Stanford CS 193G

Lecture 15:
Optimizing Parallel GPU Performance

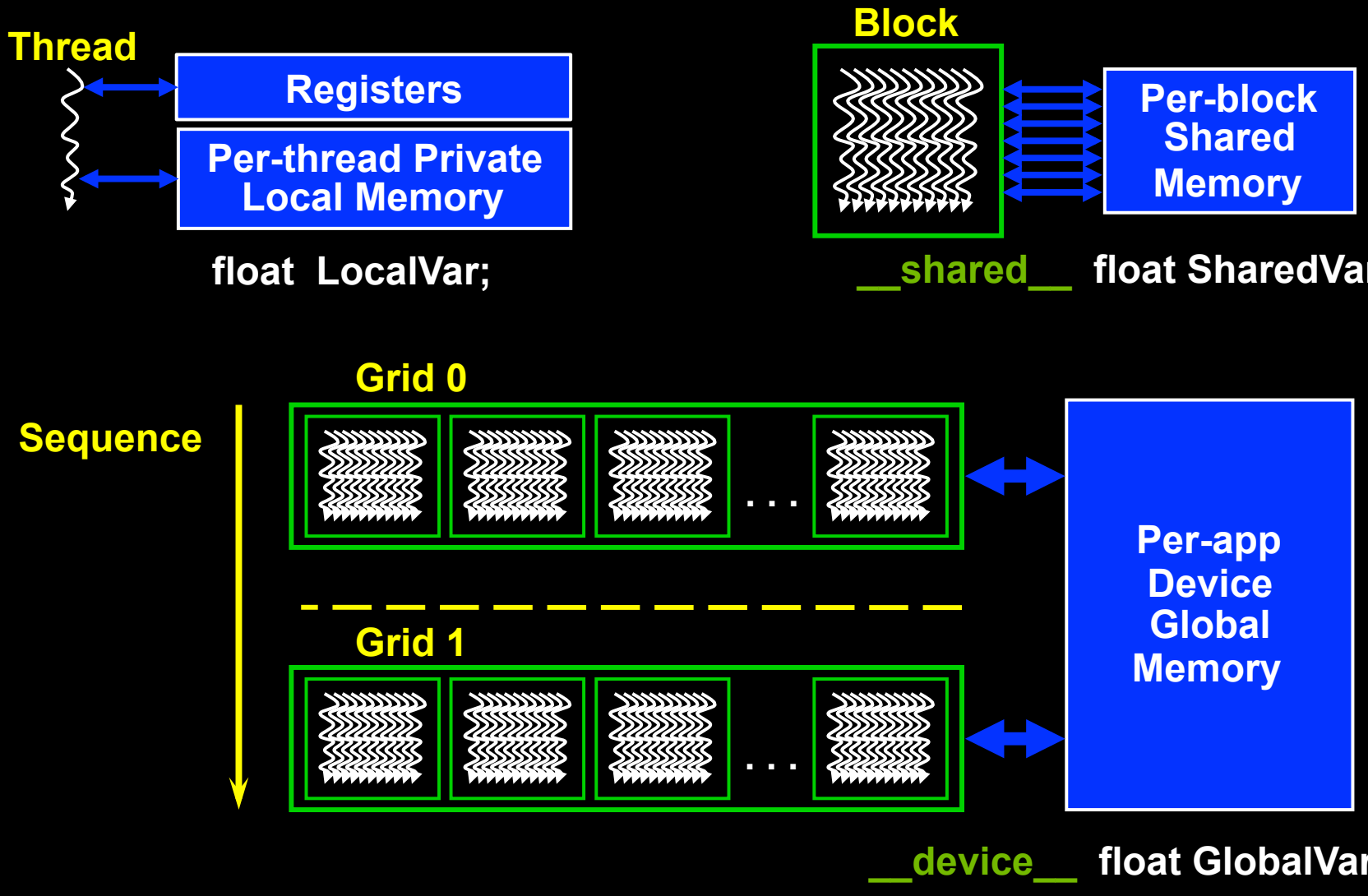
2010-05-20

John Nickolls

Optimizing parallel performance

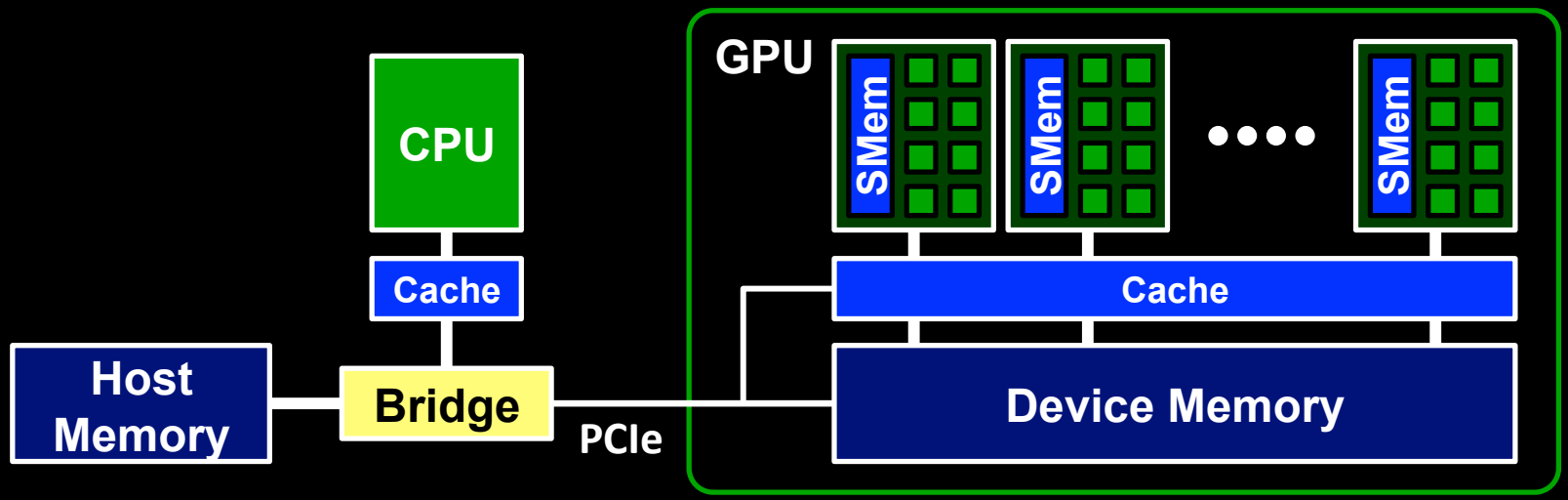
- Understand how software maps to architecture
- Use heterogeneous CPU+GPU computing
- Use massive amounts of parallelism
- Understand SIMT instruction execution
- Enable global memory coalescing
- Understand cache behavior
- Use Shared memory
- Optimize memory copies
- Understand PTX instructions

CUDA Parallel Threads and Memory



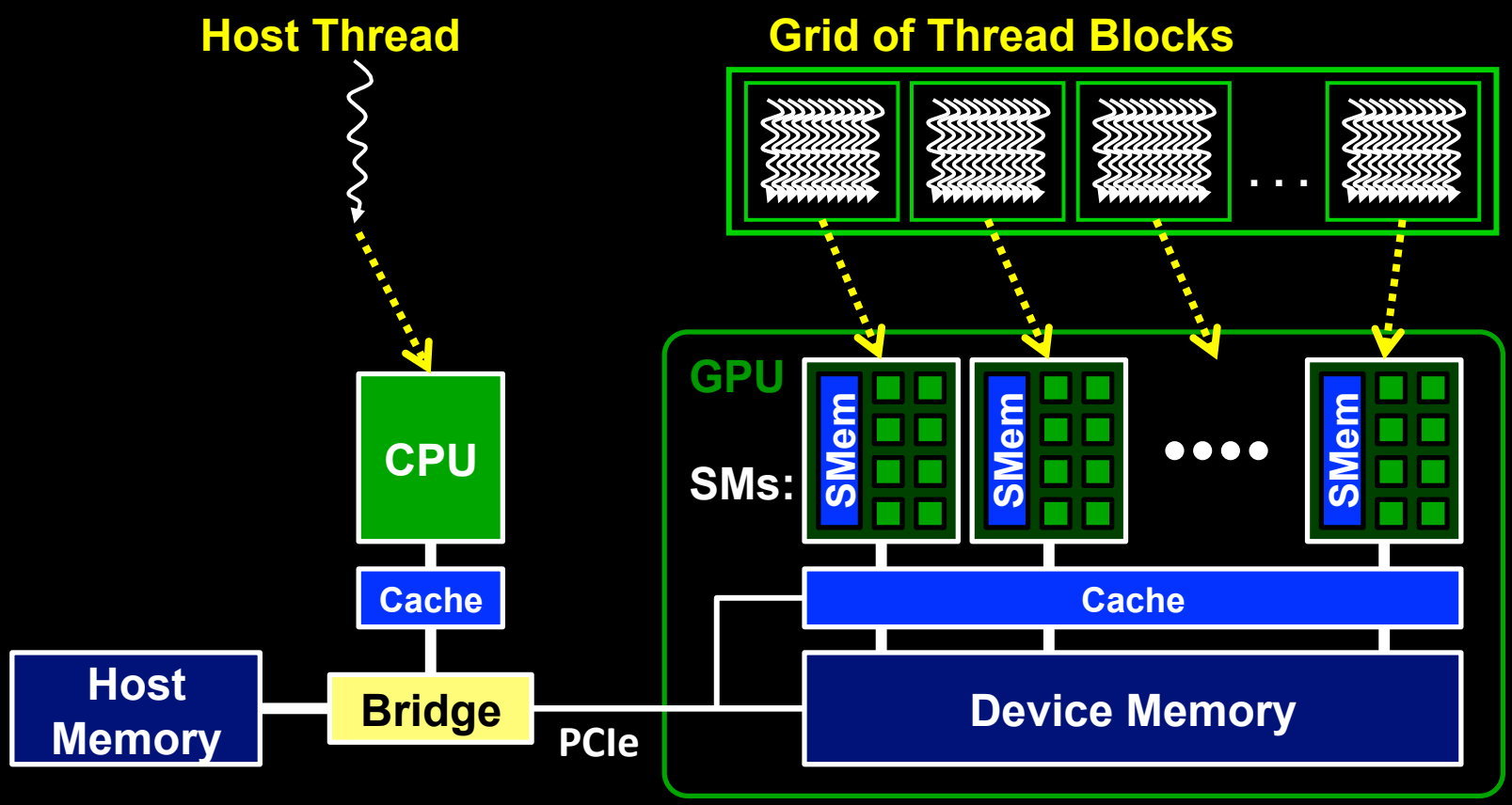
Using CPU+GPU Architecture

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

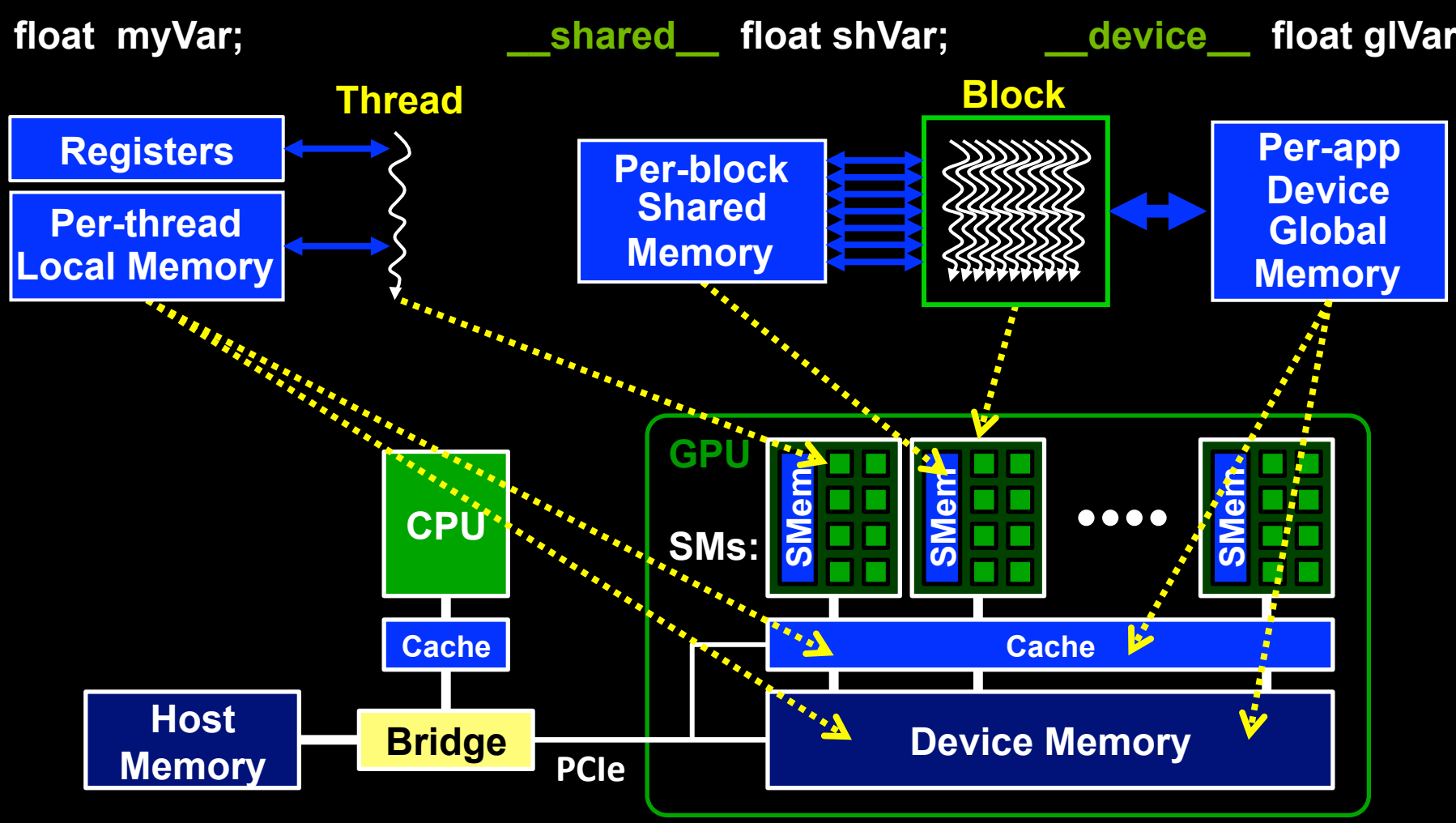


CUDA kernel maps to Grid of Blocks

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

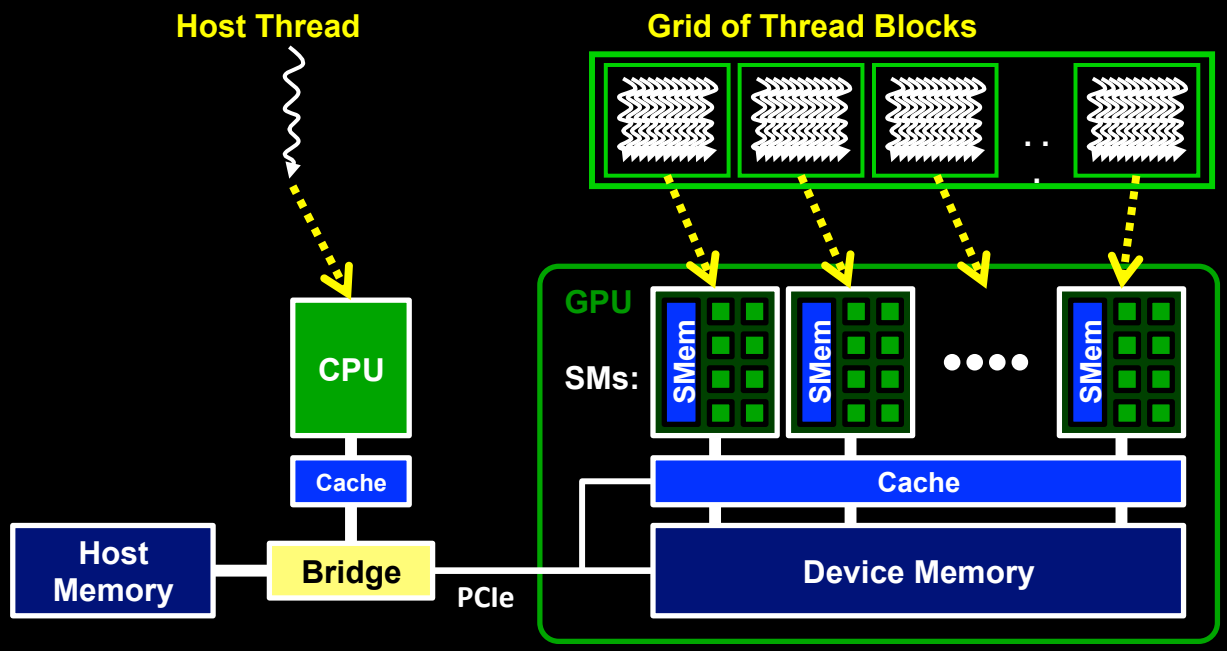


Thread blocks execute on an SM
Thread instructions execute on a core



CUDA Parallelism

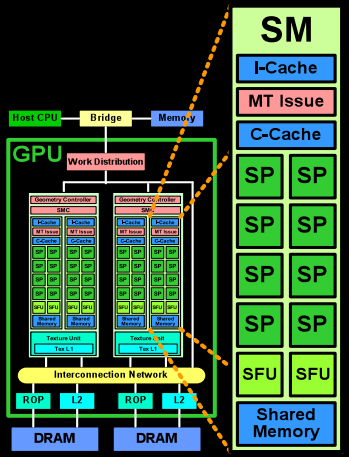
- CUDA virtualizes the physical hardware
  - Thread is a virtualized scalar processor (registers, PC, state)
  - Block is a virtualized multiprocessor (threads, shared mem.)
- Scheduled onto physical hardware without pre-emption
  - Threads/blocks launch & run to completion
  - Blocks execute independently

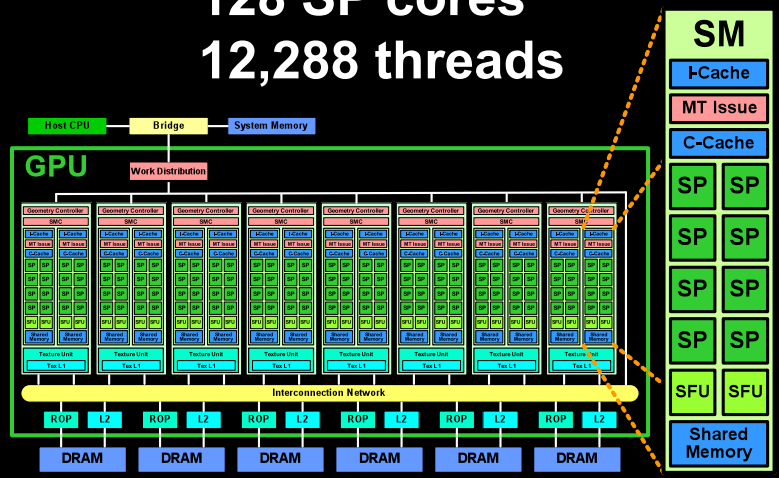


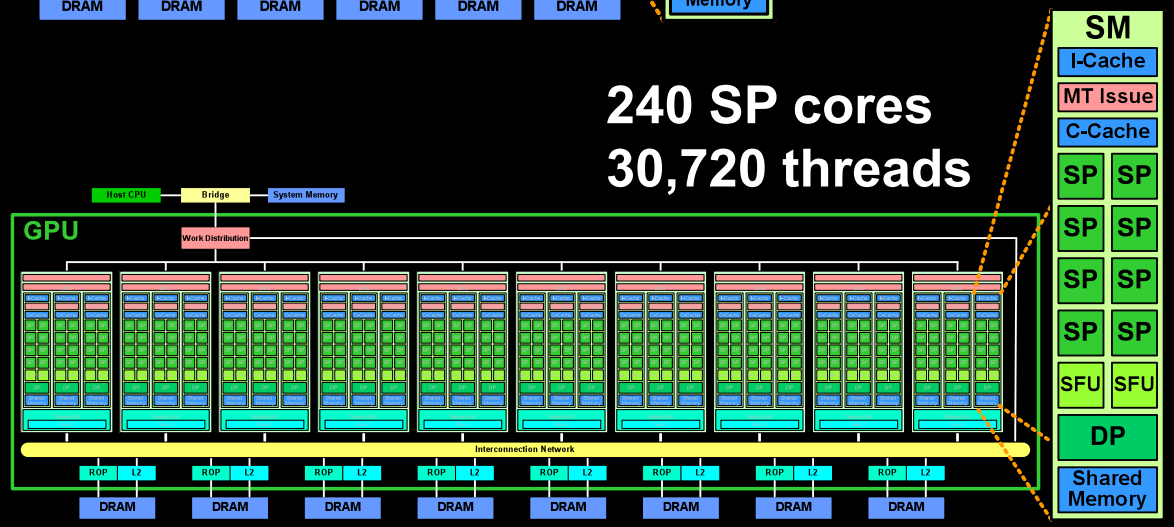
Expose Massive Parallelism

- Use hundreds to thousands of thread blocks
  - A thread block executes on one SM
  - Need many blocks to use 10s of SMs
  - SM executes 2 to 8 concurrent blocks efficiently
  - Need many blocks to scale to different GPUs
  - Coarse-grained data parallelism, task parallelism
- Use hundreds of threads per thread block
  - A thread instruction executes on one core
  - Need 384 – 512 threads/SM to use all the cores all the time
  - Use multiple of 32 threads (warp) per thread block
  - Fine-grained data parallelism, vector parallelism, thread parallelism, instruction-level parallelism

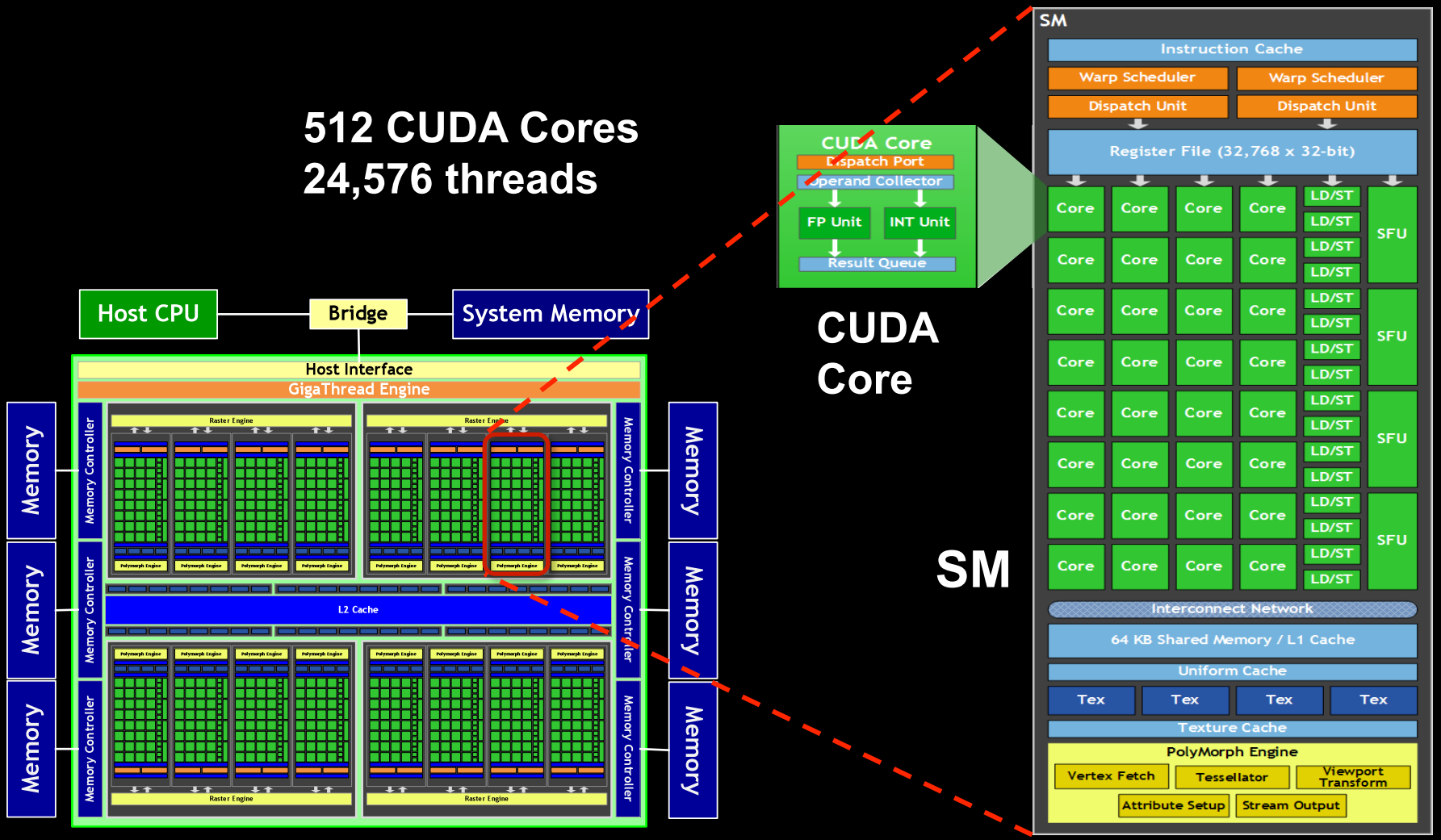
Scalable Parallel Architectures run thousands of concurrent threads





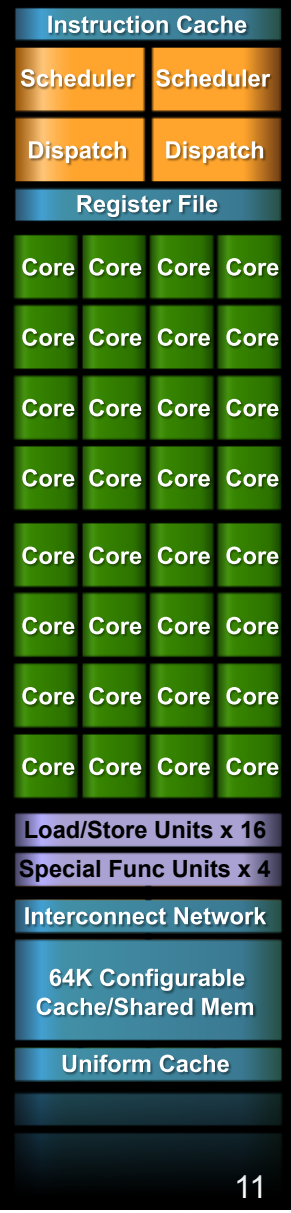


Fermi SM increases instruction-level parallelism

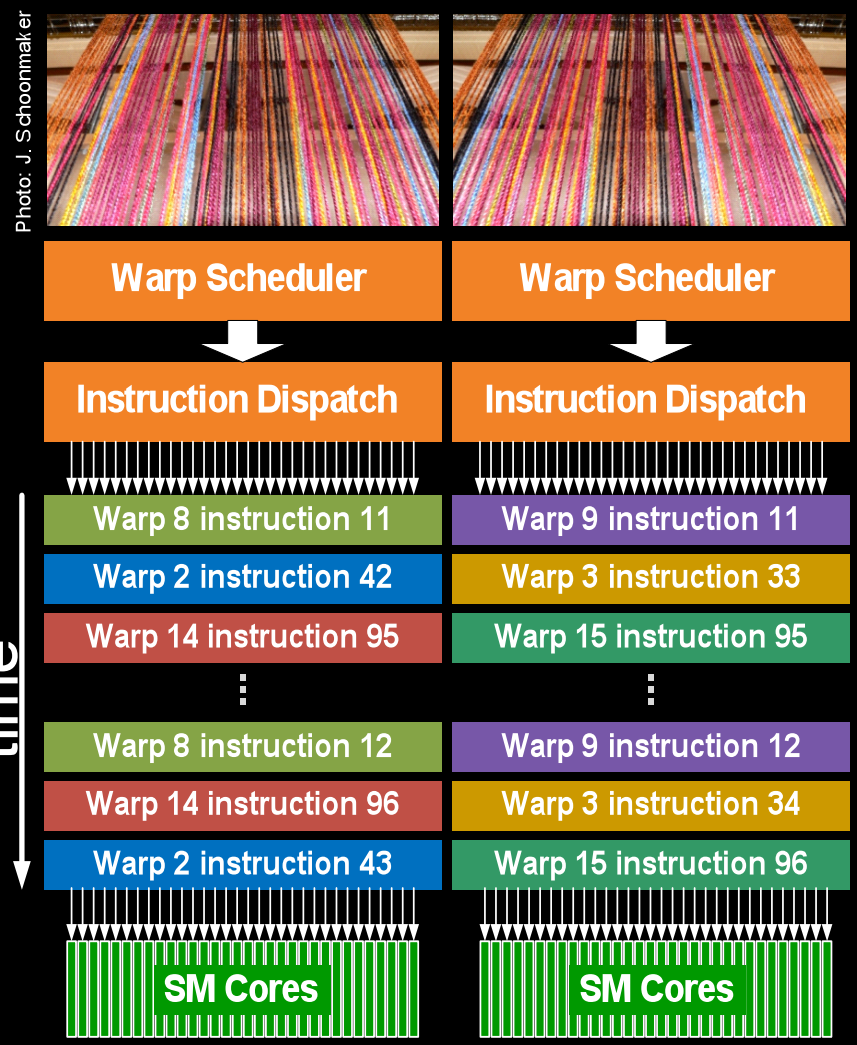


SM parallel instruction execution

- **SIMT** (Single Instruction Multiple Thread) **execution**
  - Threads run in groups of 32 called **warps**
  - Threads in a warp share instruction unit (IU)
  - HW automatically handles branch divergence
- **Hardware multithreading**
  - HW resource allocation & thread scheduling
  - HW relies on threads to hide latency
- **Threads have all resources needed to run**
  - Any warp not waiting for something can run
  - Warp context switches are zero overhead



SIMT Warp Execution in the SM



Warp: a set of 32 parallel threads that execute an instruction together

SIMT: Single-Instruction Multi-Thread applies instruction to warp of independent parallel threads

- SM dual issue pipelines select two warps to issue to parallel cores
- SIMT warp executes each instruction for 32 threads
- Predicates enable/disable individual thread execution
- Stack manages per-thread branching
- Redundant regular computation faster than irregular branching

Enable Global Memory Coalescing

- Individual threads access independent addresses
- A thread loads/stores 1, 2, 4, 8, 16 B per access
- LD.sz / ST.sz; sz = {8, 16, 32, 64, 128} bits per thread

- For 32 parallel threads in a warp, SM load/store units coalesce individual thread accesses into minimum number of 128B cache line accesses or 32B memory block accesses
- Access serializes to distinct cache lines or memory blocks

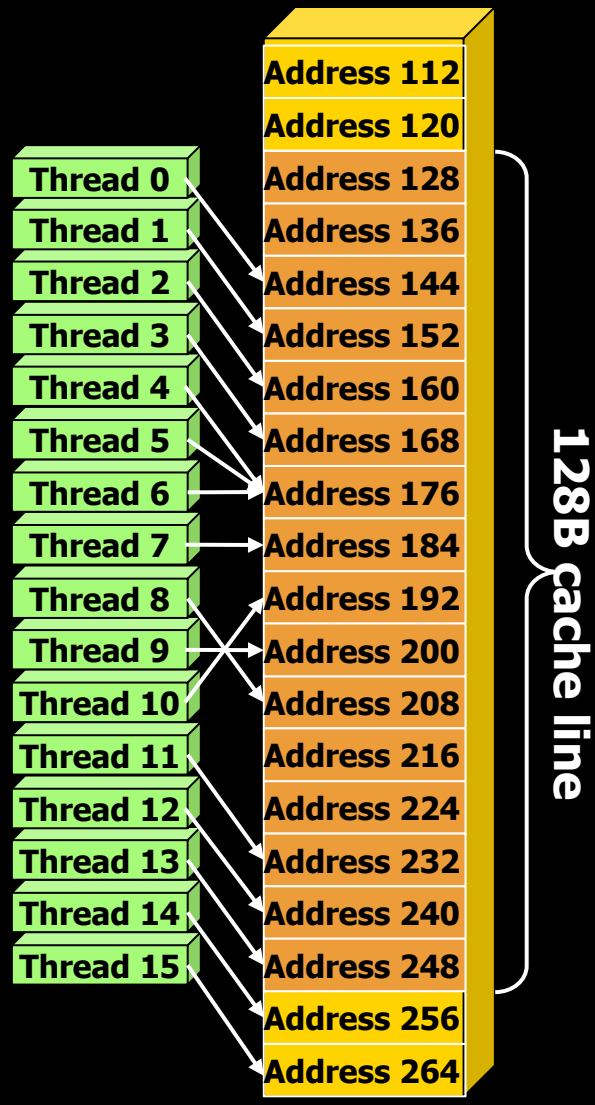
- Use nearby addresses for threads in a warp
- Use unit stride accesses when possible
- Use Structure of Arrays (SoA) to get unit stride

Unit stride accesses coalesce well


```
__global__ void kernel(float* arrayIn,
                       float* arrayOut)
{
  int i = blockDim.x * blockIdx.x
        + threadIdx.x;
  // Stride 1 coalesced load access
  float val = arrayIn[i];
  // Stride 1 coalesced store access
  arrayOut[i] = val + 1;
}
```

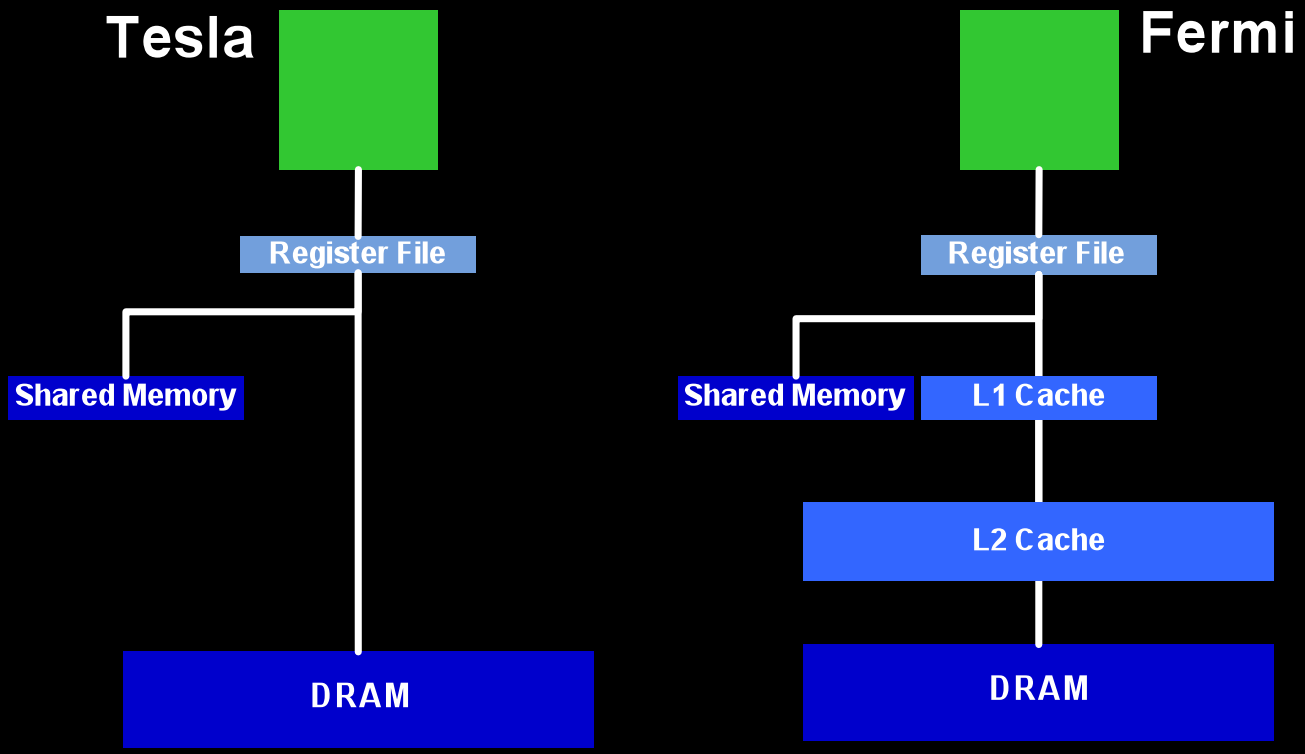
Example Coalesced Memory Access

- 16 threads within a warp load 8 B per thread:
  LD.64 Rd, [Ra + offset] ;
- 16 individual 8B thread accesses fall in two 128B cache lines
- LD.64 coalesces 16 individual accesses into 2 cache line accesses
- Implements parallel vector scatter/gather
- Loads from same address are broadcast
- Stores to same address select a winner
- Atomics to same address serialize
- Coalescing scales gracefully with the number of unique cache lines or memory blocks accessed

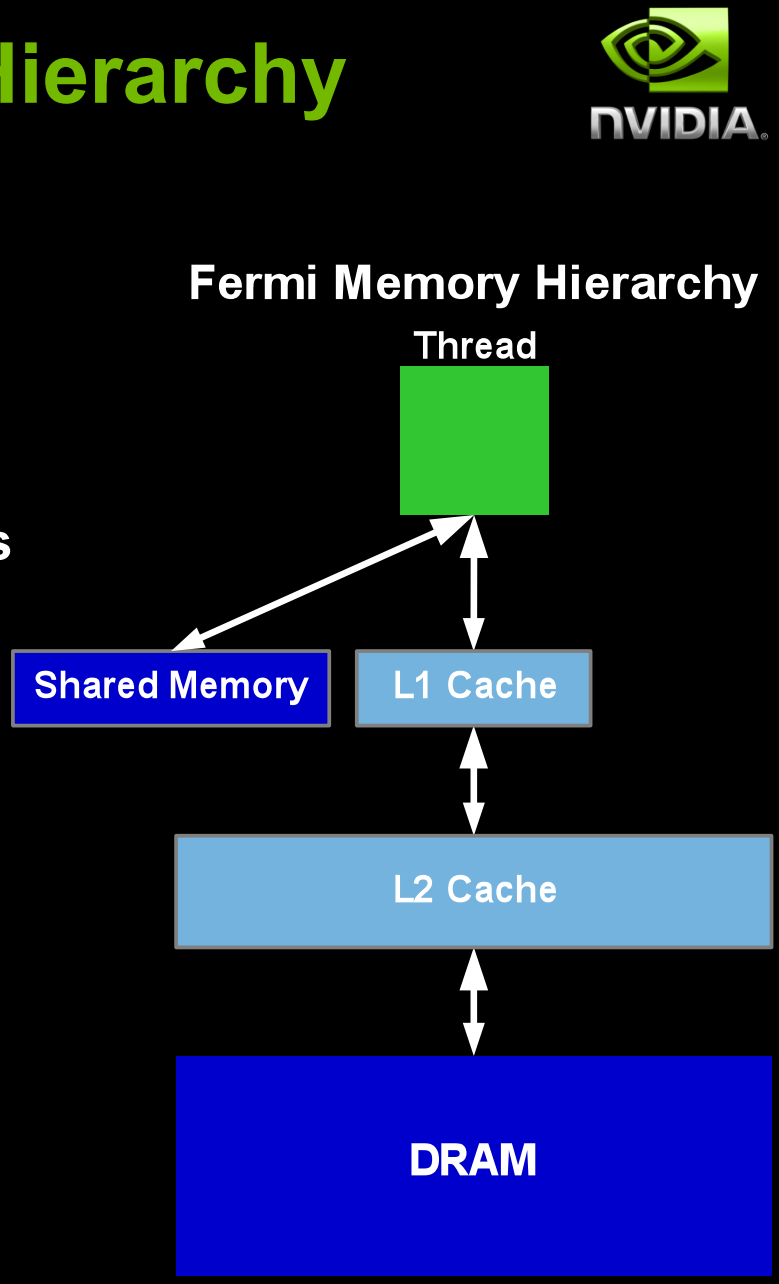


Memory Access Pipeline

- Load/store/atomic memory accesses are pipelined
- Latency to DRAM is a few hundred clocks
- Batch load requests together, then use return values
- Latency to Shared Memory / L1 Cache is 10 – 20 cycles



Fermi Cached Memory Hierarchy



- Configurable L1 cache per SM
  - 16KB L1$ / 48KB Shared Memory
  - 48KB L1$ / 16KB Shared Memory
- L1 caches per-thread local accesses
  - Register spilling, stack access
- L1 caches global LD accesses
- Global stores bypass L1

- Shared 768KB L2 cache
- L2 cache speeds atomic operations
- Caching captures locality, amplifies bandwidth, reduces latency
- Caching aids irregular or unpredictable accesses

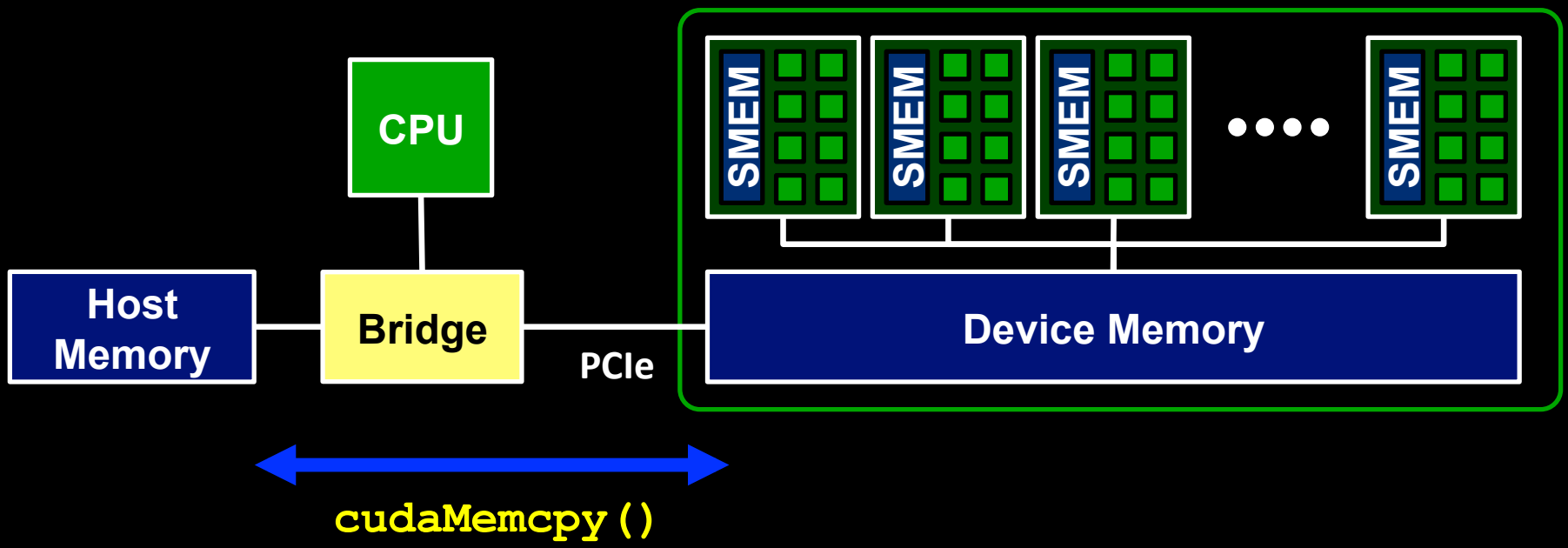
Use per-Block Shared Memory

- Latency is an order of magnitude lower than L2 or DRAM
- Bandwidth is 4x – 8x higher than L2 or DRAM

- Place data blocks or tiles in shared memory when the data is accessed multiple times
- Communicate among threads in a block using Shared memory
- Use synchronization barriers between communication steps
  - __syncthreads() is single bar.sync instruction – very fast

- Threads of warp access shared memory banks in parallel via fast crossbar network
- Bank conflicts can occur – incur a minor performance impact
- Pad 2D tiles with extra column for parallel column access if tile width == # of banks (16 or 32)

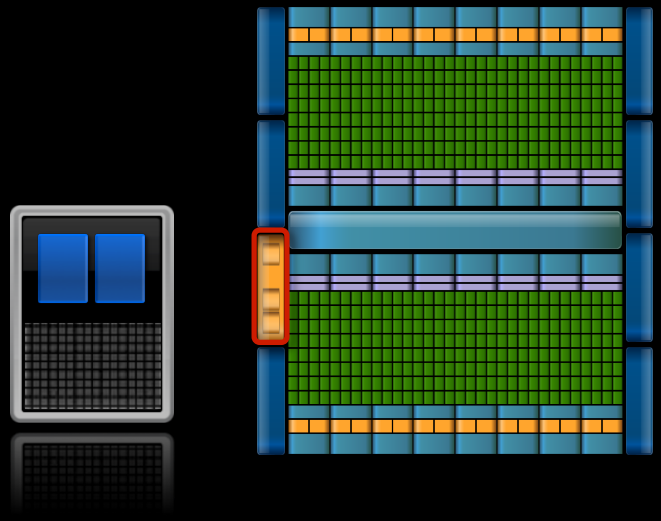
Using cudaMemCpy()



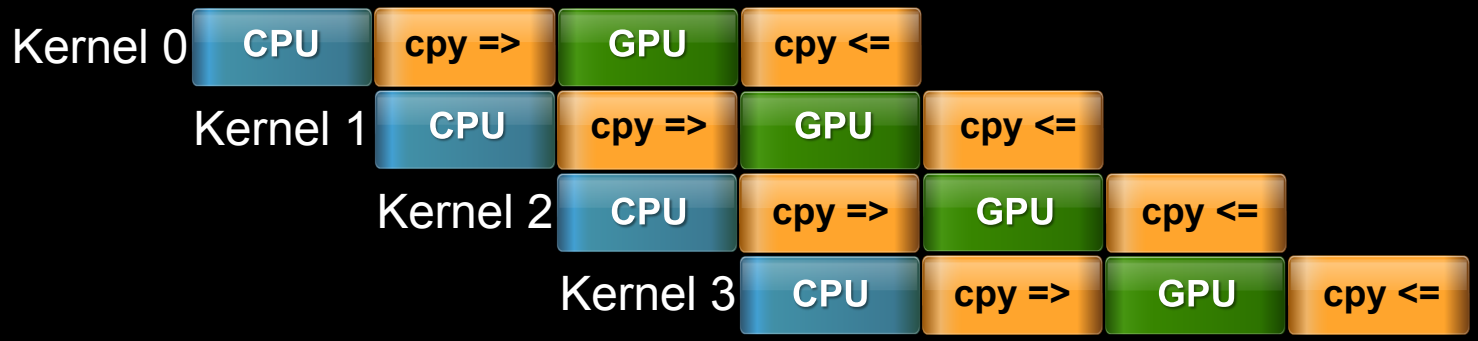
- cudaMemcpy() invokes a DMA copy engine
- Minimize the number of copies
- Use data as long as possible in a given place
- PCIe gen2 peak bandwidth = 6 GB/s
- GPU load/store DRAM peak bandwidth = 150 GB/s

Overlap computing & CPU⇔GPU transfers

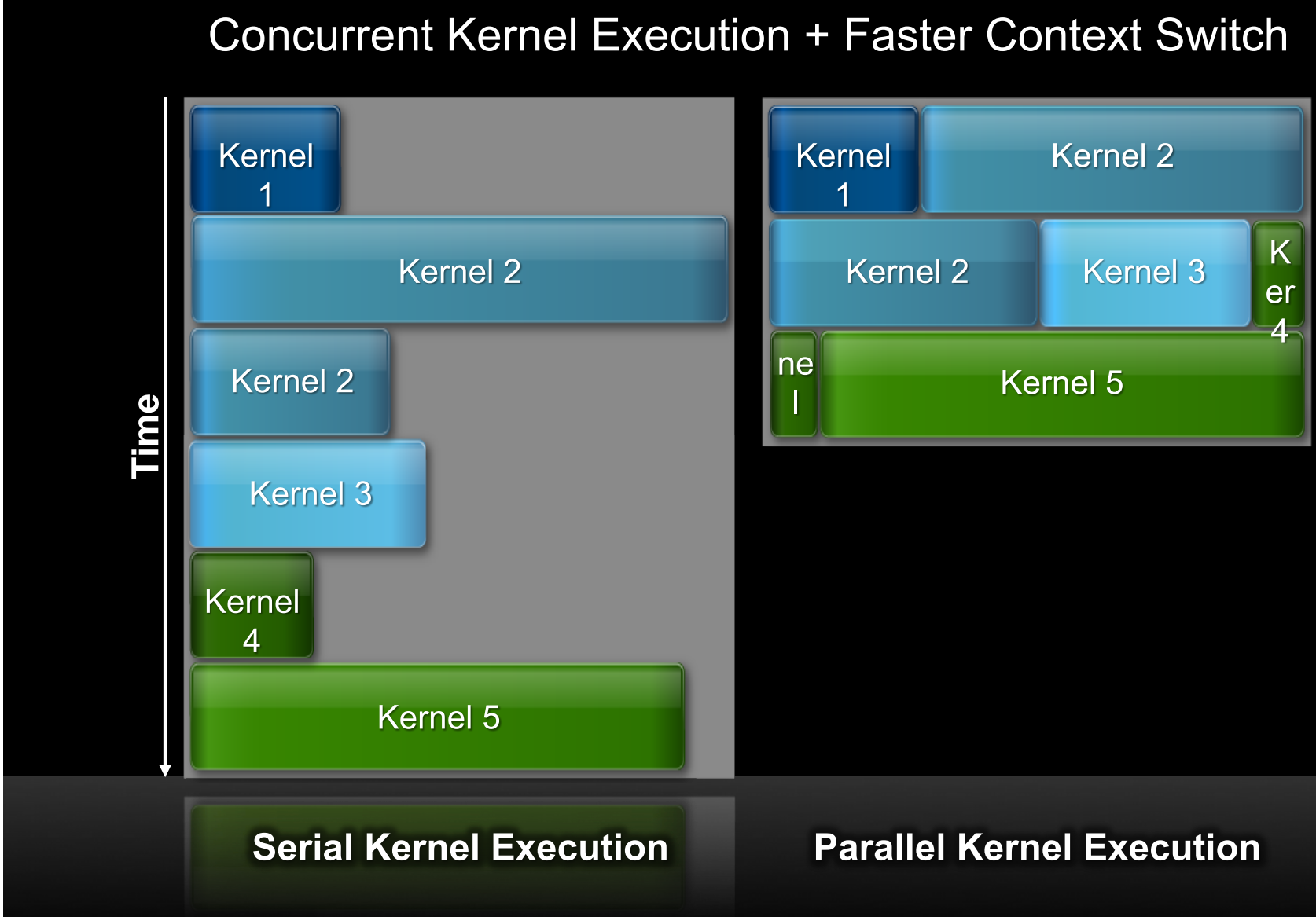
- **cudaMemcpy() invokes data transfer engines**
  - CPU→GPU and GPU→CPU data transfers
  - Overlap with CPU and GPU processing



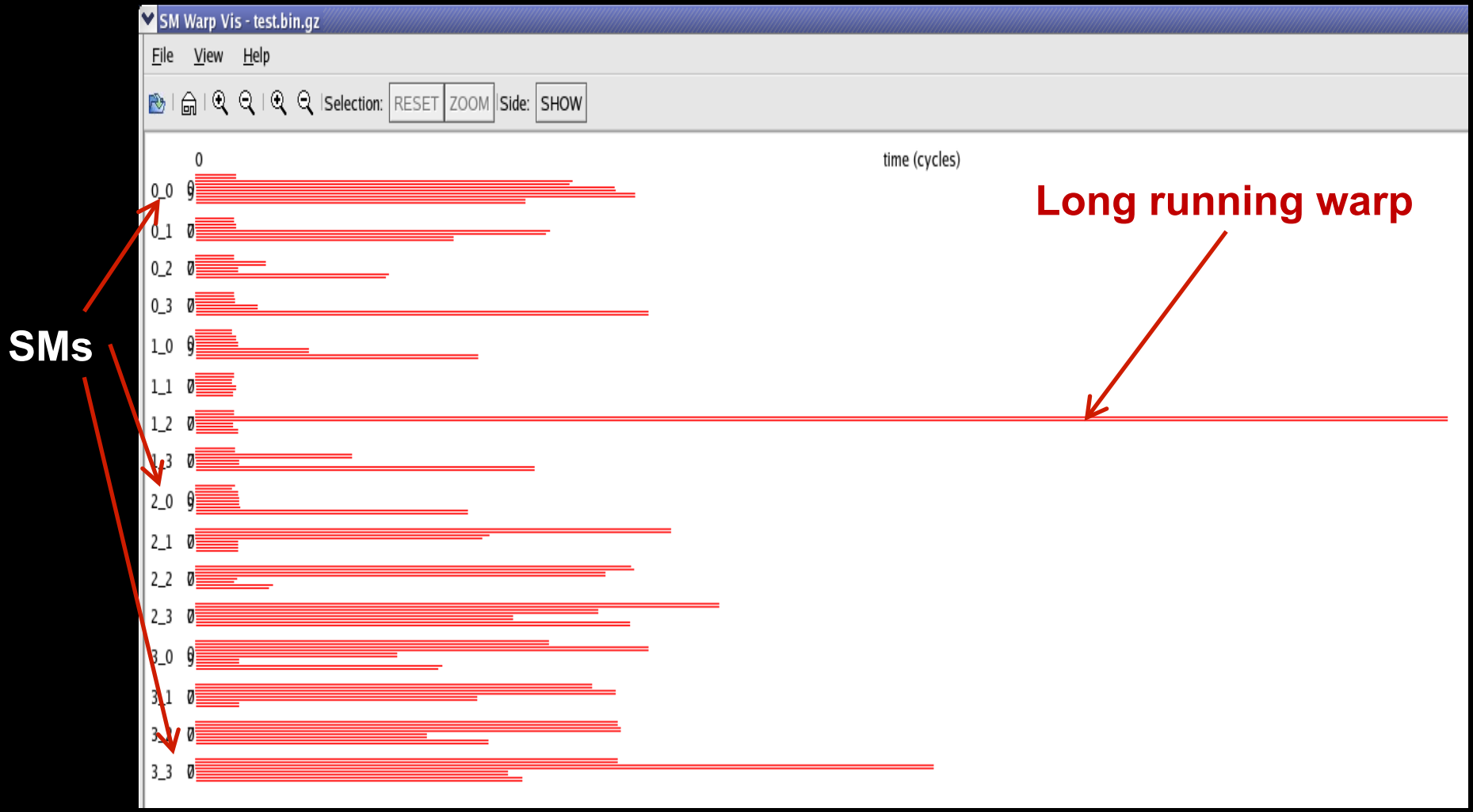
- **Pipeline Snapshot:**



Fermi runs independent kernels in parallel



Minimize thread runtime variance



Warps executing kernel with variable run time

PTX Instructions

- Generate a program.ptx file with nvcc –ptx
- http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/ptx_isa_2.0.pdf
- PTX instructions: op.*type* dest, srcA, srcB;
  - *type* = .b32, .u32, .s32, .f32, .b64, .u64, .s64, .f64
  - *memtype* = .b8, .b16, .b32, .b64, .b128
- PTX instructions map directly to Fermi instructions
  - Some map to instruction sequences, e.g. div, rem, sqrt
  - PTX virtual register operands map to SM registers
- Arithmetic: add, sub, mul, mad, fma, div, rcp, rem, abs, neg, min, max, setp.*cmp*, cvt
- Function: sqrt, sin, cos, lg2, ex2
- Logical: mov, selp, and, or, xor, not, cnot, shl, shr
- Memory: ld, st, atom.*op*, tex, suld, sust
- Control: bra, call, ret, exit, bar.sync

Optimizing Parallel GPU Performance

- Understand the parallel architecture
- Understand how application maps to architecture
- Use LOTS of parallel threads and blocks
- Often better to redundantly compute in parallel
- Access memory in local regions
- Leverage high memory bandwidth
- Keep data in GPU device memory
- Experiment and measure

- Questions?