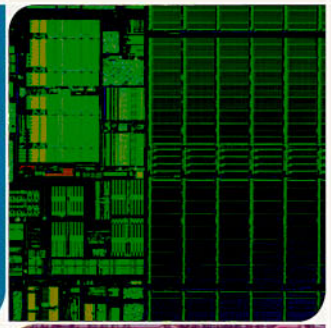
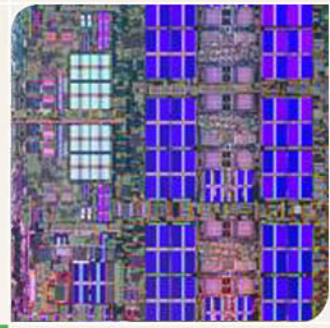
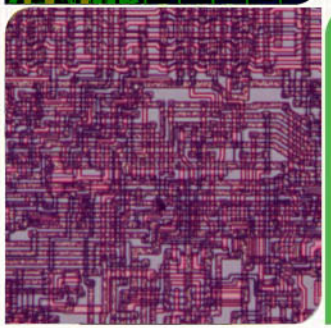
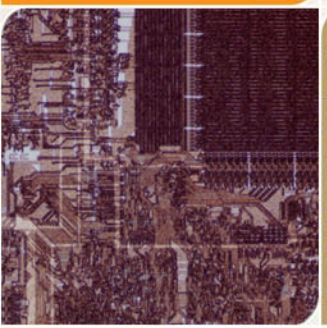




CS4290/CS6290

Fall 2011

Prof. Hyesoon Kim

Georgia Tech College of Computing



GPU Architecture

- Nvidia/AMD GPU→ many core architecture
- GPGPU→ high performance computing

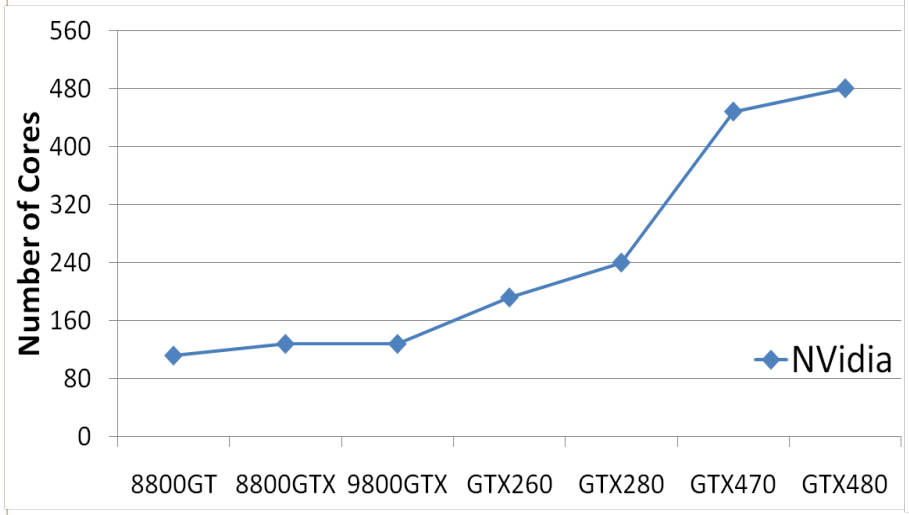


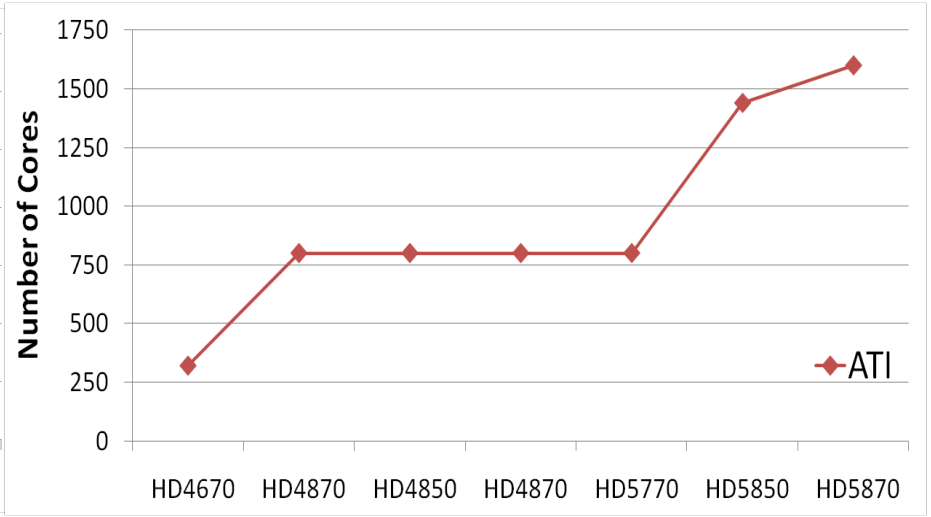
Outline of Part #1.0

- CUDA Programming 101
- GPGPU Architecture basic
- GPGPU Performance

GPU Architecture Trend

- Fixed pipelines → programmable cores → unified programmable cores → more cores → GPGPU support





[the slide is from Hong&Kim ISCA'10]



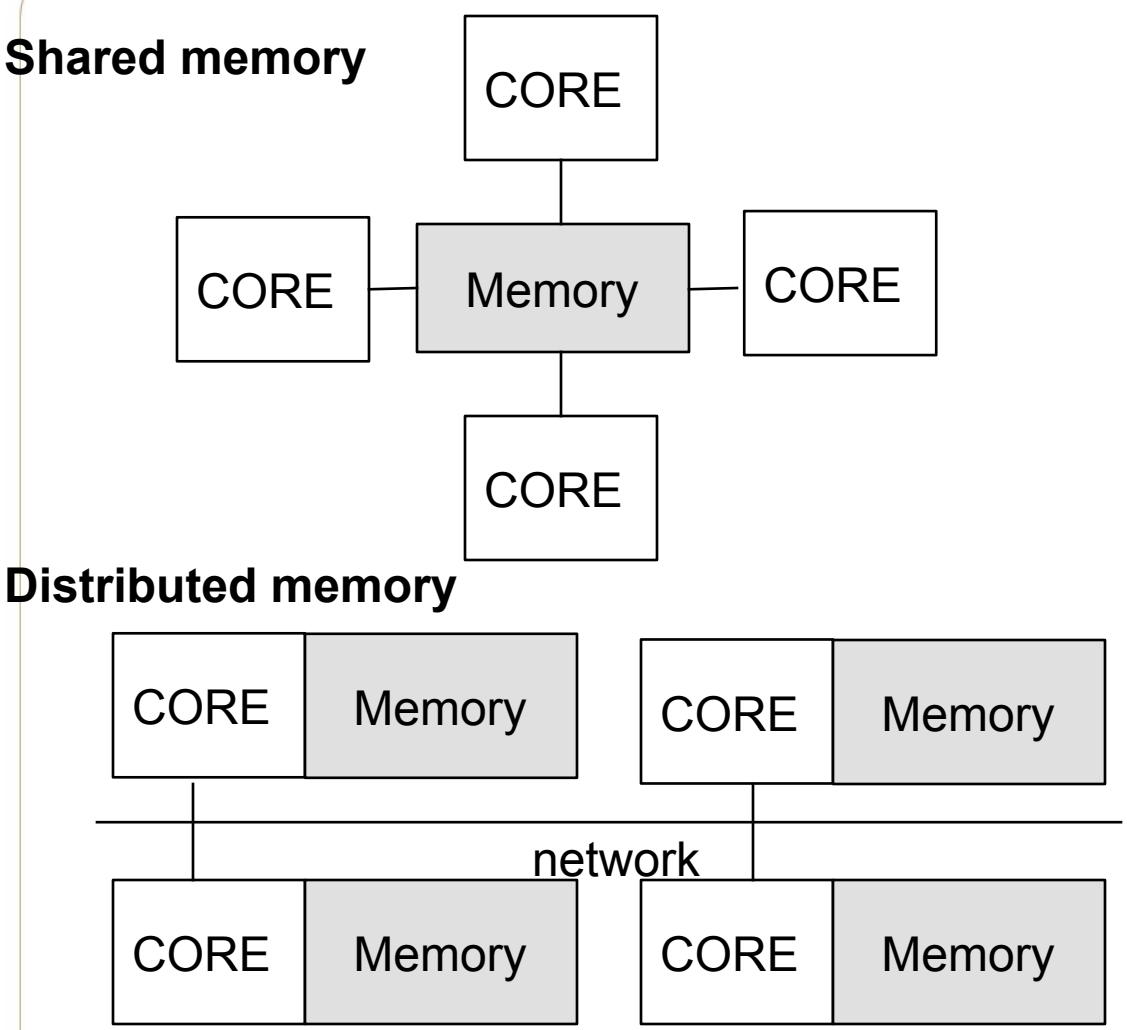
GPGPU Programming

- Become popular with CUDA (Compute Unified Device Architecture)
- CUDA (Based on Nvidia architectures)
  - Portland Group Compiler supports CUDA → x86
- OpenCL

Quick Summary of CUDA Programming Model

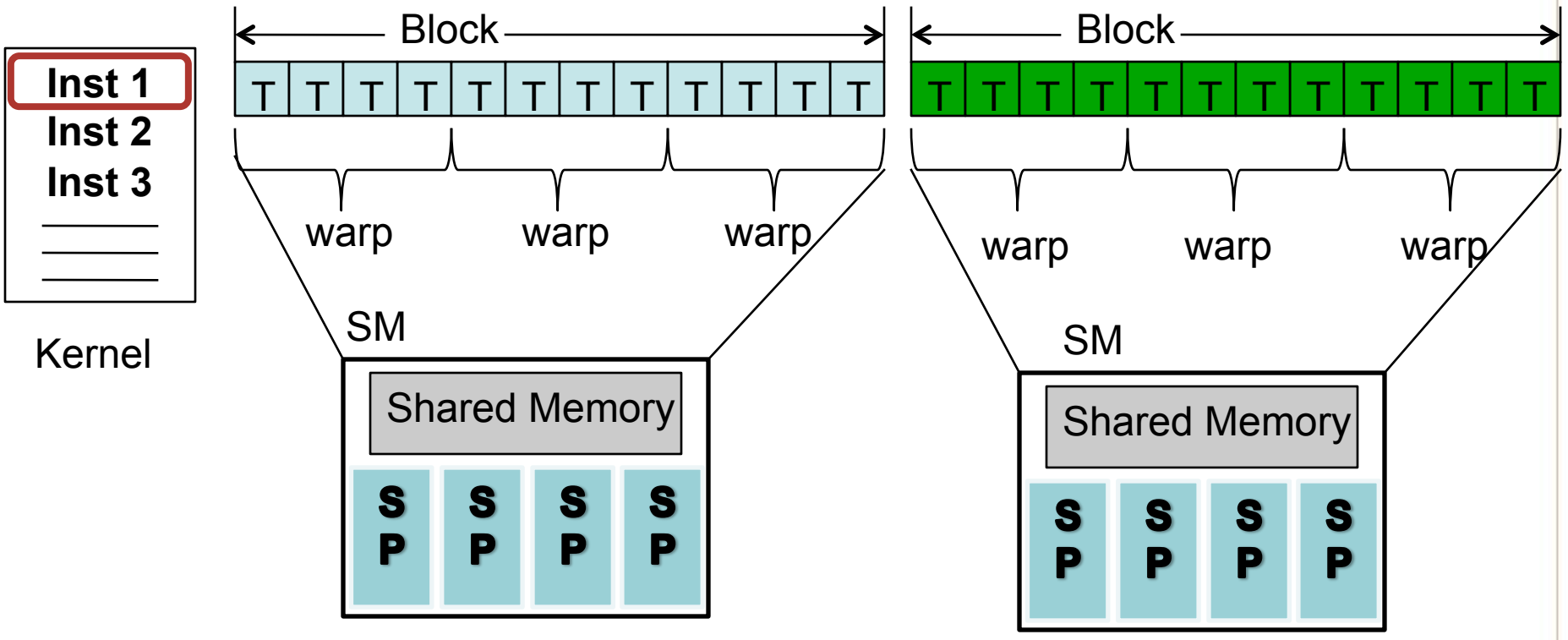
- SIMD or SIMT
  - Single instruction multiple data or single instruction multiple thread
- Unified Memory space (global memory space)
- Program hierarchy
  - Thread, block, kernel

Parallel Programming Models Based on Memory Spaces



Execution Model

- Thread & Block



Sing instruction multiple thread
Block = a group of thread which share “the shared memory space”
Warp

[the slide is from Hong&Kim ISCA’09]



Memory Space

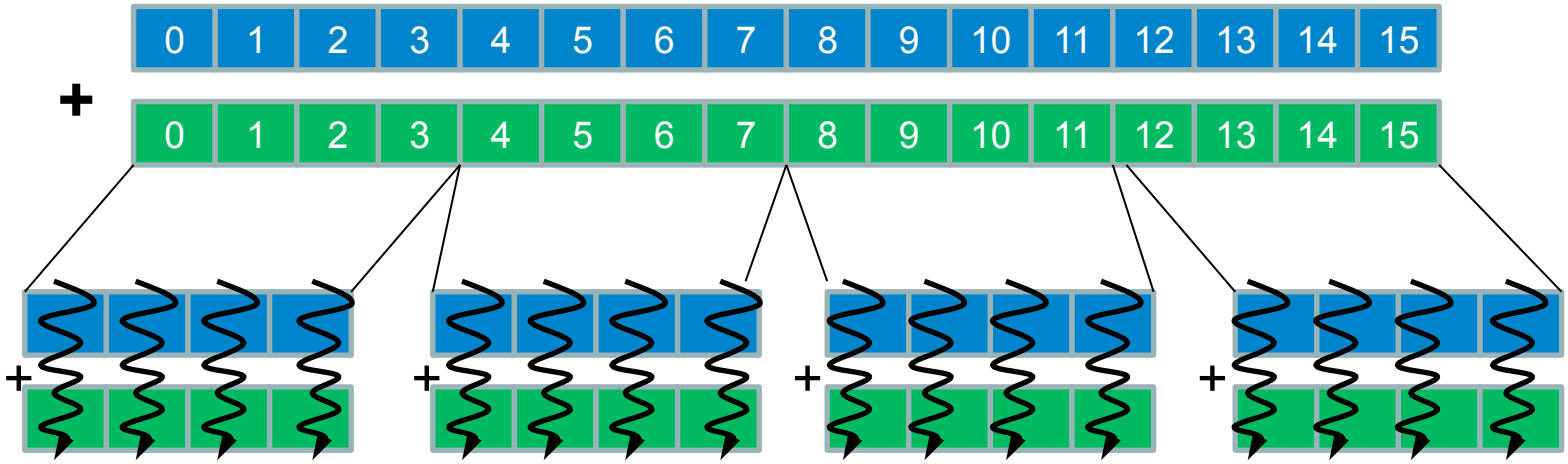
- Thread, block, and kernel have different memory spaces

	Space	~= CPU
Local Memory	Within Threads	Stack
Shared Memory	Within Blocks	Distributed memory space
Global Memory	All	Centralized storage
Constant Memory	All	Centralized read-only storage (very small)
Texture Memory	All	Centralized read-only storage (medium size, 2D- cache)

Memory Data Indexing

- SIMT-execution model
- Use thread id and block id to index data

Let's assume N=16, blockDim=4 → 4 blocks



blockIdx.x = 0
blockDim.x = 4
threadIdx.x = 0,1,2,3
Idx= 0,1,2,3

blockIdx.x = 1
blockDim.x = 4
threadIdx.x = 0,1,2,3
Idx= 4,5,6,7

blockIdx.x = 2
blockDim.x = 4
threadIdx.x = 0,1,2,3
Idx= 8,9,10,11

blockIdx.x = 3
blockDim.x = 4
threadIdx.x = 0,1,2,3
Idx= 12,13,14,15

1D, 2D, 3D data structures

CPU code

```
for (ii = 0; ii < 100; ++ii) {
C[ii] = A[ii] + B[ii];
}
```

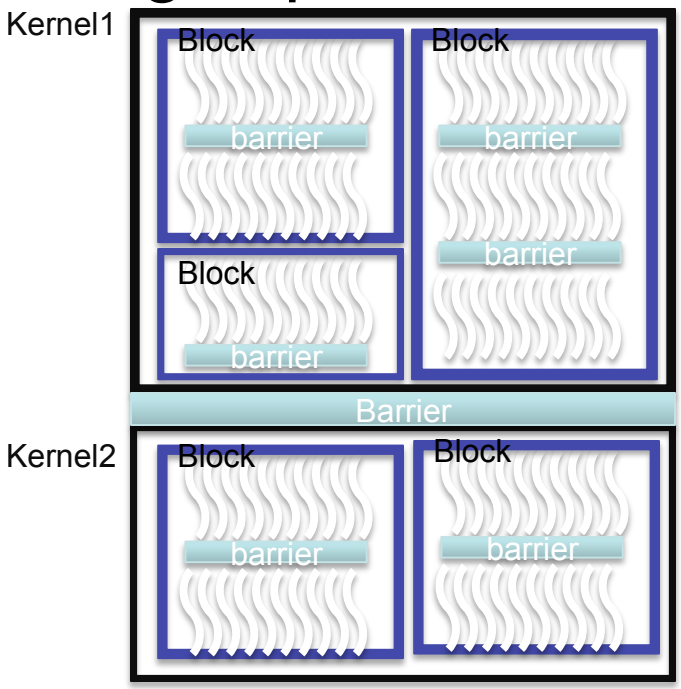
CUDA code

```
// there are 100 threads
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varA = aa[tid];
  int varB = bb[tid];
  C[tid] = varA + varB;
}
```

- A kernel is executed as a grid of thread blocks
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Loop index in sequential loop
  - Use thread ids, block ids
  - 1D array index= c1*threadId.x + c2*block Id.x
  - 2D array index = c1*threadId.x +c2*blockId.x+c3*threadId.y +c4*blockId.y

Synchronization Model

- Bulk-Synchronous Parallel (BSP) program (Valiant [90])
- Synchronization within blocks using explicit barrier
- Implicit barrier across kernels
  - Kernel 1 → Kernel 2
  - C.f.) Cuda 3.x



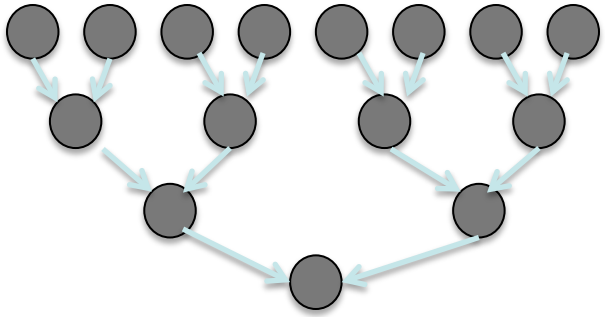
MIMD with CUDA

- Use thread id to write serial programs or reduce the number of running threads

If (threadId.x%==2)

If (threadId.x%==4)

If (threadId.x%==8)



(reduction example)

- Use block id to generate MIMD effects
  - If (blockId.x == 1) do work 1
  - If (blockid.x == 2) do work 2



Global Communications

- Use multiple kernels
- Write to same memory addresses
  - Behavior is not guaranteed
  - Data race
- Atomic operation
  - No other threads can write to the same location
  - Memory Write order is still arbitrary
    - `Keep being updated: atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`
- Performance degradation
  - Fermi increases atomic performance by 5x to 20x (M. Shebanow)



New Programming Features in Fermi

- Supporting pointers
  - Limited stacks
- Recursive programming
- Concurrent Kernel executions from the same application
  - Efficient pipelining parallel program paradigm
- More…



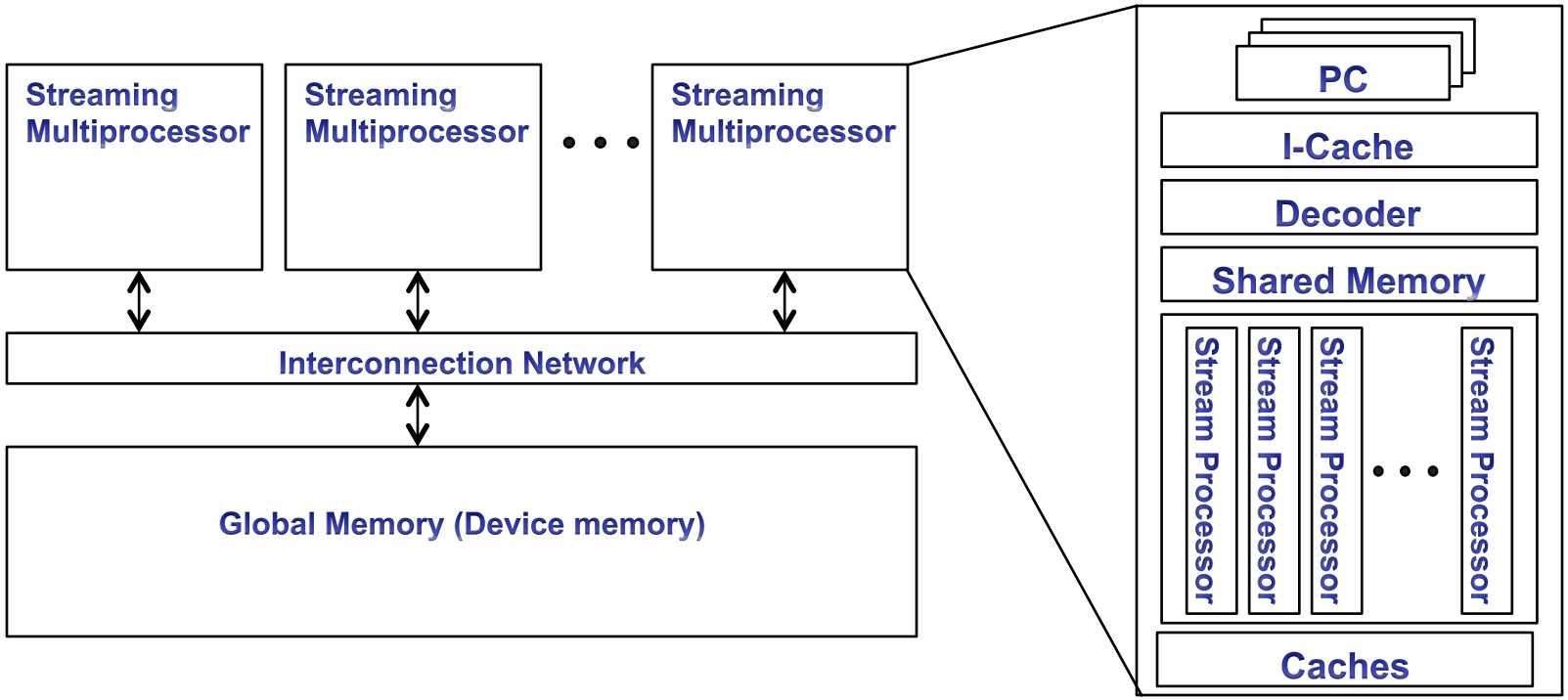
OpenCL vs. CUDA

	OpenCL	CUDA
Execution Model	Work-groups/work-items	Block/Thread
Memory model	Global/constant/local/private	Global/constant/shared/local + Texture
Memory consistency	Weak consistency	Weak consistency
Synchronization	Synchronization using a work-group barrier (between work-items)	Using synch_threads Between threads



GPU ARCHITECTURE 101

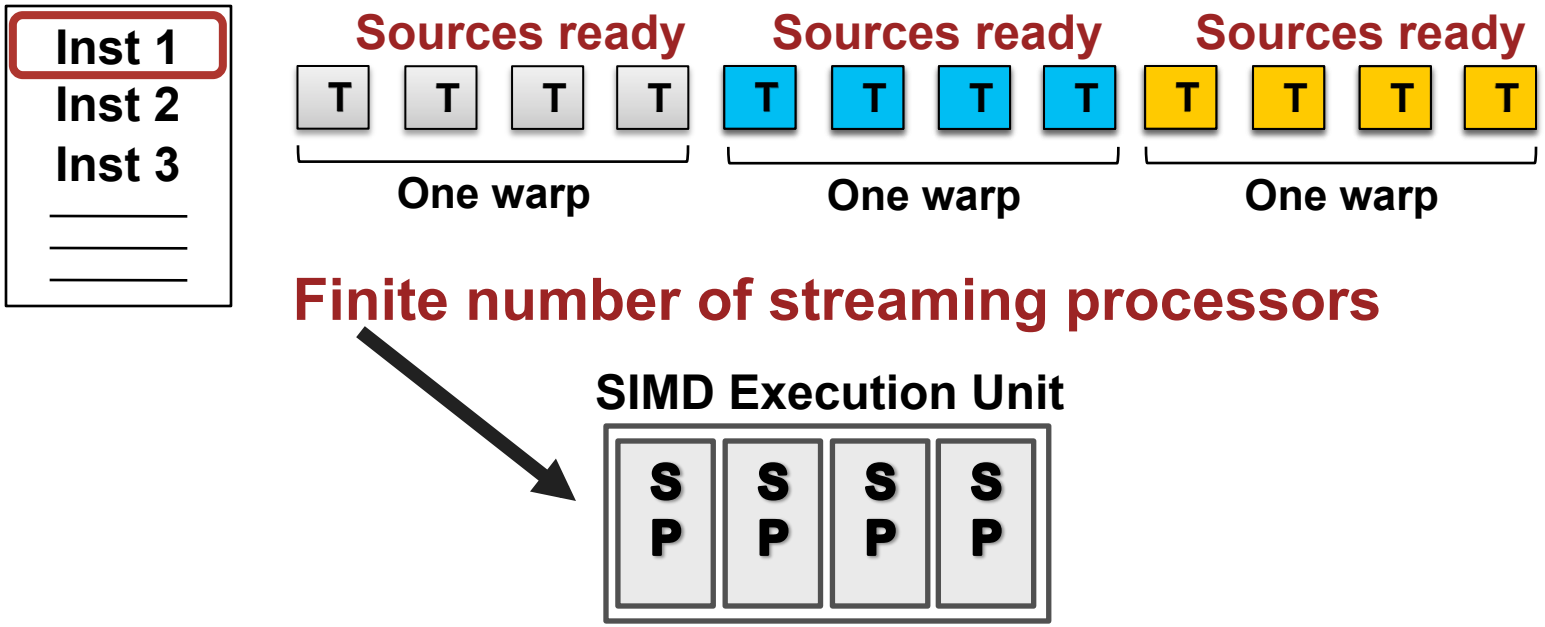
Overview of GPU (Tesla) Architecture



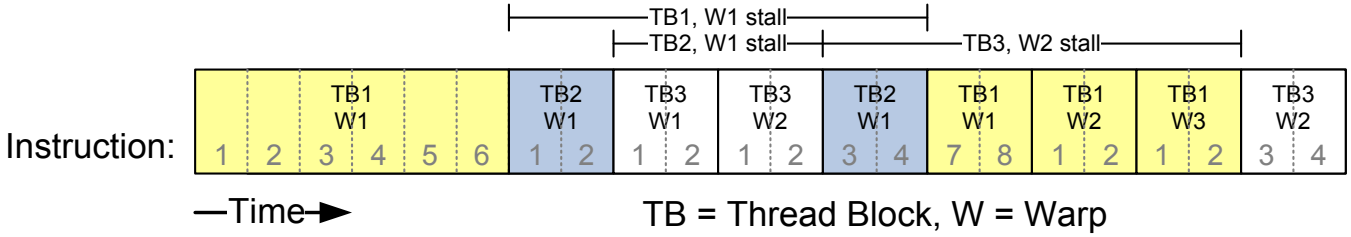
Execution Unit: Warp

- **Warp is the basic unit of execution**
  - A group of threads (e.g. 32 threads for the Tesla GPU architecture)

Warp Execution



Pipeline

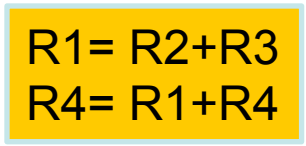


Kirk & Hwu

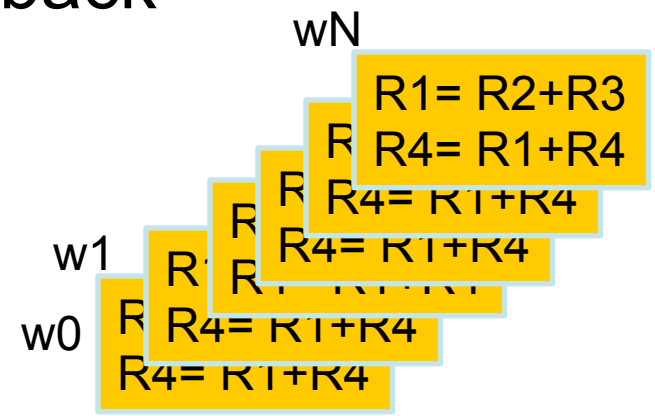
- Fetch
  - One instruction for each warp (could be further optimizations)
  - Round Robin, Greedy-fetch (switch when stall events such as branch, I-cache misses, buffer full)
- Thread scheduling polices
  - Execute when all sources are ready
  - In-order execution within warps
  - Scheduling polices: Greedy-execution, round-robin

Register Read & ILP & TLP

- Register read is fully pipelined.
- Back-to-back operation is in the critical path
- ILP across warps (~= TLP) can hide the latency of back-to-back



1 warp
24 cycles delay between 2 insts



1 warp
24 cycle delay is hidden by TLP

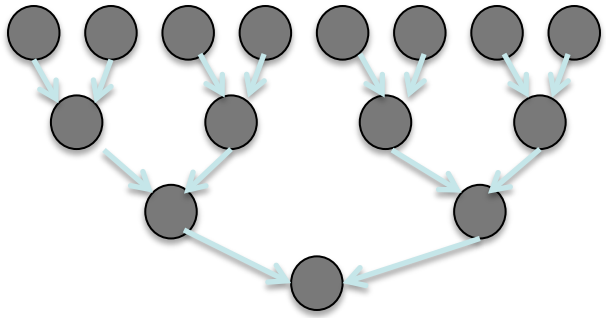
Handling Branch Instructions

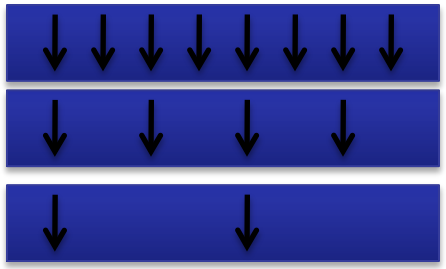
- Recall the reduction example

If (threadId.x%==2)

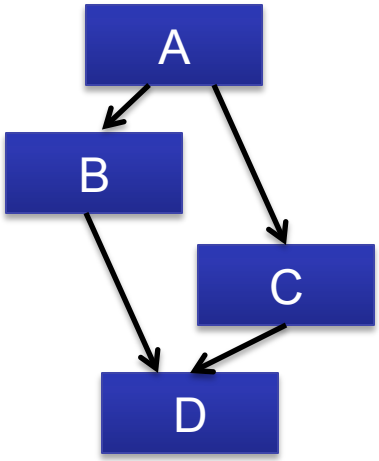
If (threadId.x%==4)

If (threadId.x%==8)



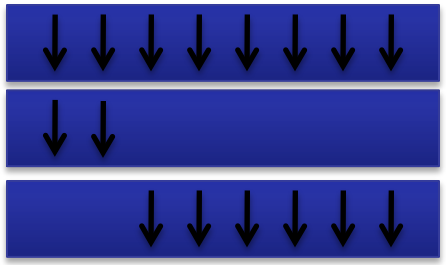


- What about other threads?
- What about different paths?




```
If (threadid.x>2) {
        do work B}
else {
        do work C
}
```

Divergent branch!



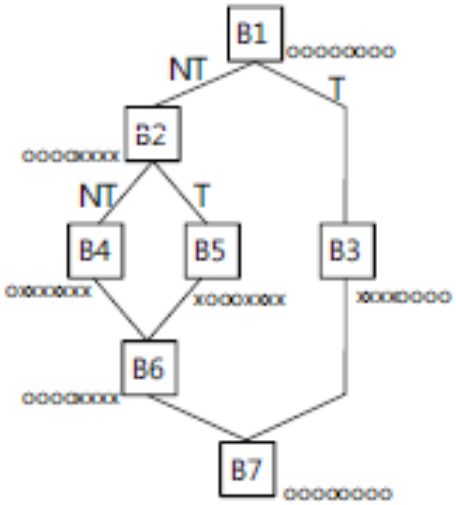


Divergent Branches

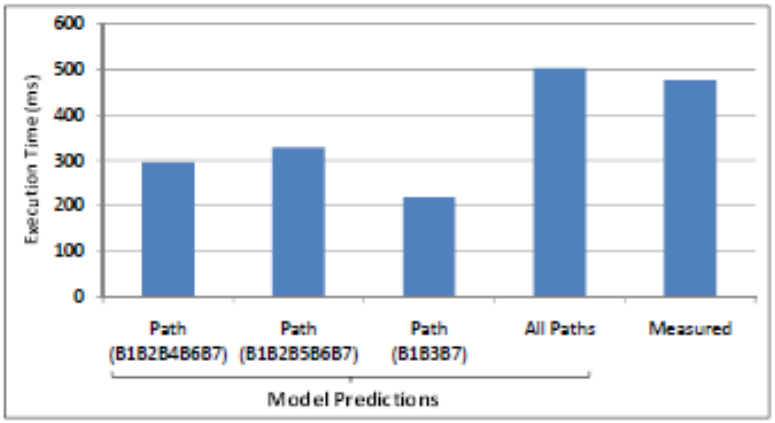
- All branch conditions are serialized and will be executed
  - Parallel code → sequential code
- Divergence occurs within a warp granularity.
- It's a performance issue
  - Degree of nested branches
- Depending on memory instructions, (cache hits or misses), divergent warps can occur
  - Dynamic warp subdivision [Meng'10]
- Hardware solutions to reduce divergent branches
  - Dynamic warp formation [Fung'07]
- On-the-fly elimination
  - Pure software solution [Zhang'11]
- Block compaction: [Fung and Aamodt'11]

Divergent Branches Execution Time

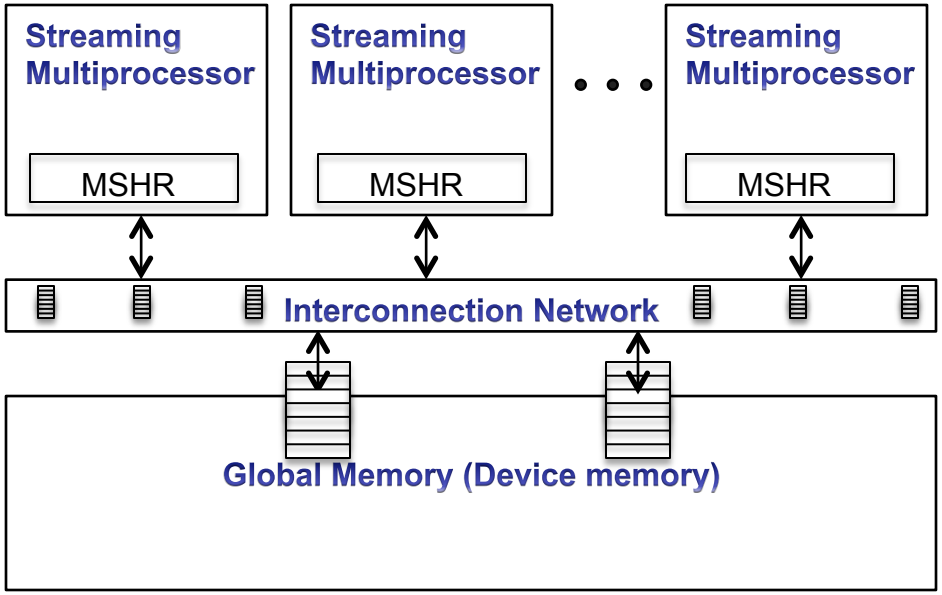
- Divergent branches serialize execution of




```
int mod = threadIdx.x & 31; // modulus
If (mod < A)                                   B1
{
    Memory_load
    FP Operations                              B3
}
else
{
    Memory_load
    if (mod < B)                               B2
    {
        FP Operations                          B5
    }
    else
    {
        FP Operations                          B4
    }
    FP Operations                              B6
}
Memory_store                                   B7
```



GPU Memory System



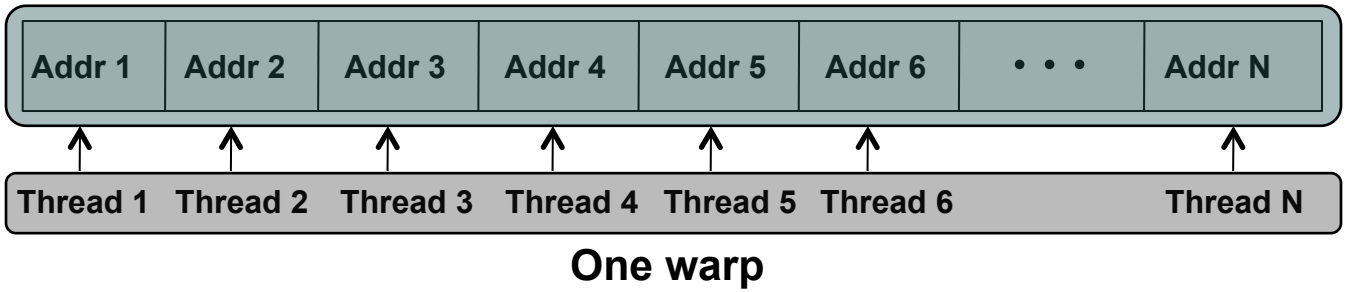
- Many levels of queues
- Large size of queues
- High number of DRAM banks
- Sensitive to memory scheduling algorithms
  - FRFCFS >> FCFS
- Interconnection network algorithm to get FRFCFS Effects
  - Yan'09,

Multiple Memory Transactions

One warp generates a memory request

One memory transaction

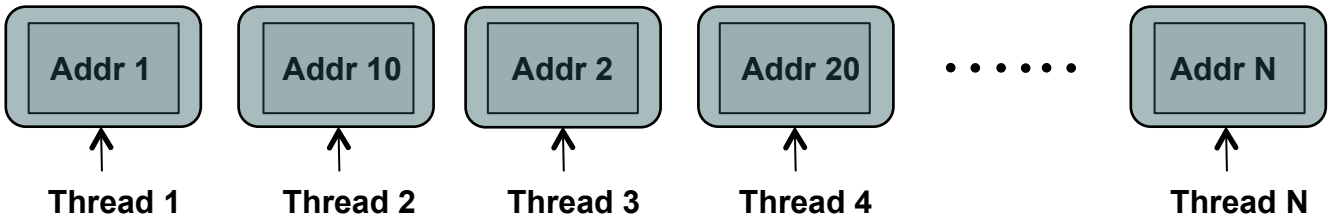
Coalesced memory access type



- Even coalesced memory accesses generate multiple transactions. 4B*32 = 128 B req size

Uncoalesced memory access type

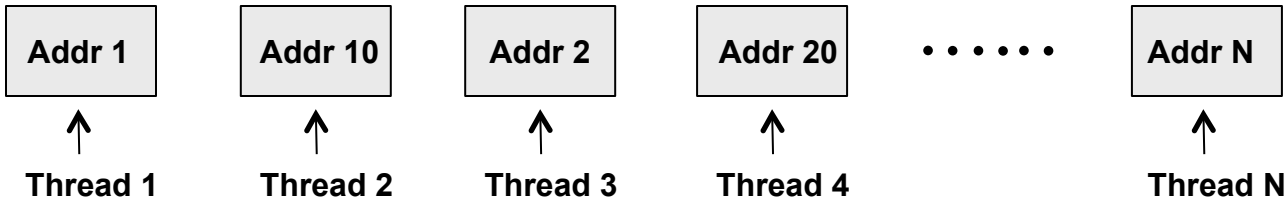
Multiple memory transactions

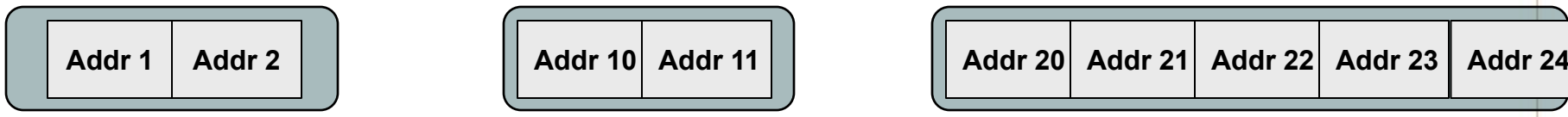


- More processing cycles for the uncoalesced case

Memory Transactions

- Compute capability > 1.2

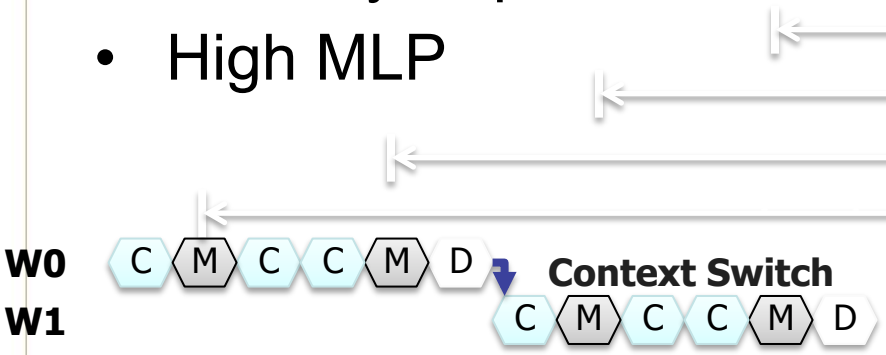




- Reduce the number of memory transactions as few as possible

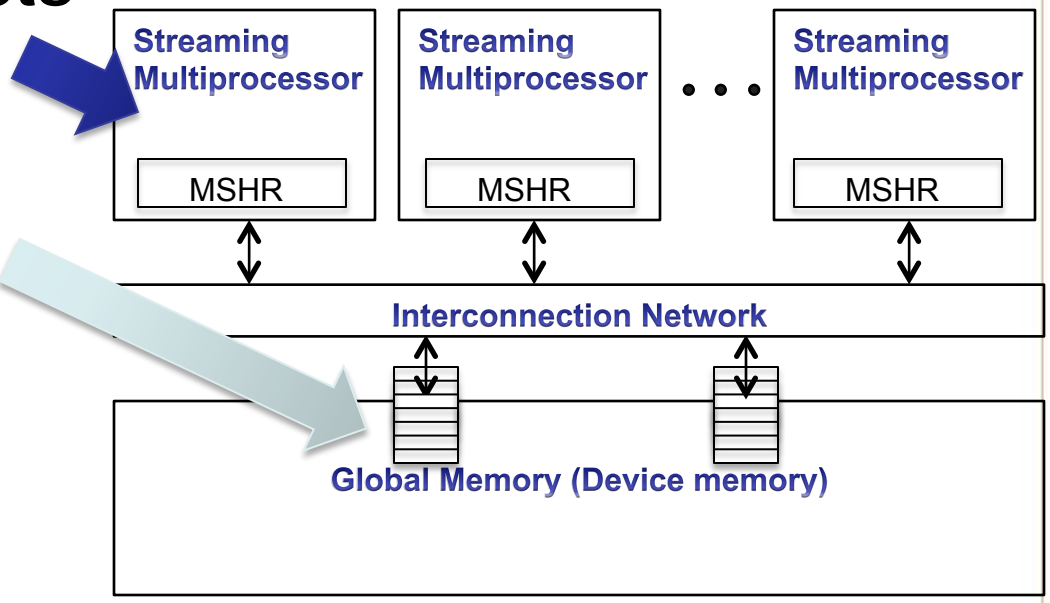
Multiple In-flight Memory Requests

- In-order execution but
- Warp cannot execute an instruction when sources are dependent on memory instructions, not when it generate memory requests
- High MLP



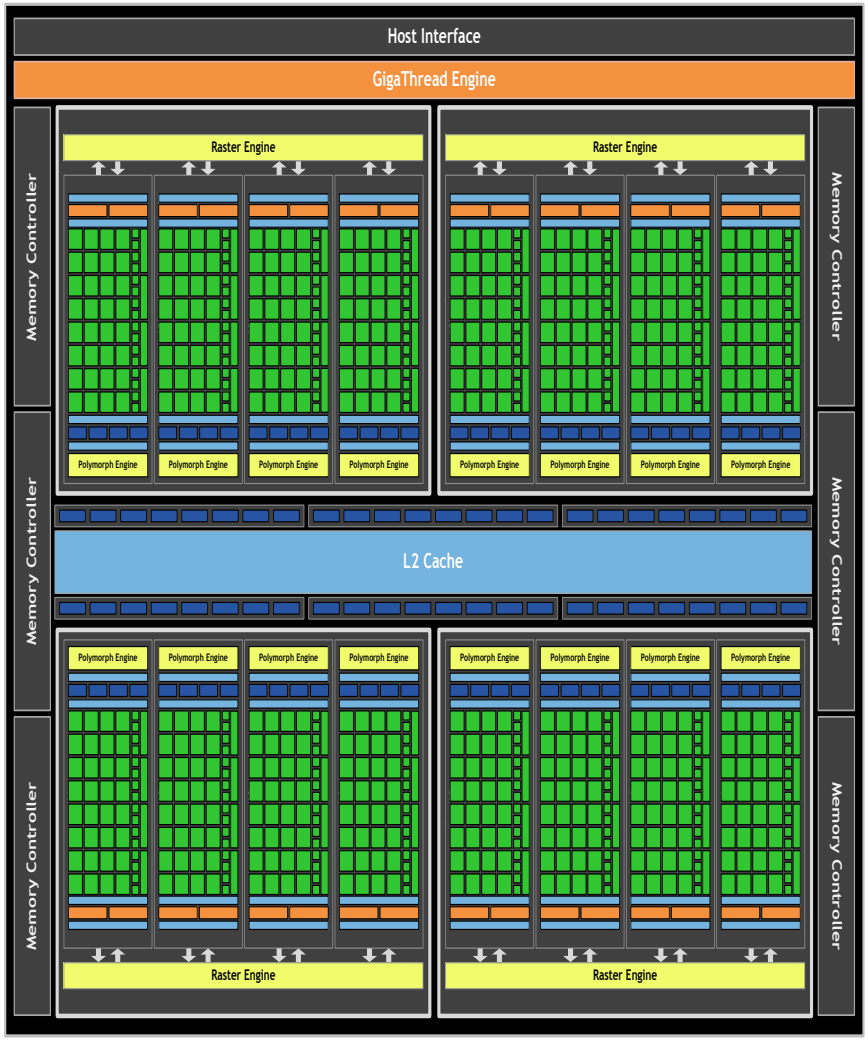
Same Data from Multiple Threads (SDMT)

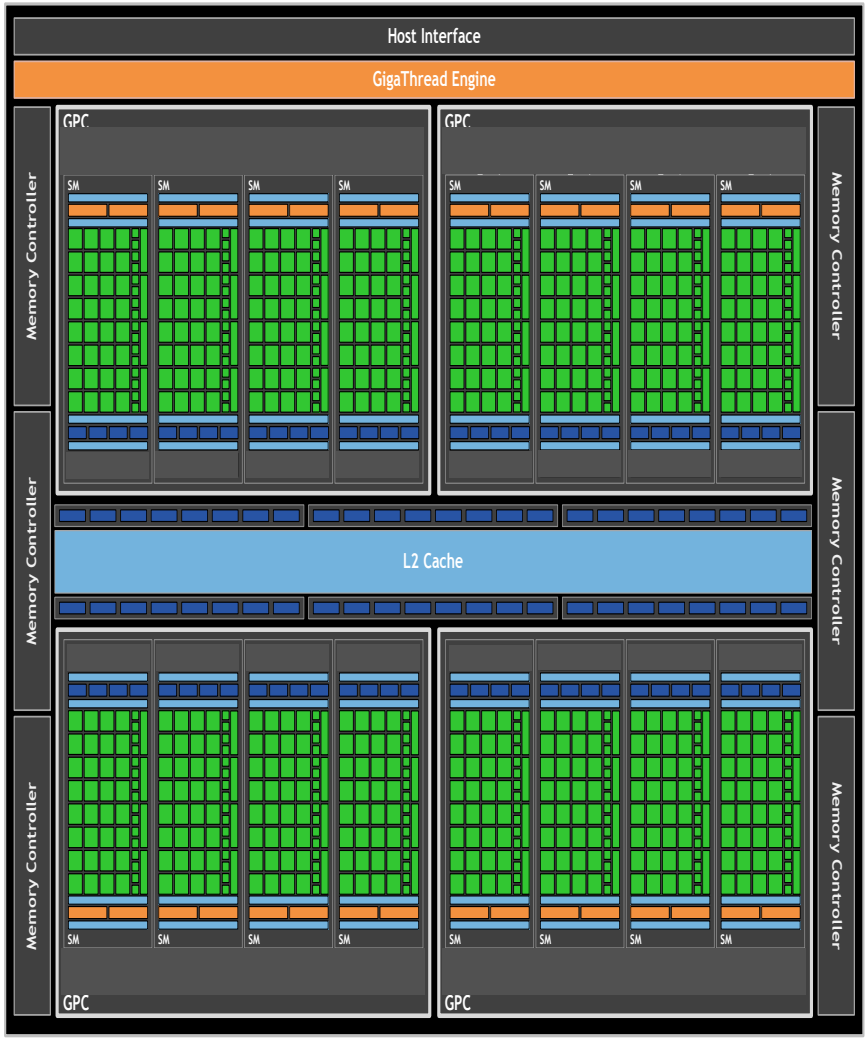
- High Merging Effects
  - Inter-core merging
  - Intra-core merging



- Techniques to take advantages of this SDMT
  - Compiler optimization[Yang'10]: increase memory reuse
  - Cache coherence [Tarjan'10]
  - Cache increase reuses

Fermi Architecture





Georgia Tech College of Computing