Graphics Processing Units (GPUs)

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.

Why Study GPUs?

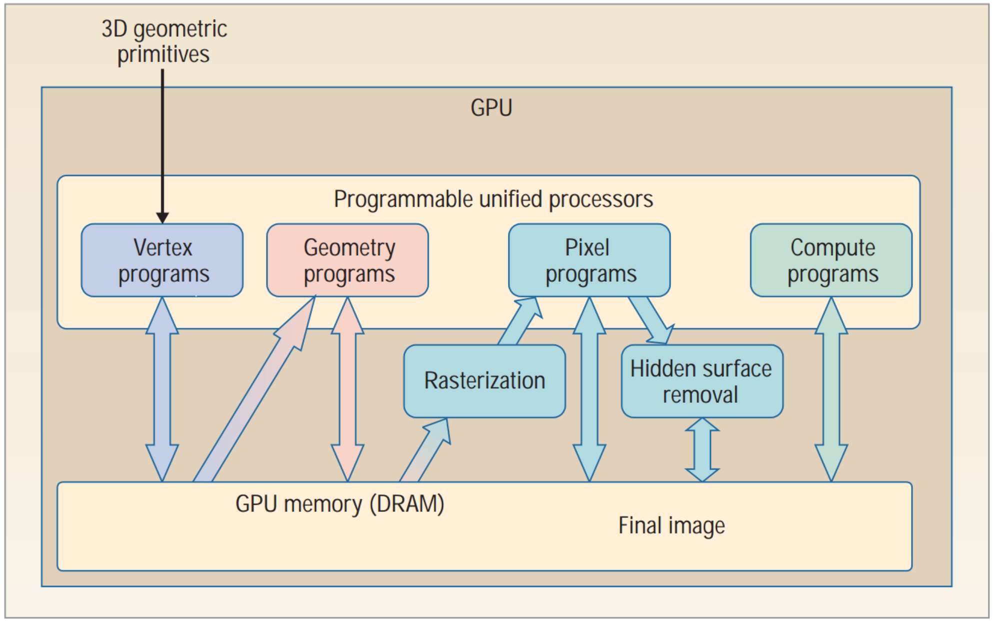
- Very successful commodity accelerator/co-processor
- GPUs combine two strategies to increase efficiency
  - Massive parallelism
  - Specialization
- Illustrates tension between performance and programmability in accelerators

Graphics Processors Timeline

- Until mid-90s
  - Most graphics processing in CPU
  - VGA controllers used to accelerate some display functions

- Mid-90s to mid-2000s
  - Fixed-function accelerators for 2D and 3D graphics
    - triangle setup & rasterization, texture mapping & shading
  - Programming:
    - OpenGL and DirectX APIs

Contemporary GPUs



Luebke and Humphreys, 2007

- Modern GPUs
  - Some fixed-function hardware (texture, raster ops, …)
  - Plus programmable data-parallel multiprocessors
  - Programming:
    - OpenGL/DirectX
    - Plus more general purpose languages (CUDA, OpenCL, …)

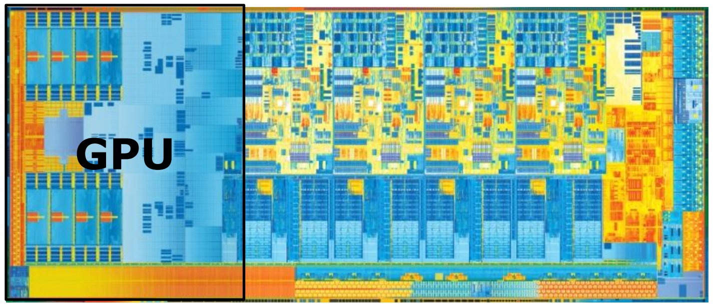
GPUs in Modern Systems

- Discrete GPUs
  - PCIe-based accelerator
  - Separate GPU memory

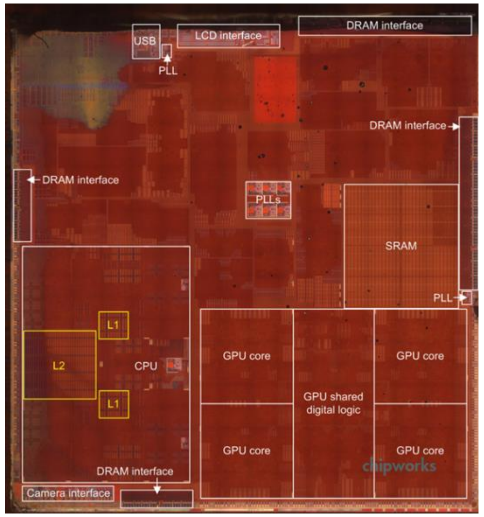
Nvidia Kepler

- Integrated GPUs
  - CPU and GPU on same die
  - Shared main memory and last-level cache

- Pros/cons?

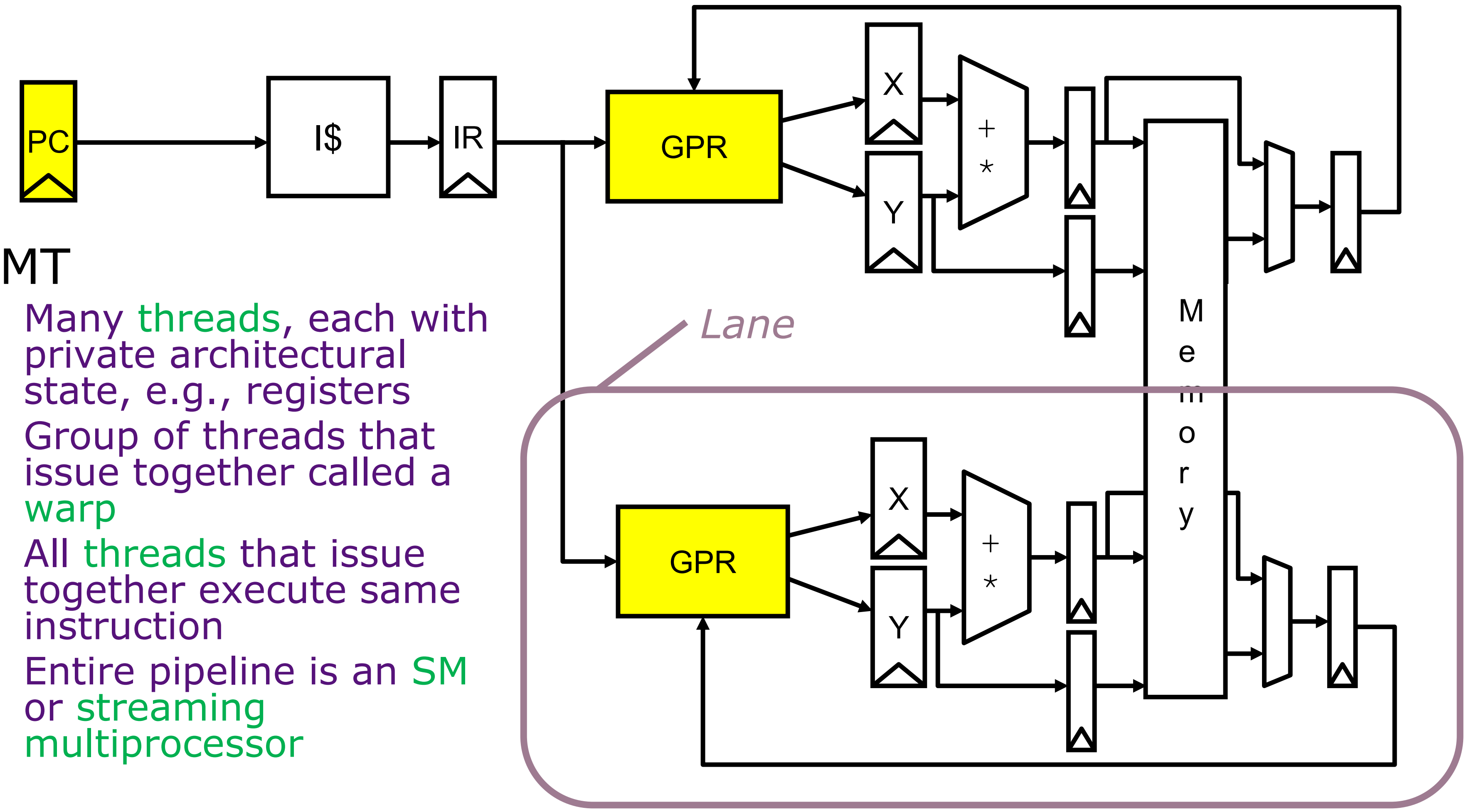


Intel Ivy Bridge, 22nm 160mm^2



Apple A7, 28nm TSMC, 102mm^2

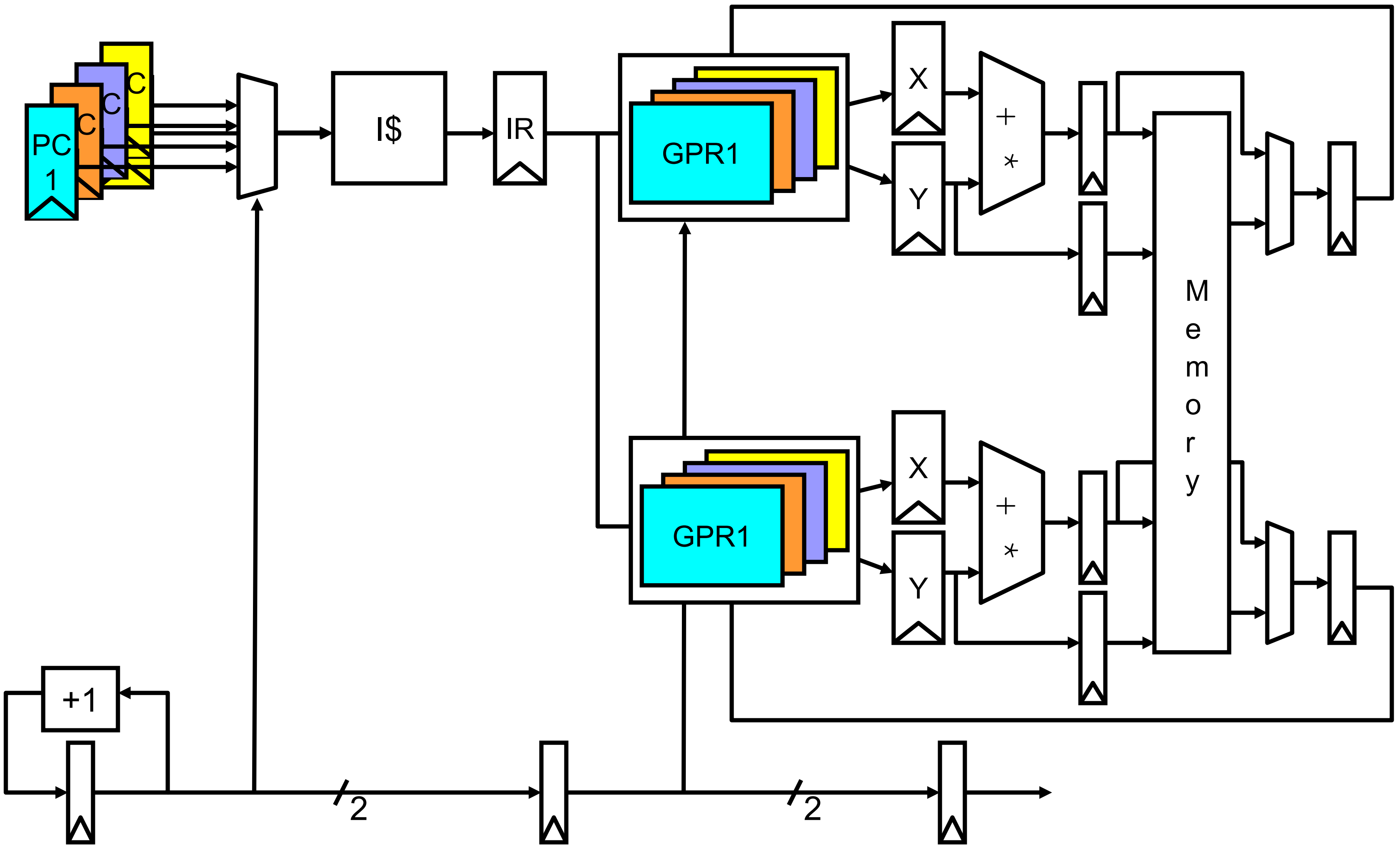
Single Instruction Multiple Thread



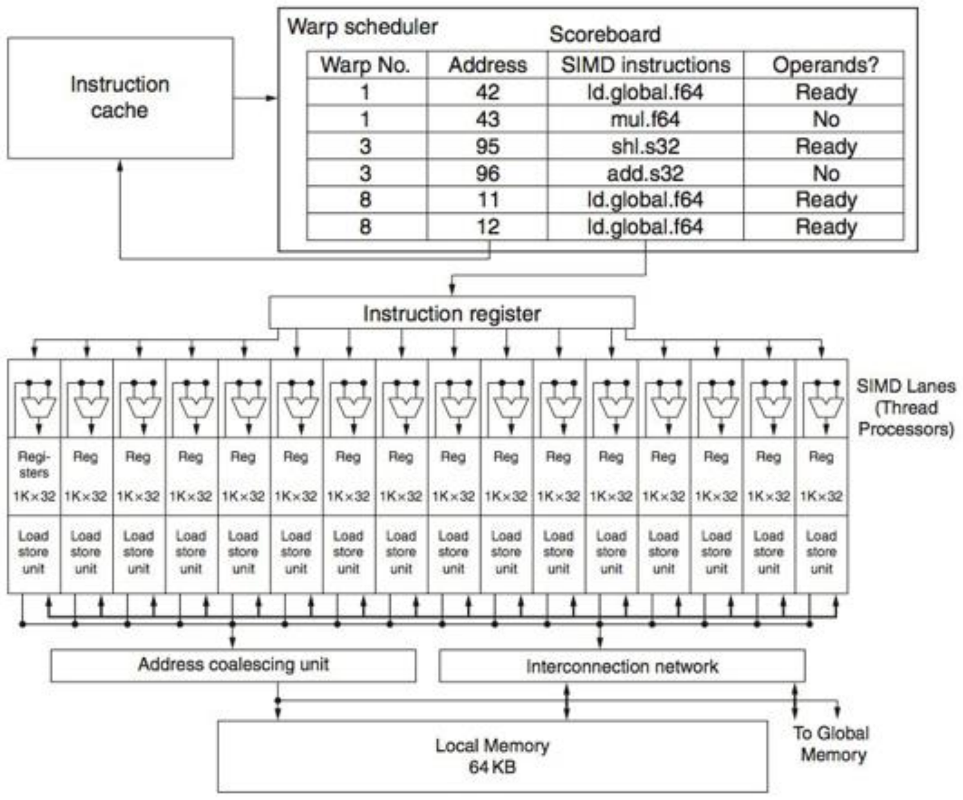
SIMT
- Many threads, each with private architectural state, e.g., registers
- Group of threads that issue together called a warp
- All threads that issue together execute same instruction
- Entire pipeline is an SM or streaming multiprocessor

green-> Nvidia terminology

Multithreading + Single Instruction Multiple Thread



Streaming Multiprocessor Overview



Warp No.	Address	SIMD instructions	Operands?
1	42	ld.global.f64	Ready
1	43	mul.f64	No
3	95	shl.s32	Ready
3	96	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

- Each SM supports 10s of warps (e.g., 64 in Kepler) with 32 threads/warp
- Fetch 1 instr/cycle
- Issue 1 ready instr/cycle
  - Simple scoreboarding: all warp elements must be ready
- Instruction broadcast to all lanes
- Multithreading is the main latency-hiding mechanism

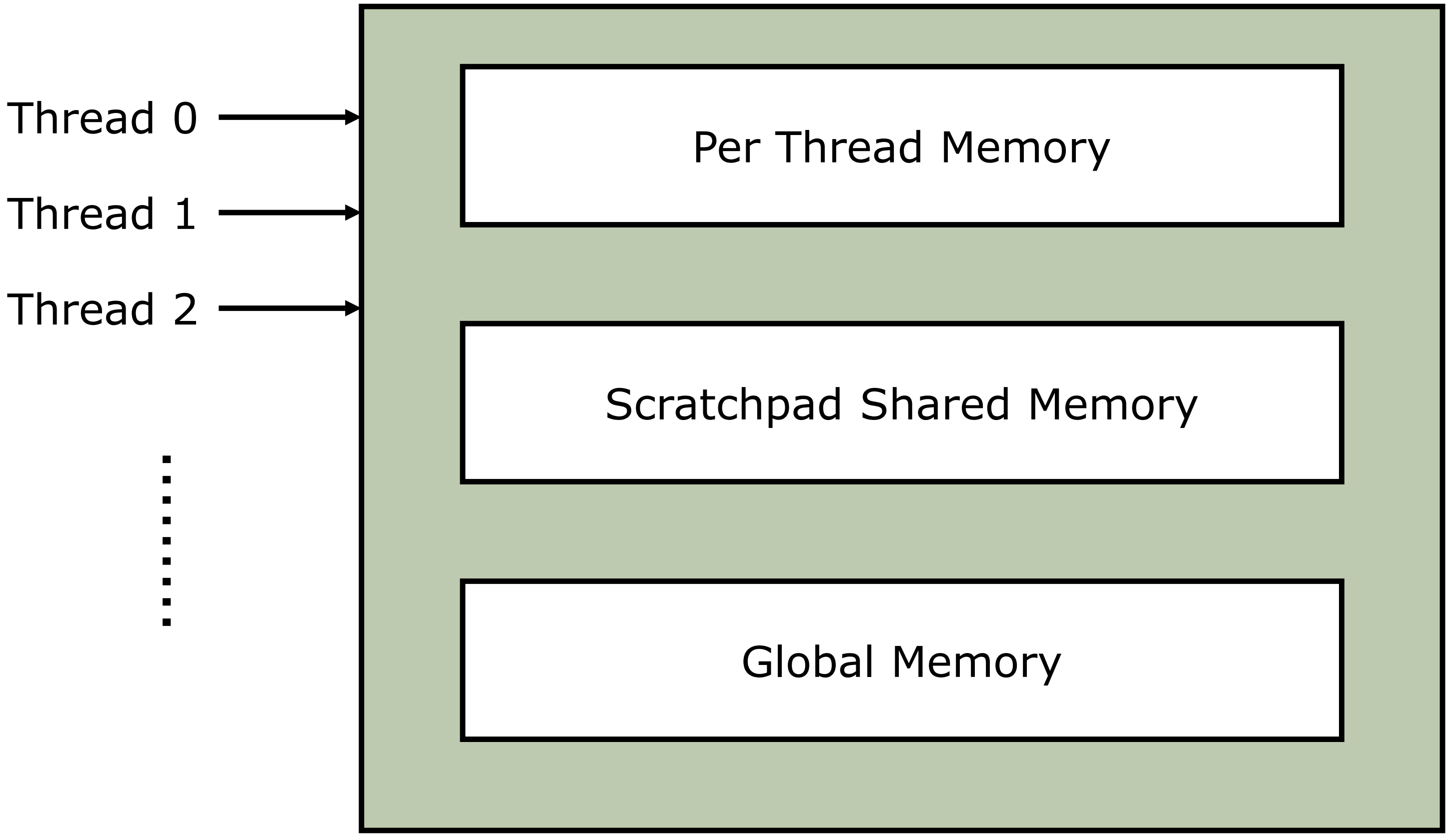
What average latency is needed to keep busy?

Context Size vs Number of Contexts

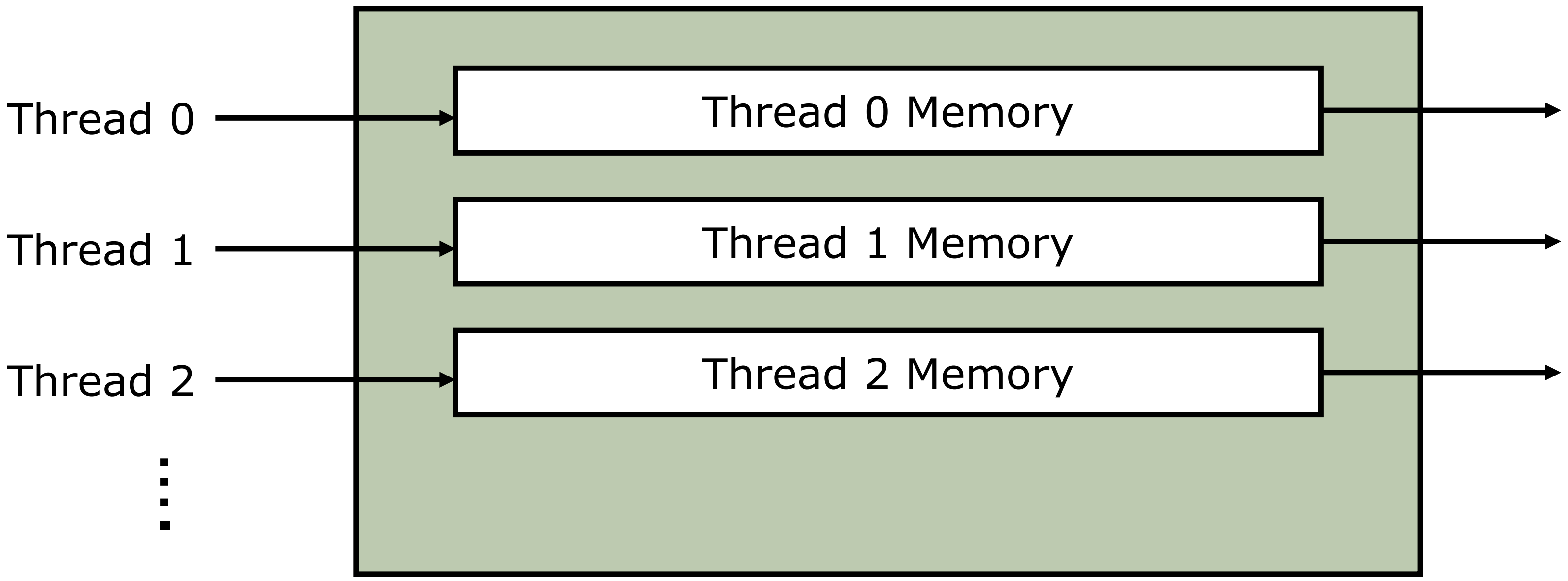
- SMs support a variable number of contexts based on required registers (and shared memory)
  - Few large contexts → Fewer register spills
  - Many small contexts → More latency tolerance
  - Choice left to the compiler

- Example: Kepler supports up to 64 warps
  - Max: 64 warps @ <=32 registers/thread
  - Min: 8 warps @ 256 registers/thread

Many Memory Types

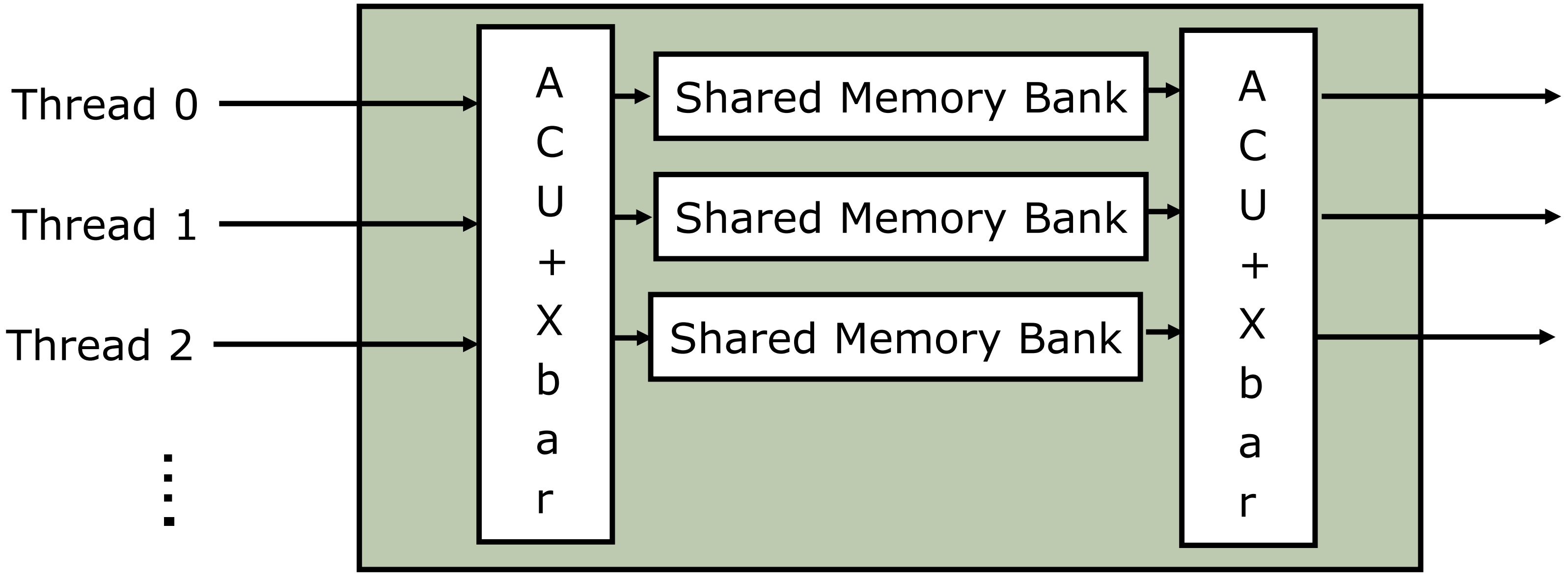


Private Per Thread Memory



- Private memory
  - No cross-thread sharing
  - Small, fixed size memory
    - Can be used for constants
  - Multi-bank implementation (can be in global memory)

Shared Scratchpad Memory

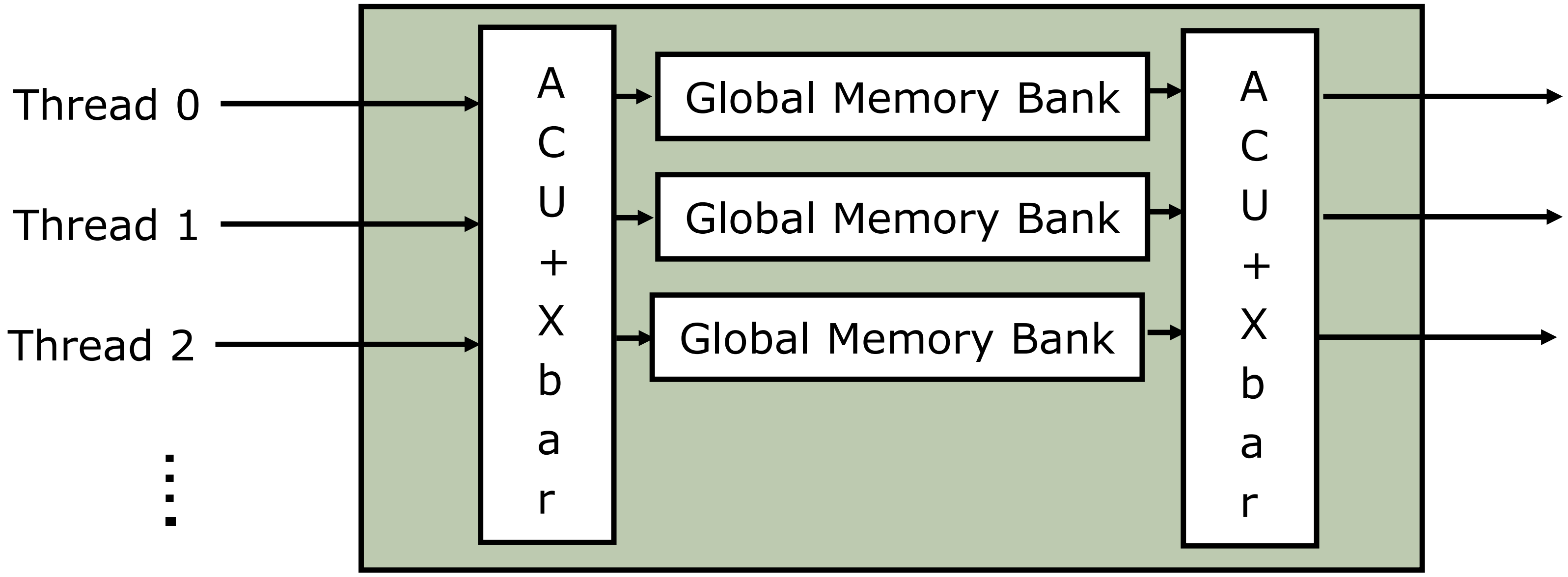


- Shared scratchpad memory (threads share data)
  - Small, fixed size memory (16K-64K per SM = ‘core’)
  - Banked for high bandwidth
  - Fed with address coalescing unit (ACU) + crossbar
    - ACU can buffer/coalesce requests

Memory Access Divergence

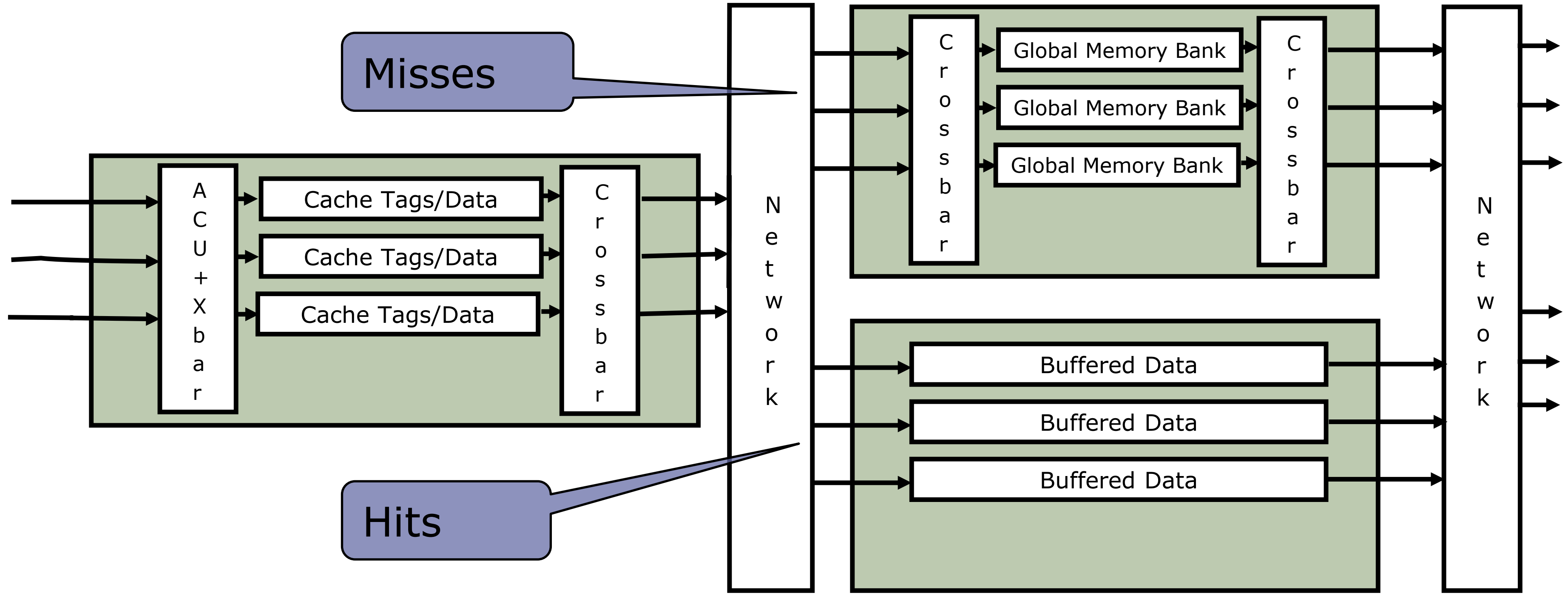
- All loads are gathers, all stores are scatters
- Address coalescing unit detects sequential and strided patterns, coalesces memory requests, but complex patterns can result in multiple lower bandwidth requests (memory divergence)
- Writing efficient GPU code requires most accesses to not conflict, even though programming model allows arbitrary patterns!

Shared Global Memory



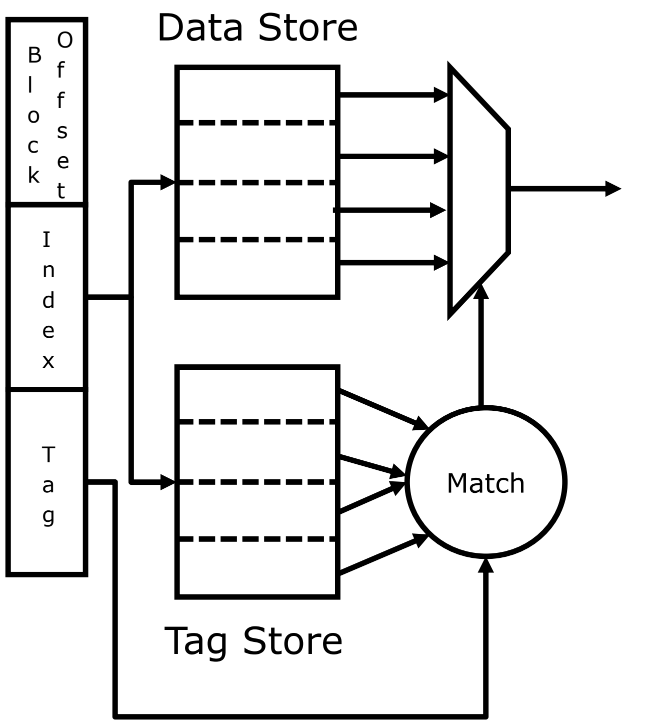
- Shared global memory
  - Large shared memory
  - Will suffer also from memory divergence

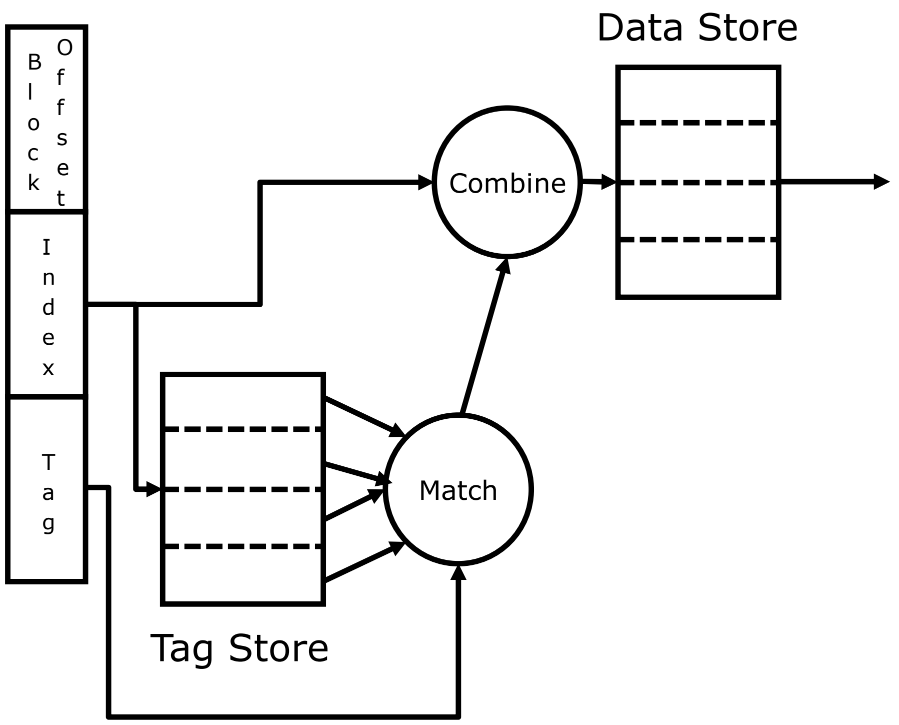
Shared Global Memory



- Memory hierarchy with caches
  - Cache to save memory bandwidth
  - Caches also enable compression/decompression of data

Serialized cache access





- Trade latency for power/flexibility
  - Only access data bank that contains data
  - Facilitate more sophisticated cache organizations
    - e.g., greater associativity

Handling Branch Divergence

- Similar to vector processors, but masks are handled internally
  - Per-warp stack stores PCs and masks of non-taken paths
- On a conditional branch
  - Push the current mask onto the stack
  - Push the mask and PC for the non-taken path
  - Set the mask for the taken path
- At the end of the taken path
  - Pop mask and PC for the non-taken path and execute
- At the end of the non-taken path
  - Pop the original mask before the branch instruction
- If a mask is all zeros, skip the block

Example: Branch Divergence

Assume 4 threads/warp,
initial mask 1111

```
if (m[i] != 0) {                 1
  if (a[i] > b[i]) {             2
    y[i] = a[i] - b[i];
  } else {                       3
    y[i] = b[i] - a[i];
  }                              4
} else {                         5
  y[i] = 0;
}                                6
```

1 Push mask 1111
Push mask 0011
Set mask 1100

2 Push mask 1100
Push mask 0100
Set mask 1000

3 Pop mask 0100

4 Pop mask 1100

5 Pop mask 0011

6 Pop mask 1111

Optimization for branches that all go same way?

Branch divergence and locking

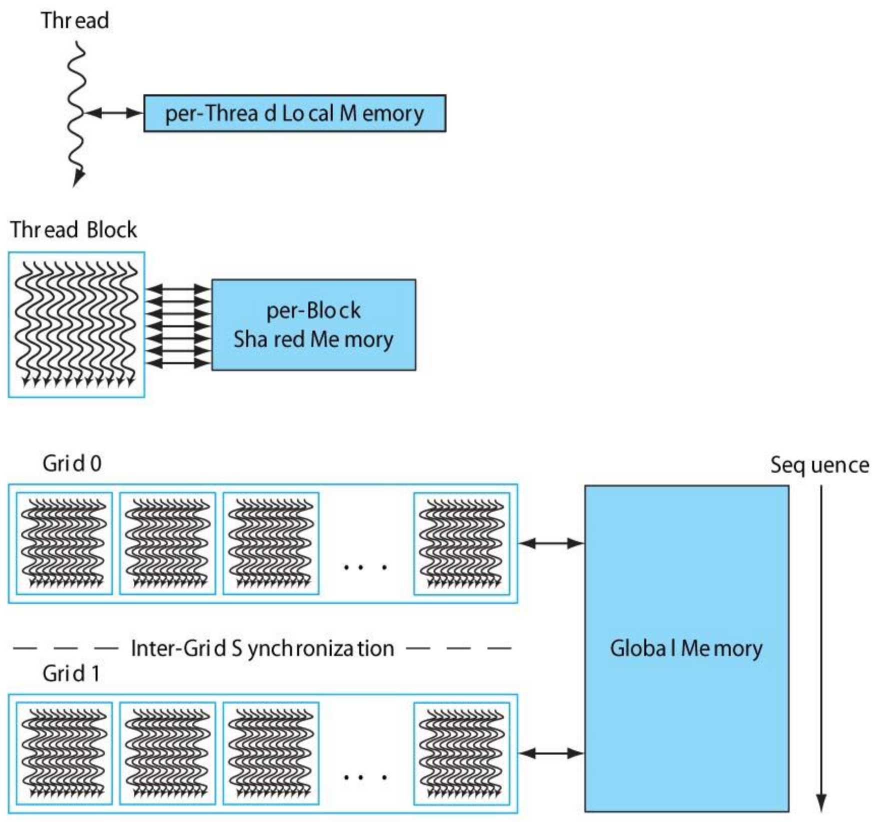
- Consider the following executing in multiple threads in a warp:

```
if (condition[i]) {
  while (locked(map0[i])){}
  lock(locks[map0[i]]);
} else {
  unlock(locks[map1[i]]);
}
```

where `i` is a thread id and `map0[]`, `map1[]` are permutations of thread ids.

What can go wrong here?

CUDA GPU Thread Model



- Single-program multiple data (SPMD) model
- Each context is a thread
  - Threads have registers
  - Threads have local memory
- Parallel threads packed in blocks
  - Blocks have shared memory
  - Threads synchronize with barrier
  - Blocks run to completion (or abort)
- Grids include independent blocks
  - May execute concurrently
  - Share global memory, but
  - Have limited inter-block synchronization

Code Example: DAXPY

C Code

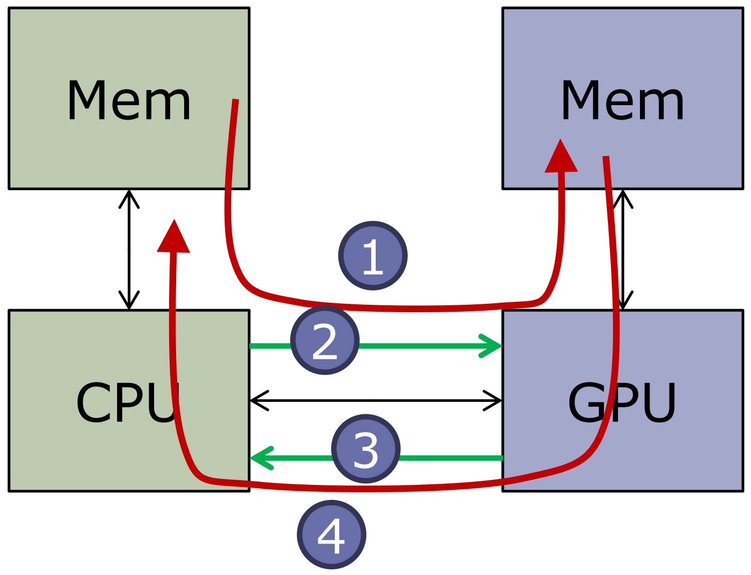
```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
                for (int i = 0; i < n; ++i)
                          y[i] = a*x[i] + y[i];
}
```

CUDA Code

```
// Invoke DAXPY with 256 threads per block
__host__
int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

- CUDA code launches 256 threads per block
- CUDA vs vector terminology:
  - Thread = 1 iteration of scalar loop (1 element in vector loop)
  - Block = Body of vectorized loop (VL=256 in this example)
  - Grid = Vectorizable loop

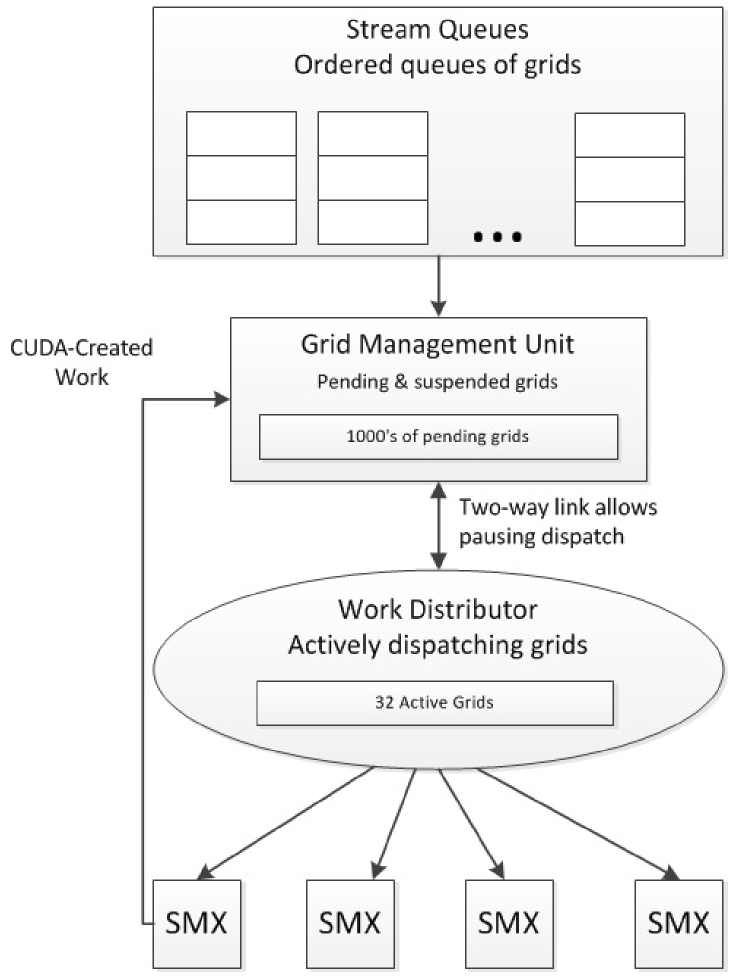
GPU Kernel Execution



1. Transfer input data from CPU to GPU memory
2. Launch kernel (grid)
3. Wait for kernel to finish (if synchronous)
4. Transfer results to CPU memory

- Data transfers can dominate execution time
- Integrated GPUs with unified address space
  → no copies, but CPU & GPU contend for memory

Hardware Scheduling



- Grids can be launched by CPU or GPU
  - Work from multiple CPU threads and processes
- HW unit schedules grids on SMs
  - Priority-based scheduling
- Multi-level scheduling
  - Limited number of active grids
  - More queued/paused

Synchronization

- Barrier synchronization within a thread block (__syncthreads())
  - Tracking simplified by grouping threads into warps
  - Counter tracks number of warps that have arrived to barrier
- Atomic operations to global memory
  - Read-modify-write operations (add, exchange, compare-and-swap, …)
  - Performed at the memory controller or at the L2
- Limited inter-block synchronization!
  - Can't wait for other blocks to finish

GPU ISA and Compilation

- GPU microarchitecture and instruction set change very frequently

- To achieve compatibility:
  - Compiler produces intermediate pseudo-assembler language (e.g., Nvidia PTX)
  - GPU driver JITs kernel, tailoring it to specific microarchitecture

- In practice, little performance portability
  - Code is often tuned to specific GPU architecture

System-Level Issues

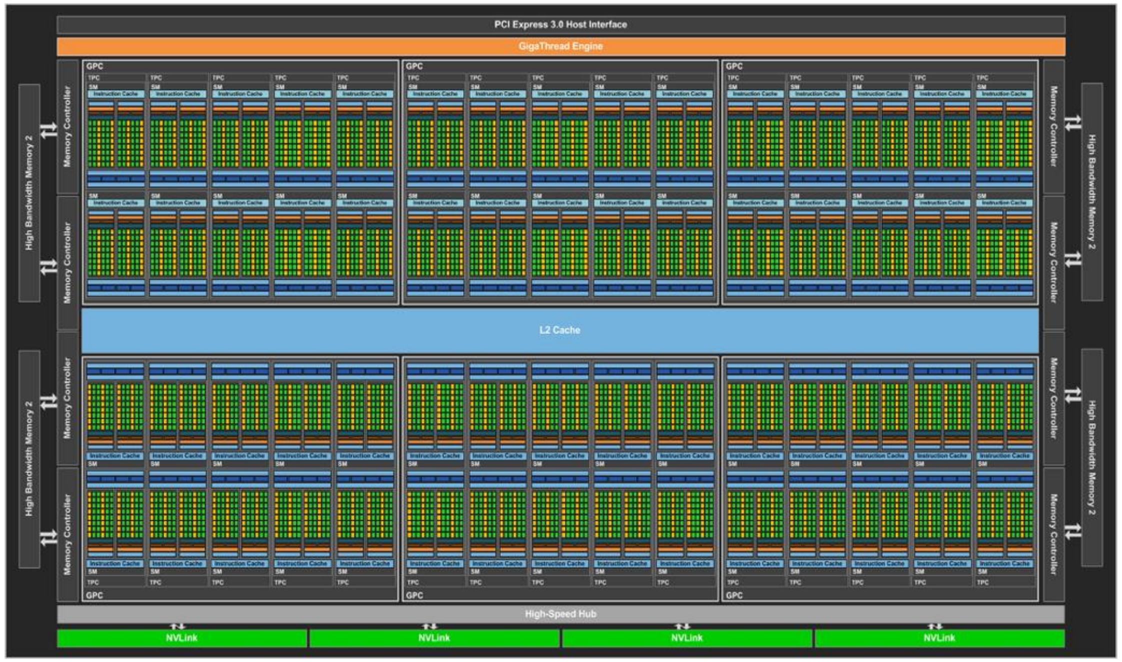
- Instruction semantics
  - Exceptions

- Scheduling
  - Each kernel is non-preemptive (but can be aborted)
  - Resource management and scheduling left to GPU driver, opaque to OS

- Memory management
  - First GPUs had no virtual memory
  - Recent support for basic virtual memory (protection among grids, no paging)
  - Host-to-device copies with separate memories (discrete GPUs)

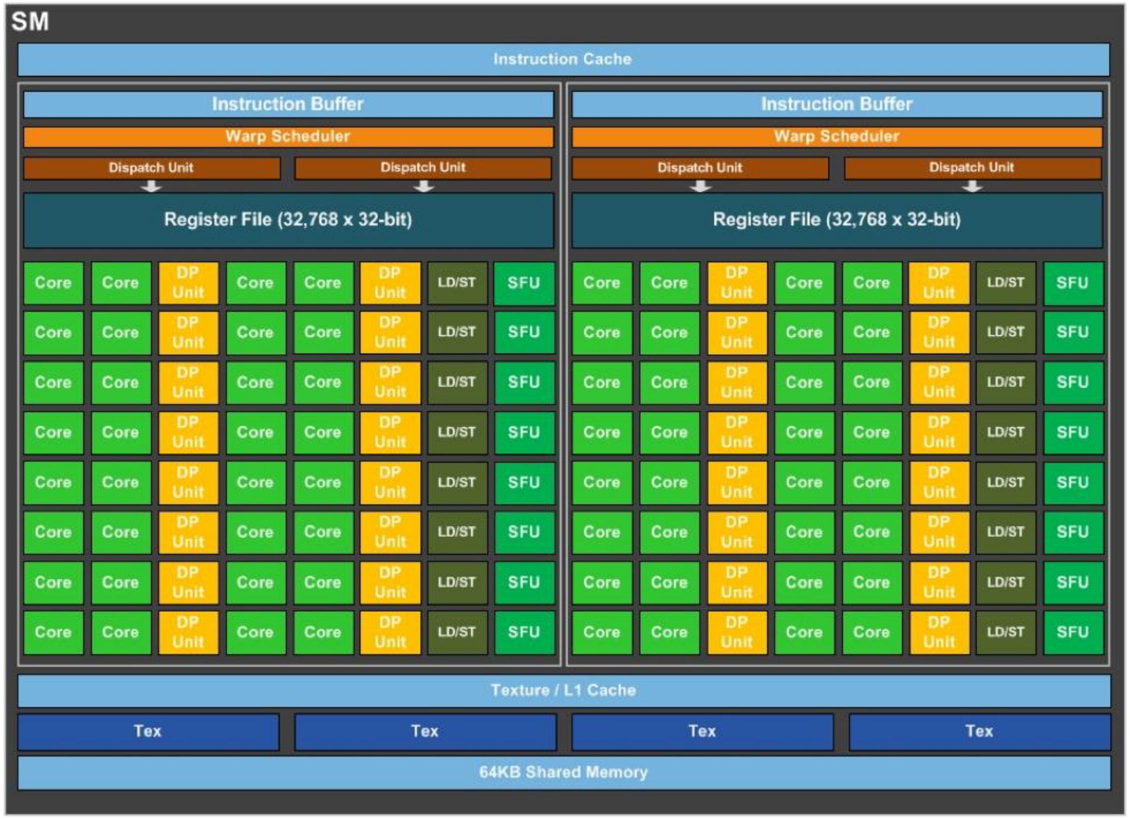
GPU: Multithreaded Multicore Chip

- Example: Nvidia Pascal GP100 (2016)



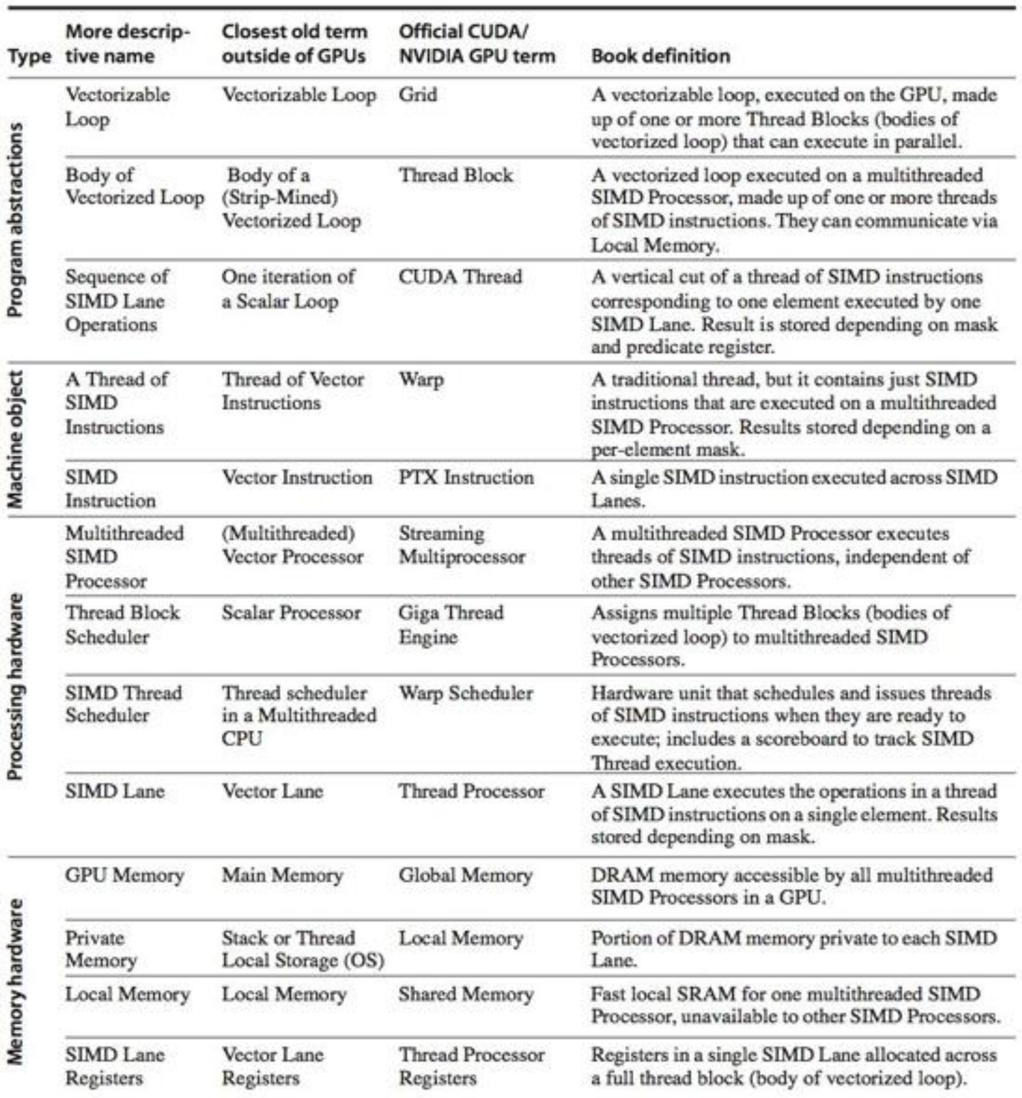
- 60 streaming multiprocessors (SMs)
- 4MB Shared L2 cache
- 8 memory controllers
  - 720 GB/s (HBM2)
- Fixed-function logic for graphics (texture units, raster ops, …)
- Scalability → change number of cores and memory channels
- Scheduling mostly controlled by hardware

Pascal Streaming Multiprocessor (SM)



- Execution units
  - 64 FUs (int and FP)
  - 16 load-store FUs
  - 16 special FUs (e.g., sqrt, sin, cos, …)
- Memory structures
  - 64K 32-bit registers
  - 64KB shared memory
- Contexts
  - 2048 threads
  - 32 blocks

Vector vs GPU Terminology



Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

[H&P5, Fig 4.25]

Thank you!

Next Lecture:
Transactional Memory