model

■ Shared Memory - G80

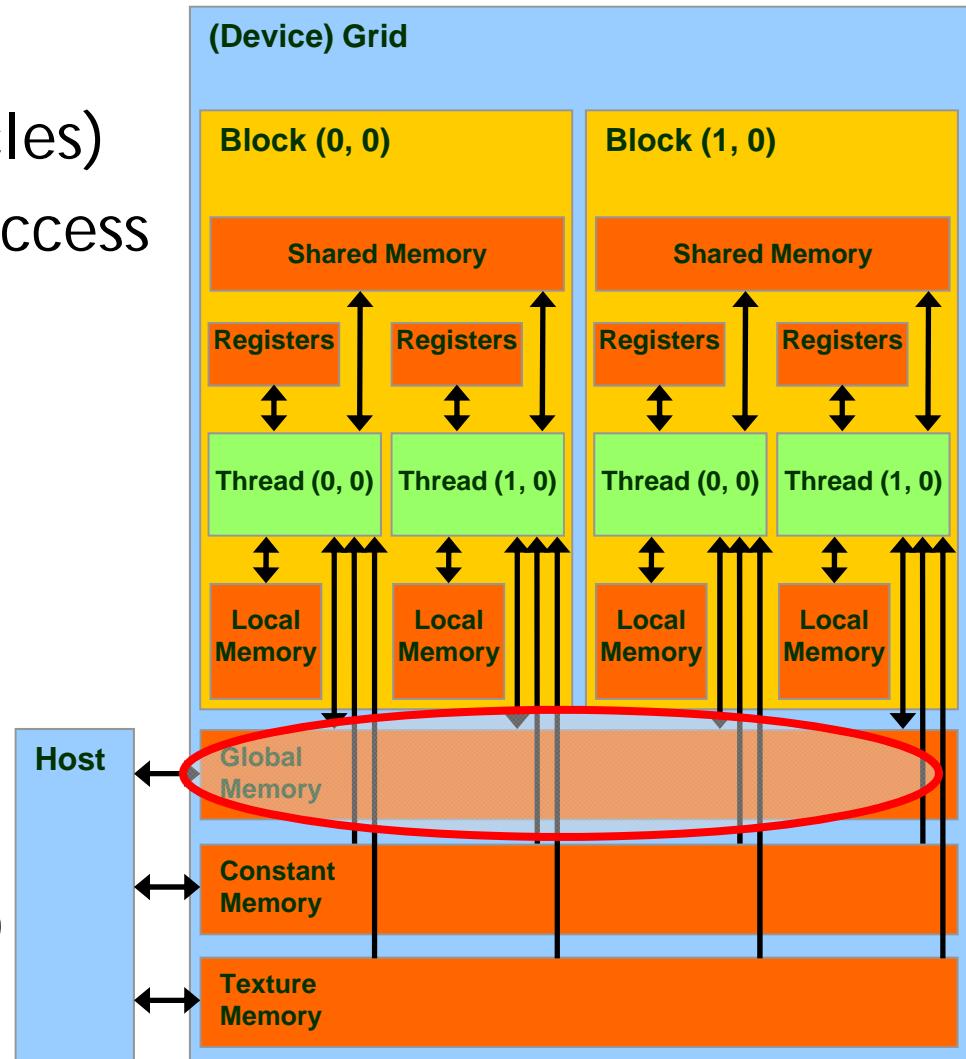
- ⊕ 16 KB / 8 = 2 KB per block
- ⊕ Example
 - ◆ If each block uses 5 KB, how many blocks can a SM host?



Memory Model

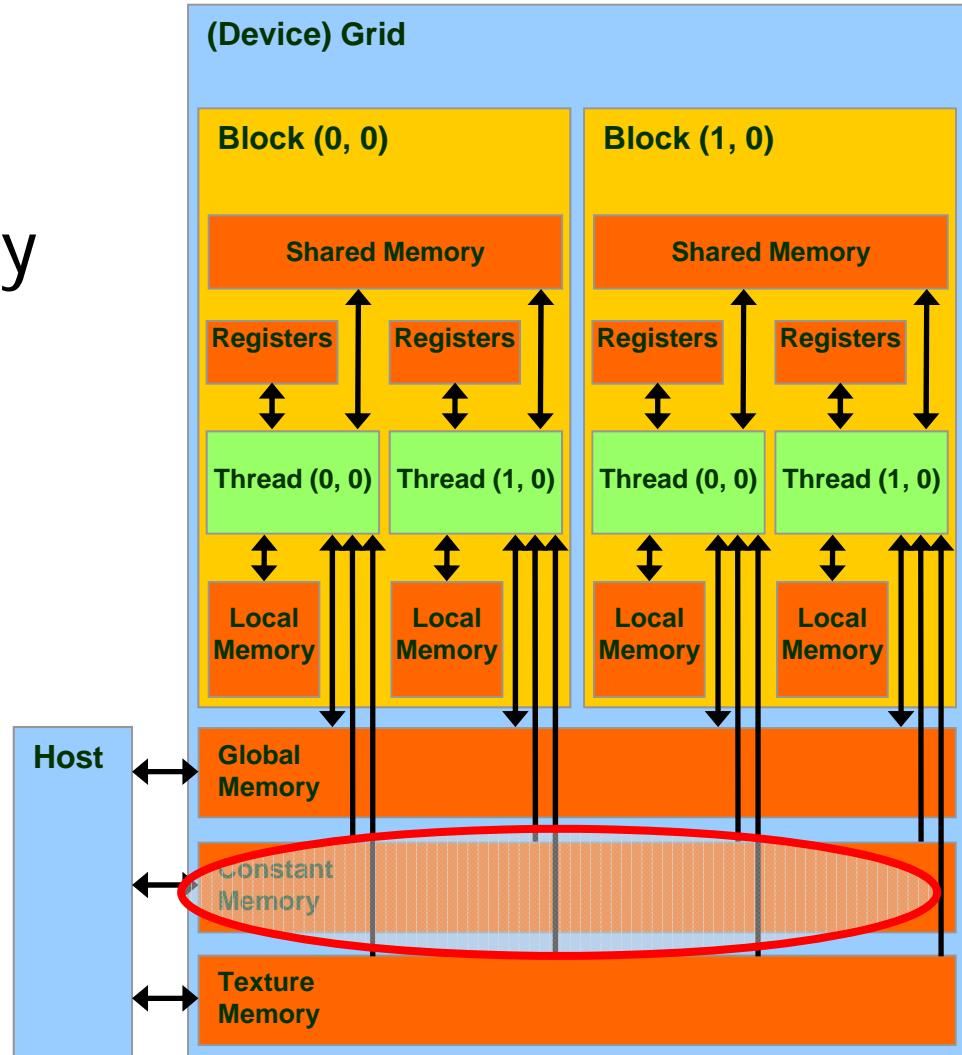
Global Memory

- ❖ Long latency (100s cycles)
- ❖ Off-chip, read/write access
- ❖ Random access causes performance hit
- ❖ Host can read/write
- ❖ G80 – 86.4 GB/s
- ❖ GT200
 - ❖ 150 GB/s
 - ❖ Up to 4 GB
- ❖ Tesla K20 (Kepler GK110)
 - ❖ Up to 208 GB/s
 - ❖ Up to 5GB



Memory Model

- Constant Memory
 - Short latency, high bandwidth, read only access when all threads access the same location
 - Stored in global memory but cached
 - Host can read/write
 - Up to 64 KB



Memory Model

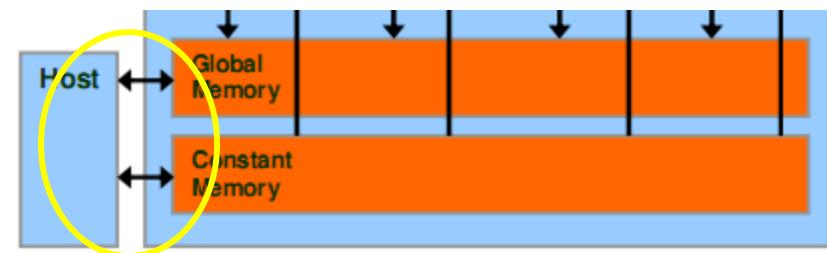
Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	register	thread	kernel
Automatic array variables	local	thread	kernel
<code>__shared__ int sharedVar;</code>	shared	block	kernel
<code>__device__ int globalVar;</code>	global	grid	application
<code>__constant__ int constantVar;</code>	constant	grid	application

- `__shared__` is used within kernels
- `__device__` and `__constant__` is used outside kernels



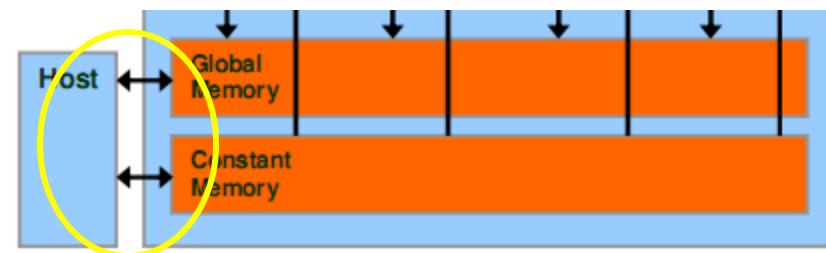
CUDA Memory Transfers

- Host can transfer to/from device
 - ⊕ *Global* memory
 - ⊕ *Constant* memory



CUDA Memory Transfers

- `cudaMalloc()`
 - ◆ Allocate global memory on device
- `cudaFree()`
 - ◆ Frees memory
- Can only be called by the host



CUDA Memory Transfers

```
float *Md  
int size = Width * Width * sizeof(float);  
cudaMalloc( (void**) &Md, size);  
...  
cudaFree(Md);
```



CUDA Memory Transfers

```
float *Md  
int size = Width * Width * sizeof(float);  
  
cudaMalloc( (void**)&Md, size );  
...  
cudaFree(Md);
```

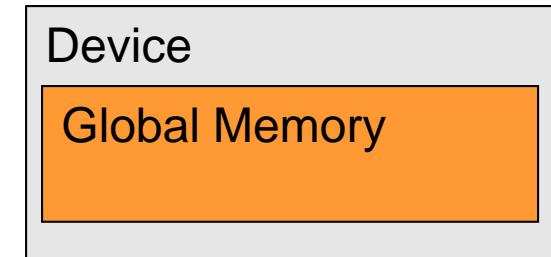
Diagram annotations:

- A yellow box highlights the memory allocation call: `cudaMalloc((void**)&Md, size);`
- A red box labeled "Size in bytes" has two arrows pointing to the variable `size` in the code.
- A red box labeled "Pointer to device memory" has one arrow pointing to the variable `Md` in the code.



CUDA Memory Transfers

- `cudaMemcpy()`
 - ◆ Memory transfer
 - ◆ Host to host
 - ◆ Host to device
 - ◆ Device to host
 - ◆ Device to device



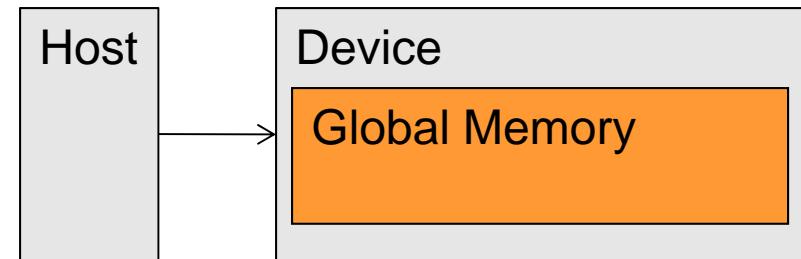
CUDA Memory Transfers

- `cudaMemcpy()`
 - ◆ Memory transfer
 - ◆ Host to host
 - ◆ Host to device
 - ◆ Device to host
 - ◆ Device to device



CUDA Memory Transfers

- `cudaMemcpy()`
 - ◆ Memory transfer
 - ◆ Host to host
 - ◆ Host to device
 - ◆ Device to host
 - ◆ Device to device



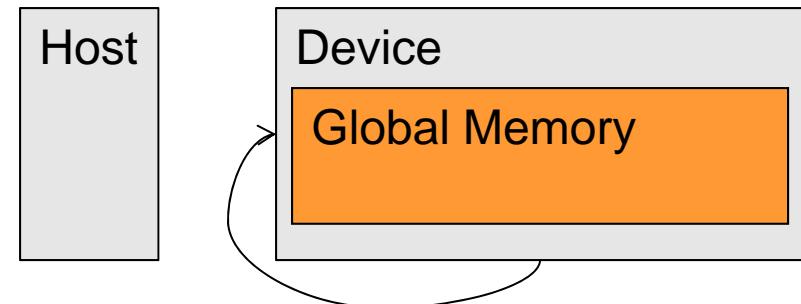
CUDA Memory Transfers

- `cudaMemcpy()`
 - ◆ Memory transfer
 - ◆ Host to host
 - ◆ Host to device
 - ◆ Device to host
 - ◆ Device to device

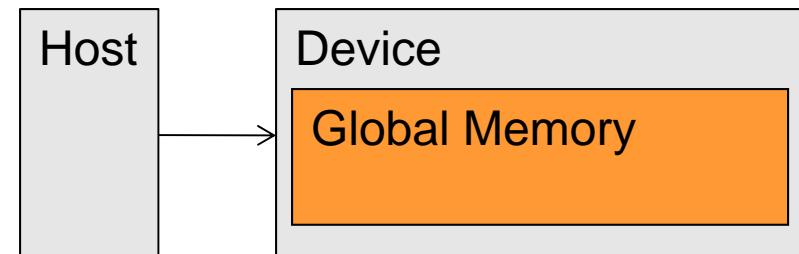
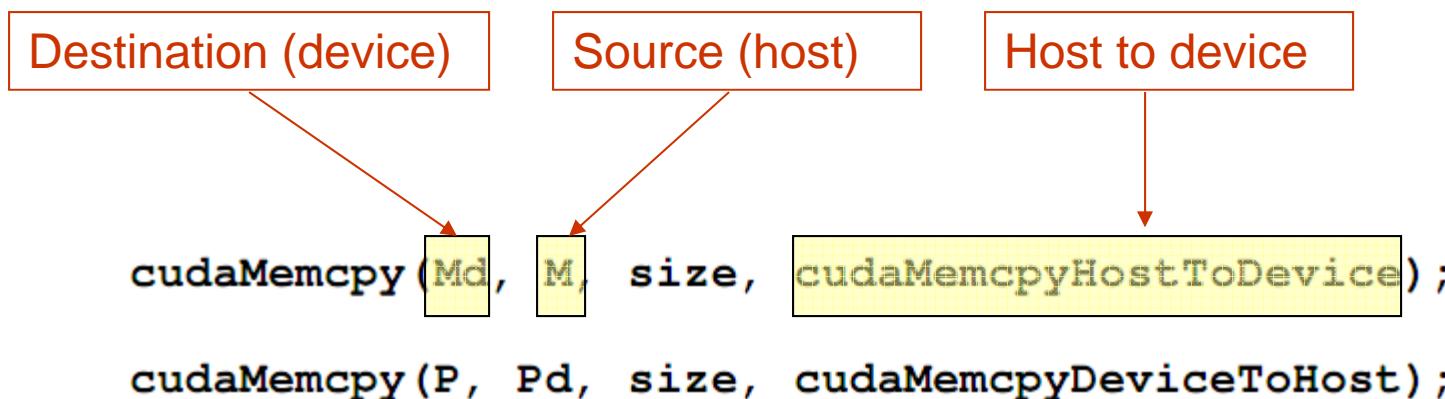


CUDA Memory Transfers

- `cudaMemcpy()`
 - ◆ Memory transfer
 - ◆ Host to host
 - ◆ Host to device
 - ◆ Device to host
 - ◆ Device to device



CUDA Memory Transfers



CUDA Memory Transfers

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);  
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

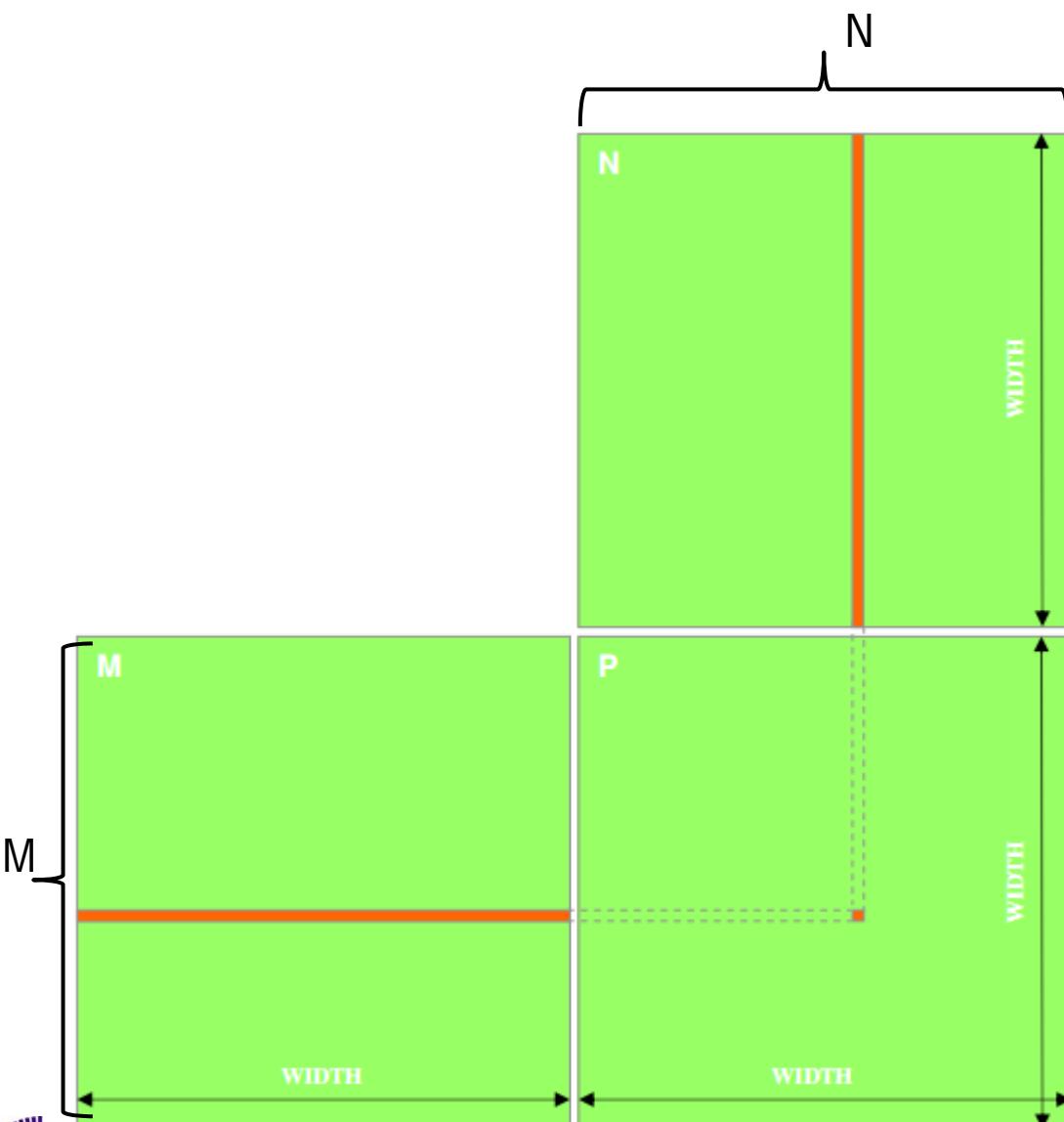


Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ◆ Built-in Data Types and Functions
 - ◆ Thread Model
 - ◆ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



Matrix Multiplication



- $P = M * N$
- Assume M and N are square for simplicity

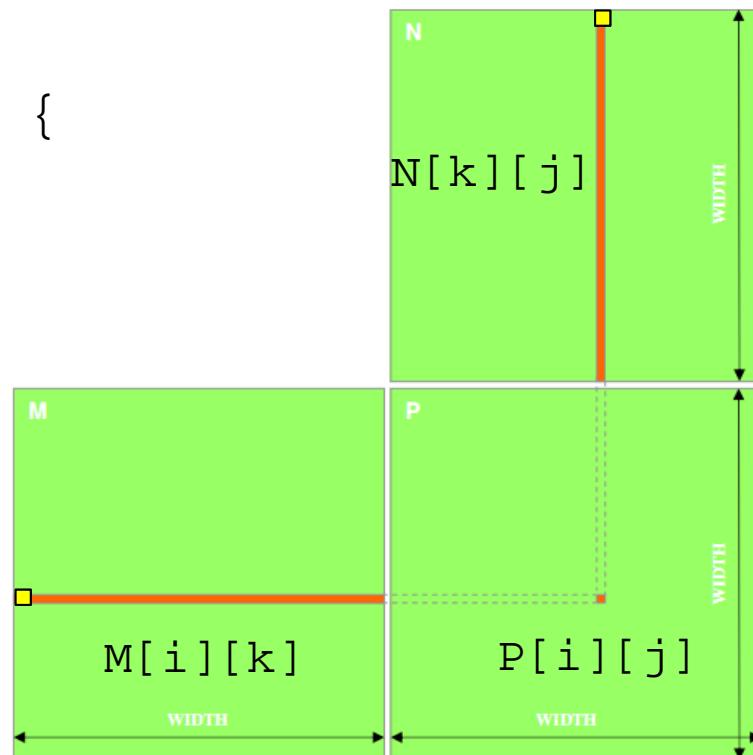
Matrix Multiplication

- 1,000 x 1,000 matrix
 - 1,000,000 dot products
 - Each 1,000 multiples and 1,000 adds



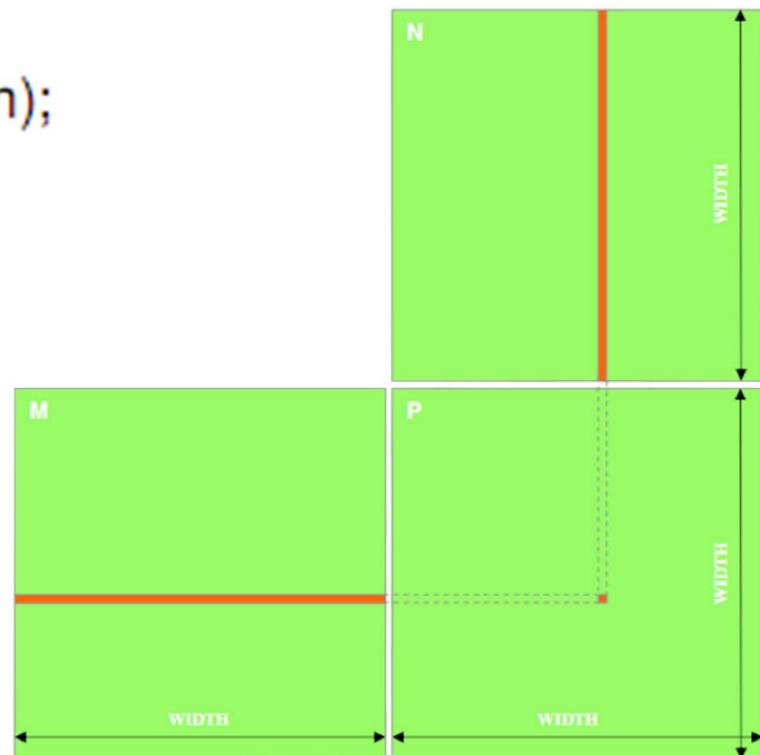
Matrix Multiplication: CPU Implementation

```
void MatrixMulOnHost(float* M, float* N, float* P, int width)
{
    for (int i = 0; i < width; ++i) {
        for (int j = 0; j < width; ++j) {
            float sum = 0;
            for (int k = 0; k < width; ++k) {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * width + j] = sum;
        }
    }
}
```



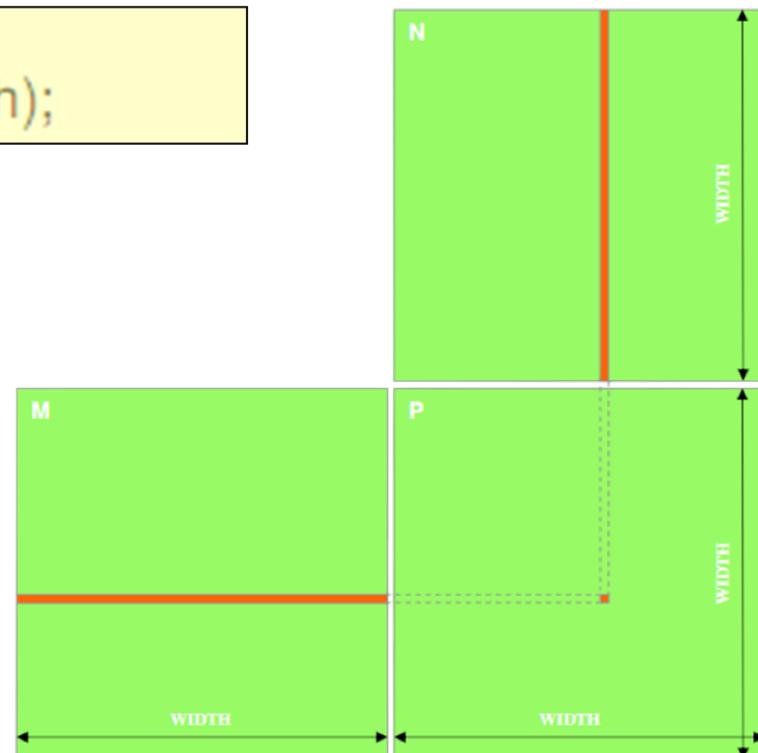
Matrix Multiplication: CUDA Skeleton

```
int main(void) {
    1. // Allocate and initialize the matrices M, N, P
        // I/O to read the input matrices M and N
    ...
    2. // M * N on the device
        MatrixMulOnDevice(M, N, P, width);
    ...
    3. // I/O to write the output matrix P
        // Free matrices M, N, P
    ...
    return 0;
}
```



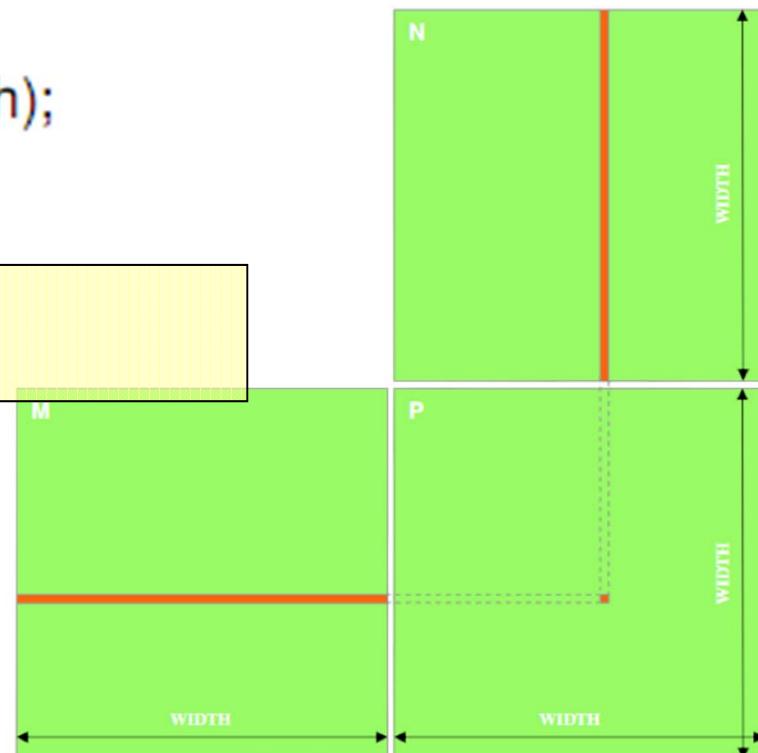
Matrix Multiplication: CUDA Skeleton

```
int main(void) {
    1. // Allocate and initialize the matrices M, N, P
        // I/O to read the input matrices M and N
    ...
    2. // M * N on the device
        MatrixMulOnDevice(M, N, P, width);
    ...
    3. // I/O to write the output matrix P
        // Free matrices M, N, P
    ...
    return 0;
}
```



Matrix Multiplication: CUDA Skeleton

```
int main(void) {
    1. // Allocate and initialize the matrices M, N, P
        // I/O to read the input matrices M and N
    ...
    2. // M * N on the device
        MatrixMulOnDevice(M, N, P, width);
    ...
    3. // I/O to write the output matrix P
        // Free matrices M, N, P
    ...
    return 0;
}
```



Matrix Multiplication

- Step 1
 - + Add *CUDA memory transfers* to the skeleton



Matrix Multiplication: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);    float *Md, *Nd, *Pd;

    1. // Load M and N to device memory
        cudaMalloc(&Md, size);
        cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
        cudaMalloc(&Nd, size);
        cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&Pd, size);

    2. // Kernel invocation code – to be shown later
    ...
    3. // Read P from the device
        cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
        // Free device matrices
        cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

Allocate input



Matrix Multiplication: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);    float *Md, *Nd, *Pd;

1. // Load M and N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&Pd, size);
```

← **Allocate output**

```
2. // Kernel invocation code – to be shown later
    ...
3. // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```



Matrix Multiplication: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);    float *Md, *Nd, *Pd;

    1. // Load M and N to device memory
        cudaMalloc(&Md, size);
        cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
        cudaMalloc(&Nd, size);
        cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

        // Allocate P on the device
        cudaMalloc(&Pd, size);

    2. // Kernel invocation code – to be shown later
        ...
    3. // Read P from the device
        cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
        // Free device matrices
        cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```



Matrix Multiplication: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);    float *Md, *Nd, *Pd;

    1. // Load M and N to device memory
        cudaMalloc(&Md, size);
        cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
        cudaMalloc(&Nd, size);
        cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

        // Allocate P on the device
        cudaMalloc(&Pd, size);

    2. // Kernel invocation code – to be shown later

    3. // Read P from the device
        cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
        // Free device matrices
        cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

Read back
from device



Matrix Multiplication

- Step 2
 - + Implement the *kernel* in CUDA C



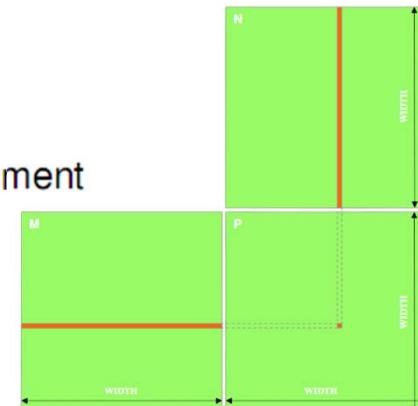
Matrix Multiplication: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y; ← Accessing a matrix, so using a 2D block

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```



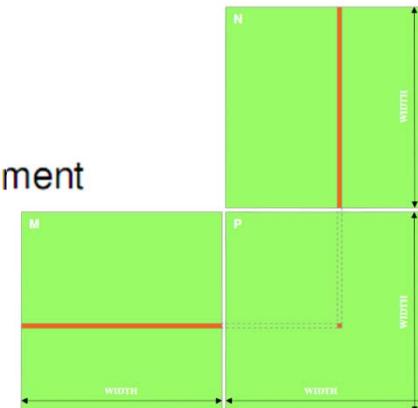
Matrix Multiplication: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0; ←
    Each kernel computes one output

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```



Matrix Multiplication: CUDA Kernel

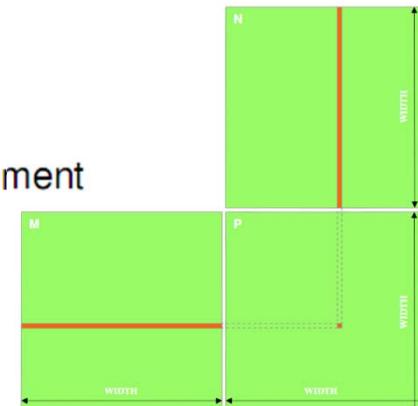
```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

Where did the two outer for loops in the CPU implementation go?



Matrix Multiplication: CUDA Kernel

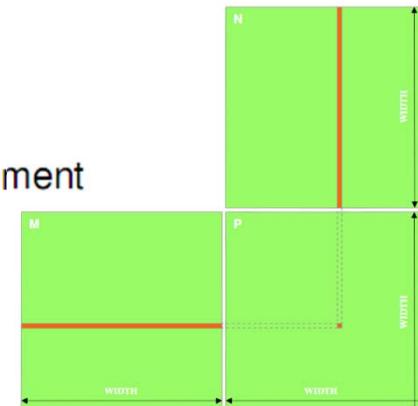
```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

No locks or synchronization, why?



Matrix Multiplication

- Step 3
 - + Invoke the *kernel* in CUDA C



Matrix Multiplication: Invoke Kernel

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);    float *Md, *Nd, *Pd;

    1. // Load M and N to device memory
        cudaMalloc(&Md, size);
        cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
        cudaMalloc(&Nd, size);
        cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&Pd, size);

    2. // Kernel invocation code – to be shown later
    ...
    3. // Read P from the device
        cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);

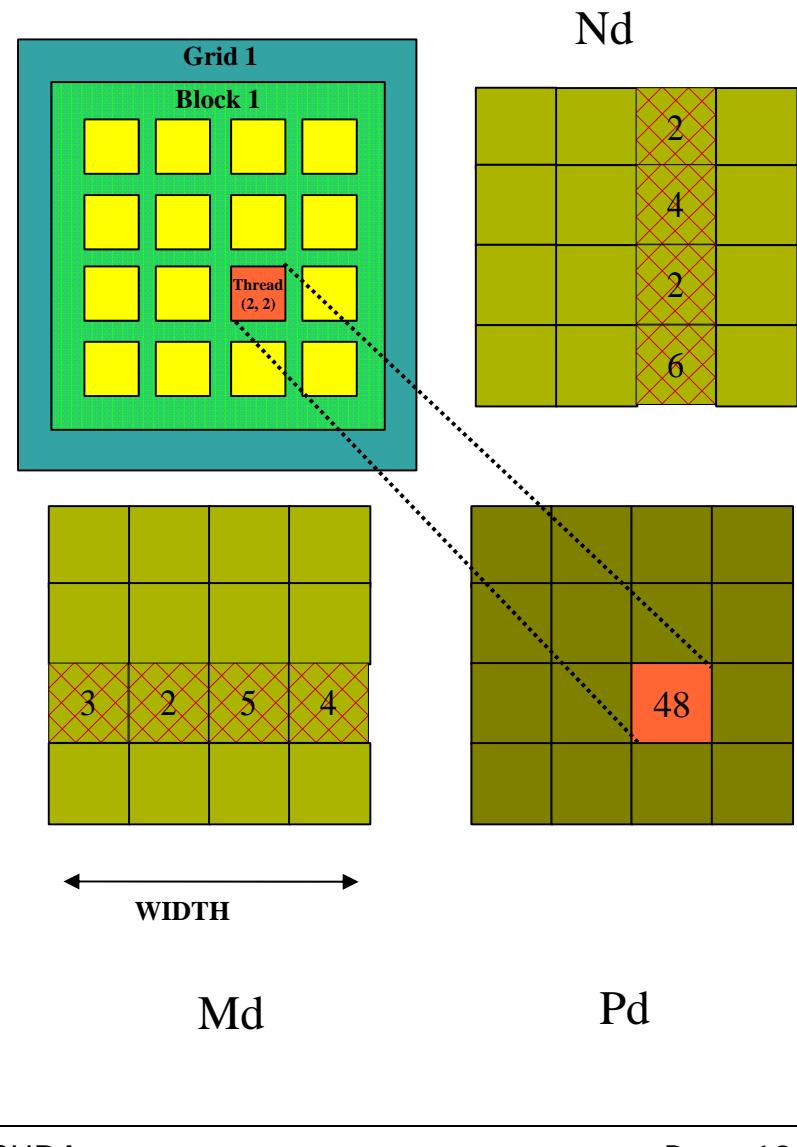
One block with width by width threads

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);



Matrix Multiplication

- One Block of threads compute matrix P_d
 - ✚ Each thread computes one element of P_d
- Each thread
 - ✚ Loads a row of matrix M_d
 - ✚ Loads a column of matrix N_d
 - ✚ Perform one multiply and addition for each pair of M_d and N_d elements
 - ✚ Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



Matrix Multiplication

■ Problems

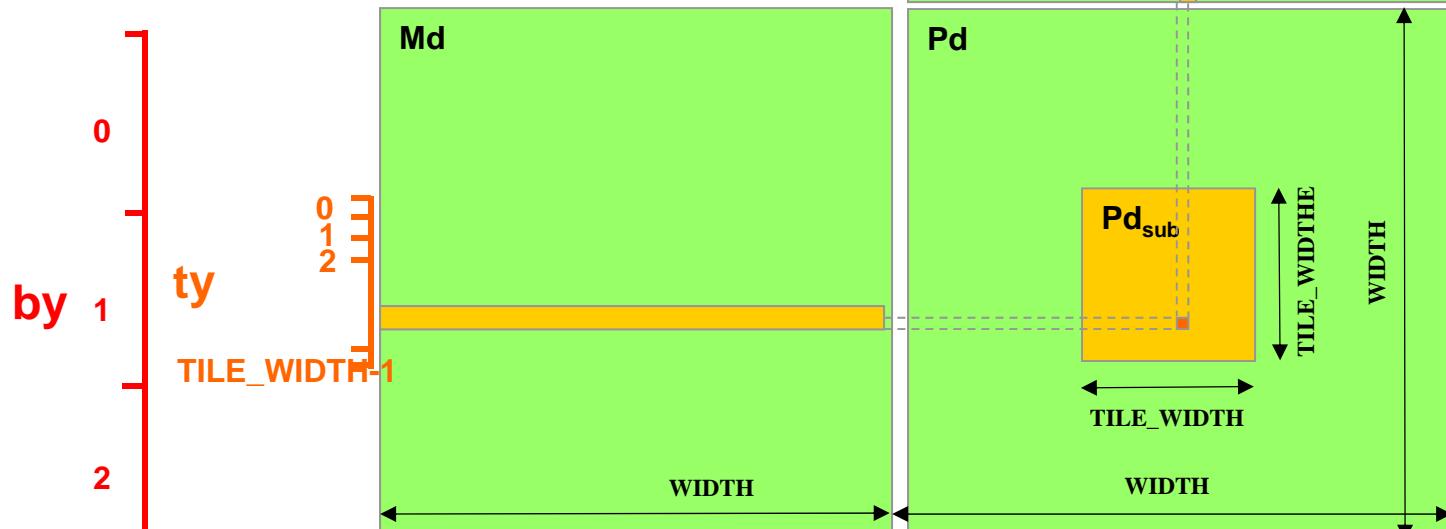
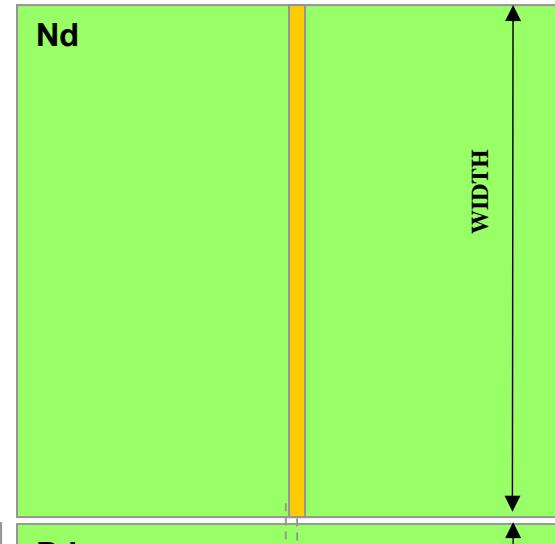
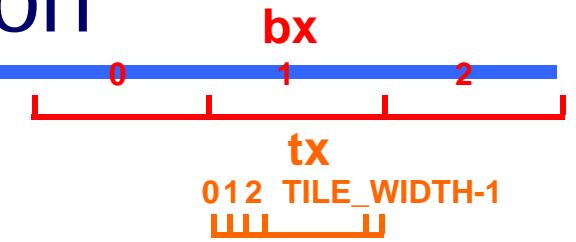
- ⊕ Limited matrix size
 - ◆ Only uses one block
 - ◆ G80 and GT200 - up to 512 threads per block
- ⊕ Lots of global memory access



Matrix Multiplication

■ Remove size limitation

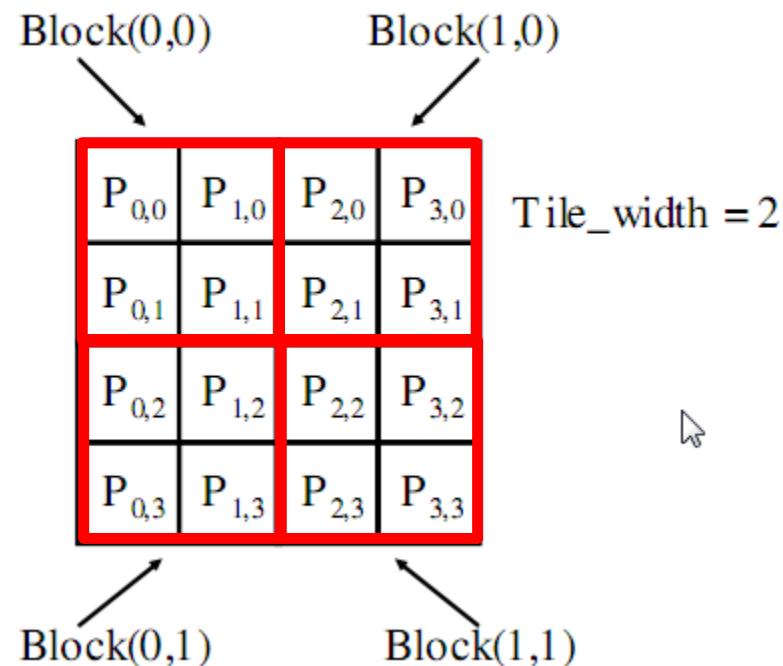
- Break Pd matrix into tiles
- Assign each tile to a block
- Use `threadIdx` and `blockIdx` for indexing



Matrix Multiplication

Example

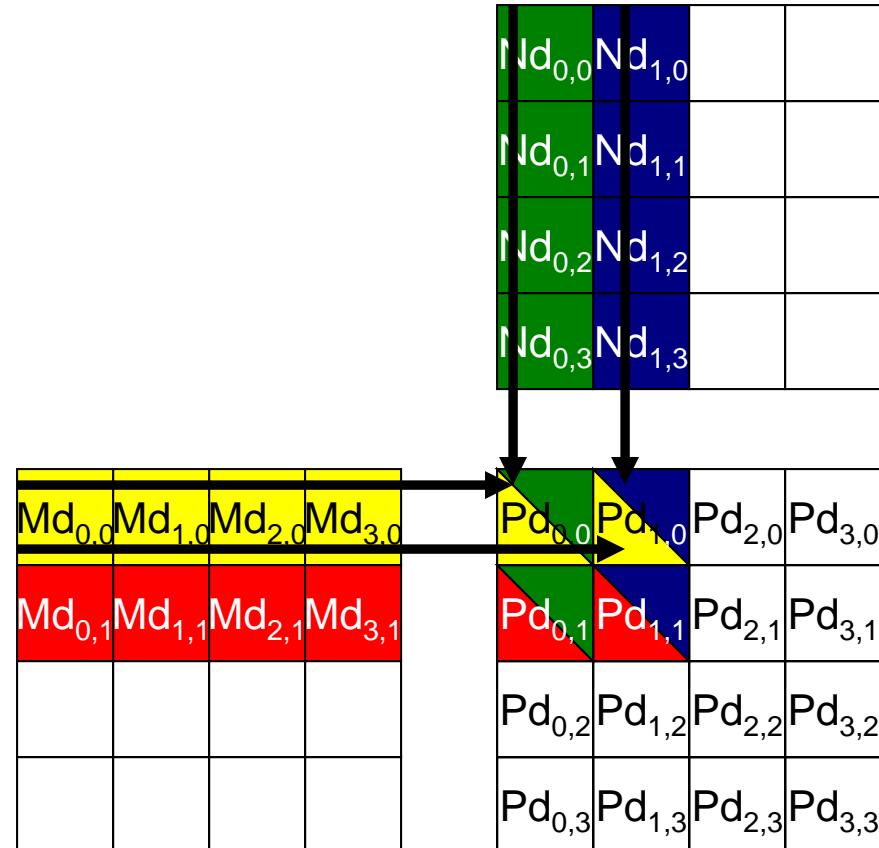
- Matrix: 4x4
 - TILE_WIDTH = 2
- =>
- Block size: 2x2



Matrix Multiplication

Example

- Matrix: 4x4
 - TILE_WIDTH = 2
- =>
- Block size: 2x2



Matrix Multiplication

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    Pd[Row * Width + Col] = Pvalue;
}
```



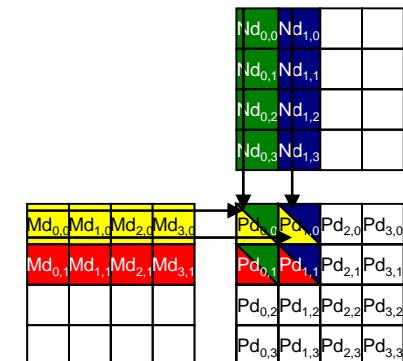
Matrix Multiplication

Calculate the row index of the Pd element and M

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    Pd[Row * Width + Col] = Pvalue;
}
```



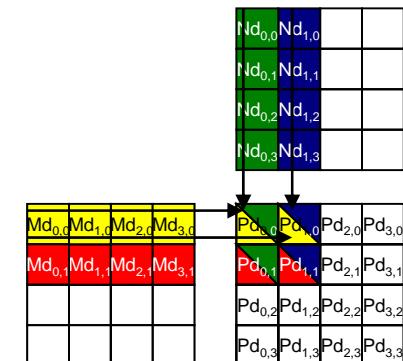
Matrix Multiplication

Calculate the column index of Pd and Nd

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    Pd[Row * Width + Col] = Pvalue;
}
```



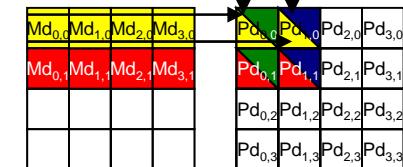
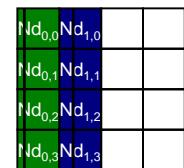
Matrix Multiplication

Each thread computes one element
of the block sub-matrix

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];
}
```

```
Pd[Row * Width + Col] = Pvalue;
```



Matrix Multiplication

- Invoke kernel:

```
dim3 dimGrid(Width / TILE_WIDTH, Height / TILE_WIDTH);  
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
```

```
MatrixMulKernel<<<dimGrid, dimBlock>>>(  
Md, Nd, Pd, WIDTH);
```



What about global memory access?



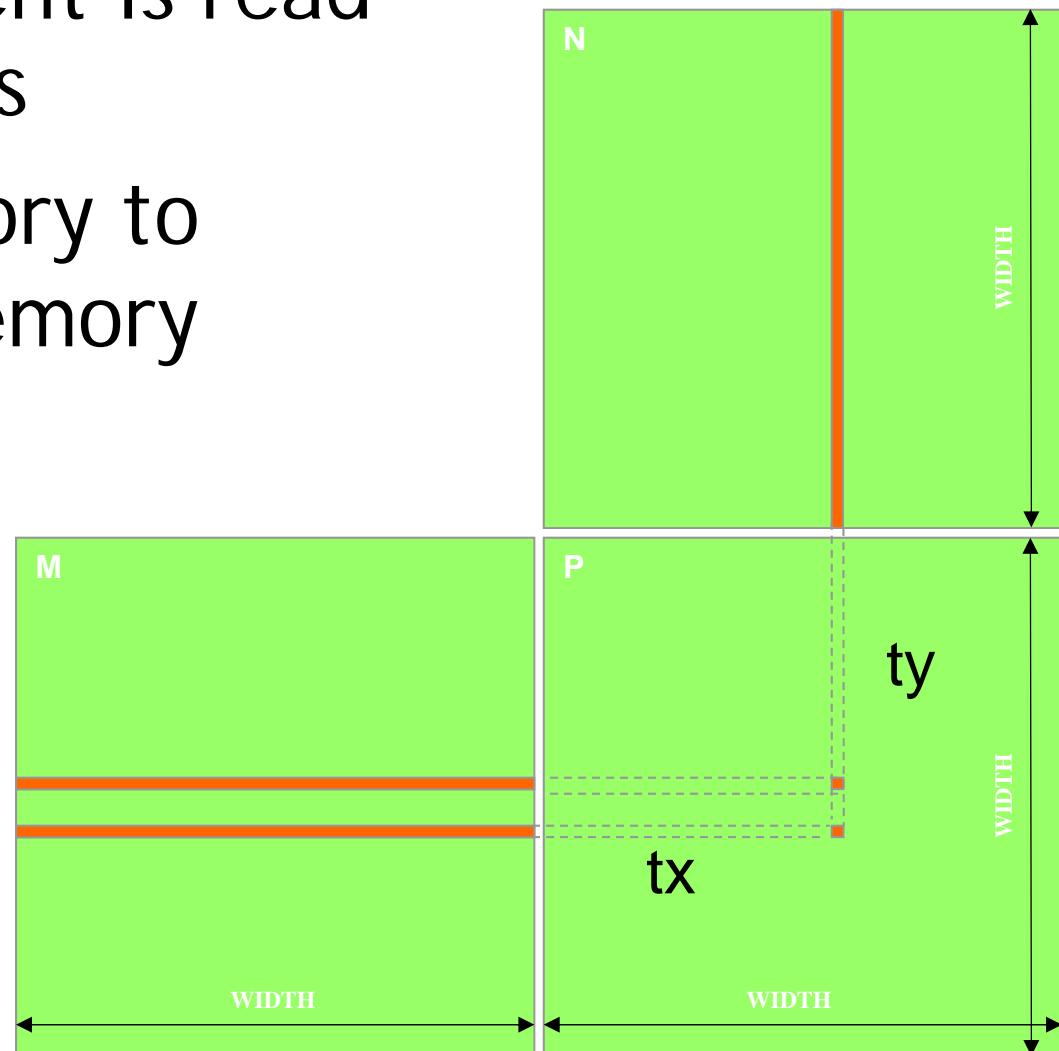
Matrix Multiplication

- Limited by global memory bandwidth
 - ❖ G80 peak GFLOPS: 346.5
 - ❖ Require 1386 GB/s to achieve this
 - ❖ G80 memory bandwidth: 86.4 GB/s
 - ◆ Limits code to 21.6 GFLOPS
 - ◆ In practice, code runs at 15 GFLOPS
 - ❖ Must drastically reduce global memory access



Matrix Multiplication

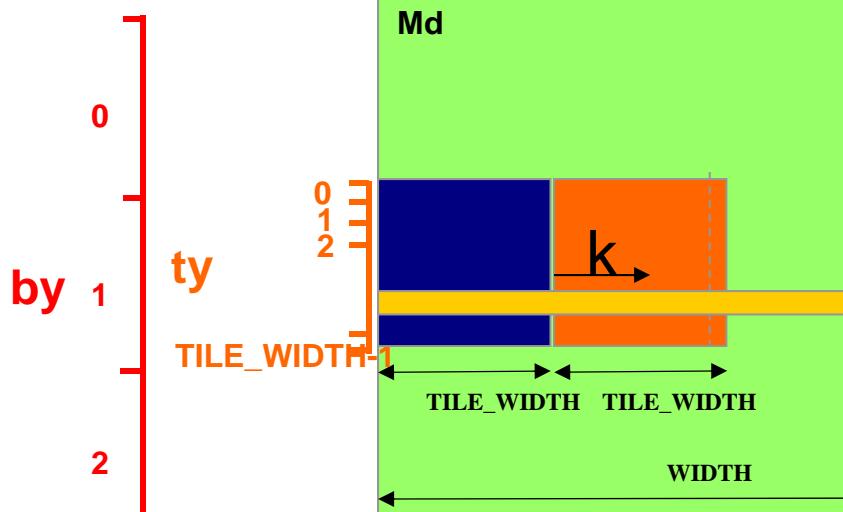
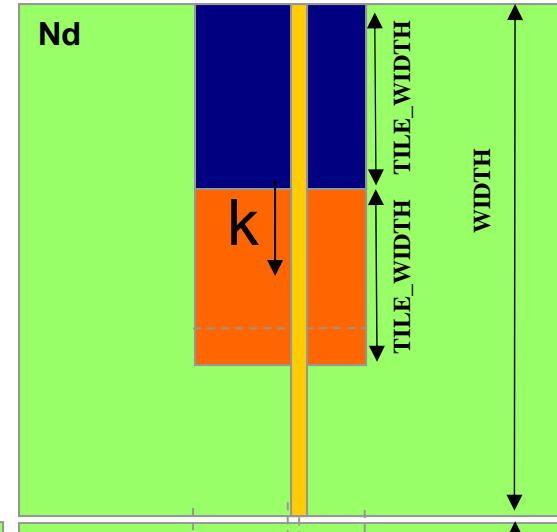
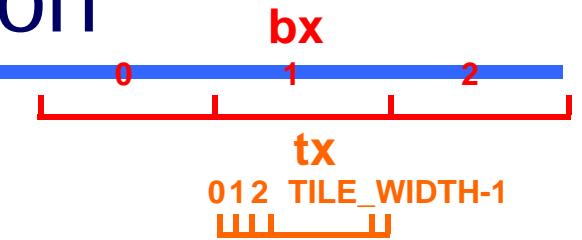
- Each input element is read by width threads
- Use shared memory to reduce global memory bandwidth



Matrix Multiplication

■ Break kernel into phases

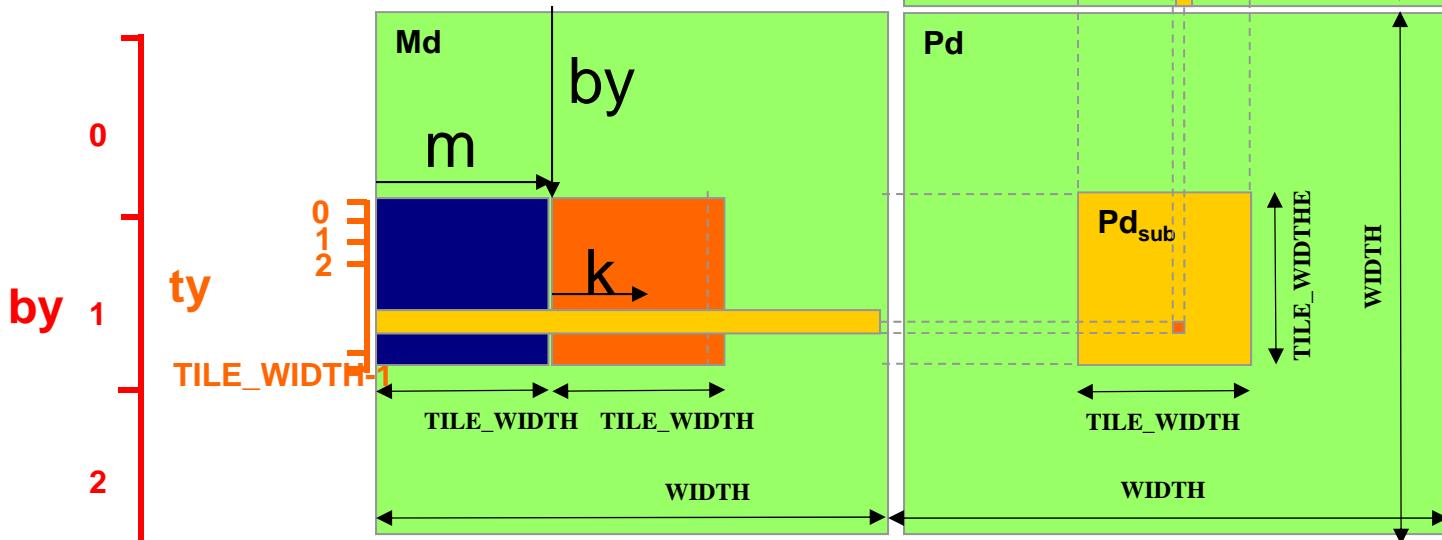
- Each phase accumulates P_d using a subset of M_d and N_d
- Each phase has good data locality



Matrix Multiplication



- Each thread
 - ◆ loads one element of M_d and N_d in the tile into shared memory



Matrix Multiplication

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;    int by = blockIdx.y;
    int tx = threadIdx.x;  int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}
```



Matrix Multiplication

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;    int by = blockIdx.y;
    int tx = threadIdx.x;  int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}
```

Shared memory for a subset of Md and Nd



Matrix Multiplication

```

__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

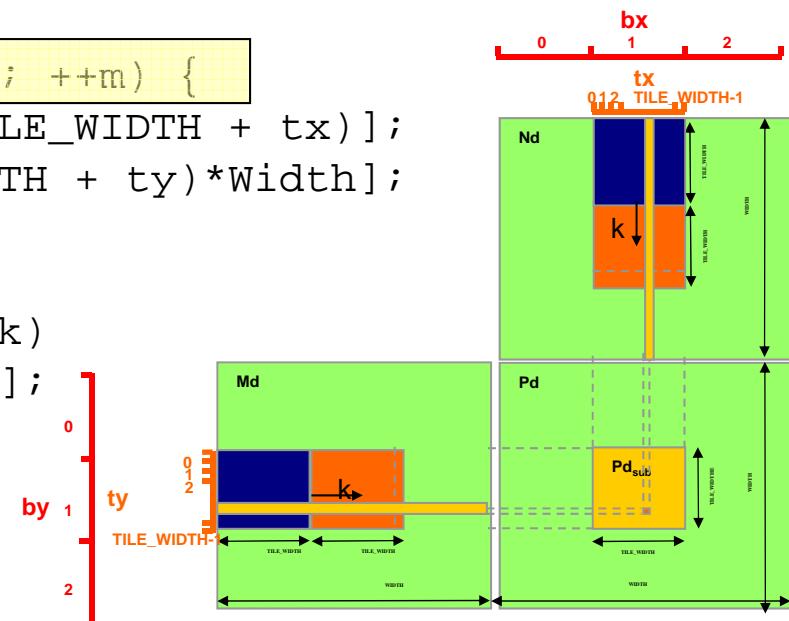
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}

```

- Width/TILE_WIDTH
- Number of phases
 - m
 - Index for current phase



Matrix Multiplication

```

__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

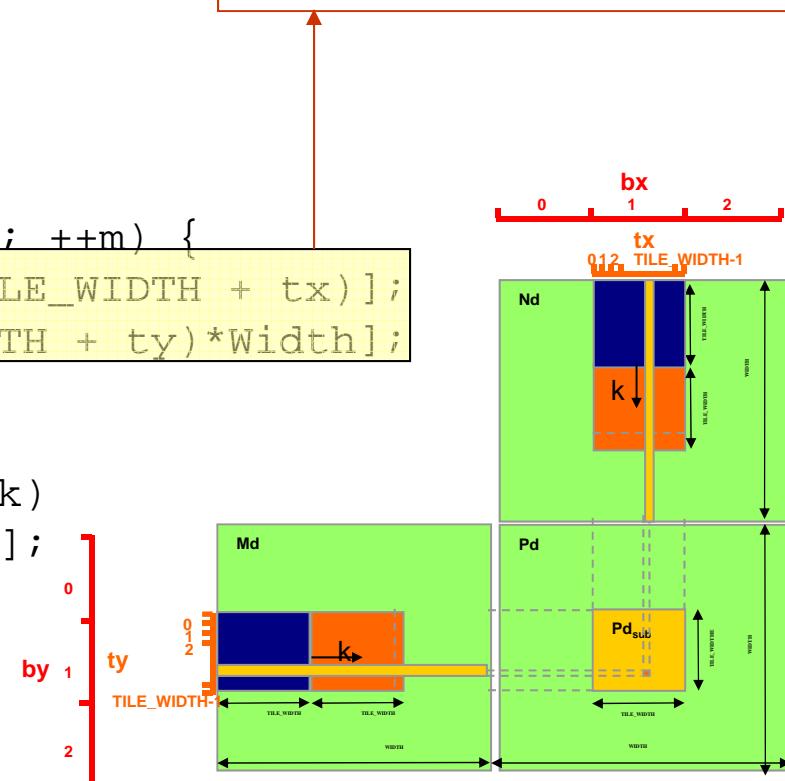
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}

```

Bring one element each from Md and Nd into shared memory



Matrix Multiplication

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads(); // Wait for every thread in the block, i.e., wait for the tile to be in shared memory

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}
```

Wait for every thread in the block, i.e., wait for the tile to be in shared memory



Matrix Multiplication

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}
```

Accumulate subset of dot product



Matrix Multiplication

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}
```

Why?



Matrix Multiplication

```
__global__ void MatrixMulKernel(
    float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}
```

Write final answer
to global memory



Matrix Multiplication

- How do you pick TILE_WIDTH?
 - ✚ How can it be too large?



Matrix Multiplication

- How do you pick TILE_WIDTH?
 - ◆ How can it be too large?
 - ◆ By exceeding the maximum number of threads/block
 - G80 and GT200 - 512
 - Fermi - 1024



Matrix Multiplication

- How do you pick TILE_WIDTH?
 - ◆ How can it be too large?
 - ◆ By exceeding the maximum number of threads/block
 - G80 and GT200 - 512
 - Fermi - 1024 (`TILE_WIDTH = 32`)
 - ◆ By exceeding the shared memory limitations
 - G80: 16KB per SM and up to 8 blocks per SM
 - » 2 KB per block
 - » 1 KB for Nds and 1 KB for Mds ($16 * 16 * 4$)
 - » `TILE_WIDTH = 16`
 - » A larger `TILE_WIDTH` will result in less blocks



Matrix Multiplication

- Shared memory tiling benefits
 - ◆ Reduces global memory access by a factor of TILE_WIDTH
 - ◆ 16x16 tiles reduces by a factor of 16
 - ◆ G80
 - ◆ Now global memory supports 345.6 GFLOPS
 - $21.6 \text{ GFLOPS} * 16 = 345.6 \text{ GFLOPS}$
 - ◆ Close to maximum of 346.5 GFLOPS



Outline

- GPUs as Compute Engines
- GPU Architectures
 - ❖ GPU Threading Model
 - ❖ Separation of Memory Spaces
- Introduction to CUDA
 - ❖ Parallel Computing Platform
 - ❖ Programming Model
 - ◆ Built-in Data Types and Functions
 - ◆ Thread Model
 - ◆ Memory Model
 - ❖ Case Study: Matrix Multiplication
 - ❖ Thread Synchronization



Thread Synchronization

- Threads in a block can synchronize
 - call `__syncthreads` to create a barrier
 - A thread waits at this call until all threads in the block reach it, then all threads continue

```
Mds[ i ] = Md[ j ];  
__syncthreads( );  
func( Mds[ i ], Mds[ i + 1 ] );
```



Thread Synchronization

- Assume that the warp size is 2 and the block size is 4.

Thread 0

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Time: 0



Thread Synchronization

- Assume that the warp size is 2 and the block size is 4.

Thread 0

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Time: 1



Thread Synchronization

- Assume that the warp size is 2 and the block size is 4.

Thread 0

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Threads 0 and 1 are blocked at barrier

Time: 1



Thread Synchronization

- Assume that the warp size is 2 and the block size is 4.

Thread 0

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Time: 2



Thread Synchronization

- Assume that the warp size is 2 and the block size is 4.

Thread 0

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Time: 3



Thread Synchronization

- Assume that the warp size is 2 and the block size is 4.

Thread 0

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

All threads in block have reached barrier, any thread can continue

Time: 3



Thread Synchronization

- Assume that the warp size is 2 and the block size is 4.

Thread 0

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Time: 4



Thread Synchronization

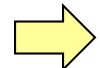
- Assume that the warp size is 2 and the block size is 4.

Thread 0

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```



Thread 2

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];  
__syncthreads();  
func(Mds[i], Mds[i+1]);
```

Time: 5



Atomic Functions

- What is the value of count if 8 threads execute `++count`?

```
__device__ unsigned int count = 0;  
// ...  
__global__ kernel(...) {  
    // ...  
    ++count;  
    // ...  
}
```



Atomic Functions

- Read-modify-write atomic operation
 - ❖ Guaranteed no interference from other threads
 - ❖ No guarantee on order
- Shared or global memory
- Requires compute capability 1.1 (> G80)



Atomic Functions

- What is the value of count if 8 threads execute `atomicAdd` below?

```
__device__ unsigned int count = 0;  
// ...  
__global__ kernel(...) {  
    // ...  
    // atomic ++count  
    atomicAdd(&count, 1);  
    // ...  
}
```



Atomic Functions

- How do you implement `atomicAdd`?

```
__device__ int atomicAdd(  
    int *address, int val);
```



Atomic Functions

- How do you implement `atomicAdd`?

```
__device__ int atomicAdd(  
    int *address, int val)  
{ // Made up keyword:  
    __lock (address) {  
        *address += val;  
    }  
}
```



Atomic Functions

- How do you implement `atomicAdd` without locking?



Atomic Functions

- How do you implement `atomicAdd` without locking?
- What if you were given an atomic compare and swap?

```
int atomicCAS( int *address, int  
compare, int val );
```

- Compare the value (old value) pointed by the pointer and the given compare value
- If old value is the same as the compare value, make the value pointed by the pointer as the new given value,
- Return the old value

There are PTX instructions for CAS and other atomics.

http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_3.0.pdf



Atomic Functions

■ `atomicCAS` pseudo implementation

```
int atomicCAS( int *address,
    int compare, int val)
{ // Made up keyword
    __lock(address) {
        int old = *address;
        *address = (old == compare) ? val : old;
        return old;
    }
}
```



Atomic Functions

■ `atomicCAS` pseudo implementation

```
int atomicCAS( int *address,
    int compare, int val)
{ // Made up keyword
    __lock(address) {
        int old = *address;
        *address = (old == compare) ? val : old;
        return old;
    }
}
```



Atomic Functions

■ `atomicCAS` pseudo implementation

```
int atomicCAS( int *address,
    int compare, int val)
{ // Made up keyword
    __lock(address) {
        int old = *address;
        *address = (old == compare) ? val : old;
    return old;
}
}
```



Atomic Functions

■ Example:

```
*addr = 1;
```

```
atomicCAS(addr, 1, 2);
```

```
atomicCAS(addr, 1, 3);
```

```
atomicCAS(addr, 2, 3);
```



Atomic Functions

■ Example:

```
*addr = 1;
```

```
atomicCAS(addr, 1, 2);
```

```
// returns 1
```

```
atomicCAS(addr, 1, 3);
```

```
// *addr = 2
```

```
atomicCAS(addr, 2, 3);
```



Atomic Functions

■ Example:

```
*addr = 1;
```

```
atomicCAS(addr, 1, 2);
```

```
atomicCAS(addr, 1, 3);
```

```
atomicCAS(addr, 2, 3);
```

```
// returns 2
```

```
// *addr = 2
```



Atomic Functions

■ Example:

```
*addr = 1;
```

```
atomicCAS(addr, 1, 2);
```

```
atomicCAS(addr, 1, 3);
```

```
atomicCAS(addr, 2, 3);
```

```
// returns 2
```

```
// *addr = 3
```



Atomic Functions

- Again, how do you implement `atomicAdd` given `atomicCAS`?

```
__device__ int atomicAdd(  
    int *address, int val);
```



Atomic Functions

```
__device__ int atomicAdd(int *address, int val)
{
    int old = *address, assumed;
    do {
        assumed = old;
        old = atomicCAS(address,
                        assumed, val + assumed);
    } while (assumed != old);
    return old;
}
```



Atomic Functions

```
__device__ int atomicAdd(int *address, int val)
{
    int old = *address, assumed;
    do {
        assumed = old;
        old = atomicCAS(address,
                        assumed, val + assumed);
    } while (assumed != old);
    return old;
}
```

assumed;

Read original value at
*address.



Atomic Functions

```
__device__ int atomicAdd(int *address, int val)
{
    int old = *address, assumed;
    do {
        assumed = old;
        old = atomicCAS(address,
                        assumed, val + assumed);
    } while (assumed != old);
    return old;
}
```

If the value at
*address didn't
change, increment it.



Atomic Functions

```
__device__ int atomicAdd(int *address, int val)
{
    int old = *address, assumed;
    do {
        assumed = old;
        old = atomicCAS(address,
                        assumed, val + assumed);
    } while (assumed != old);
    return old;
}
```

Otherwise, loop until
atomicCAS succeeds.

The value of *address after this function
returns is not necessarily the original value
of *address + val, why?



Atomic Functions

■ Lots of atomics:

// Arithmetic
atomicAdd()
atomicSub()
atomicExch()
atomicMin()
atomicMax()
atomicAdd()
atomicDec()
atomicCAS()

// Bitwise
atomicAnd()
atomicOr()
atomicXor()

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions>



Atomic Functions

- How can threads from different blocks work together?
- Use atomics sparingly. Why?



Summary

- CUDA brings C to a multiprocessor architecture
 - ❖ Pros:
 - ◆ It's easy to use and program
 - ◆ Extensive, responsive support base, including developers
 - ◆ NVIDIA is actively supporting the project
 - ◆ Some applications are remarkably successful
 - ◆ Handmade SIMD code yields impressive results
 - ❖ Cons:
 - ◆ Performance depends heavily on the algorithm
 - ◆ Handmaking SIMD code requires learning a “new” style
 - ◆ Mostly exploiting capabilities is easy, but fully exploiting capabilities is difficult

