

CSE 160

Lecture 24

Graphical Processing Units

Announcements

- Next week we meet in 1202 on Monday 3/11 only
- On Weds 3/13 we have a 2 hour session
 - Usual class time at the Rady school – final exam review
 - SDSC tour @ 2.15pm (replaces class on 3/15)
- Quiz return

Quiz

1. $T(n) = \alpha + \beta^{-1} n$, α term dominates for short messages, β term for long messages
2. Had to show that the only outcome was $s, t = \text{ONE}, \text{TWO}$
3. Two possible answers
 1. Use collective MPI_Reduce
 2. Use wildcard sender to process local sums in the order they arrive rather than a fixed order
 1. Many IRecvs not scalable, requires synchronization

Today's lecture

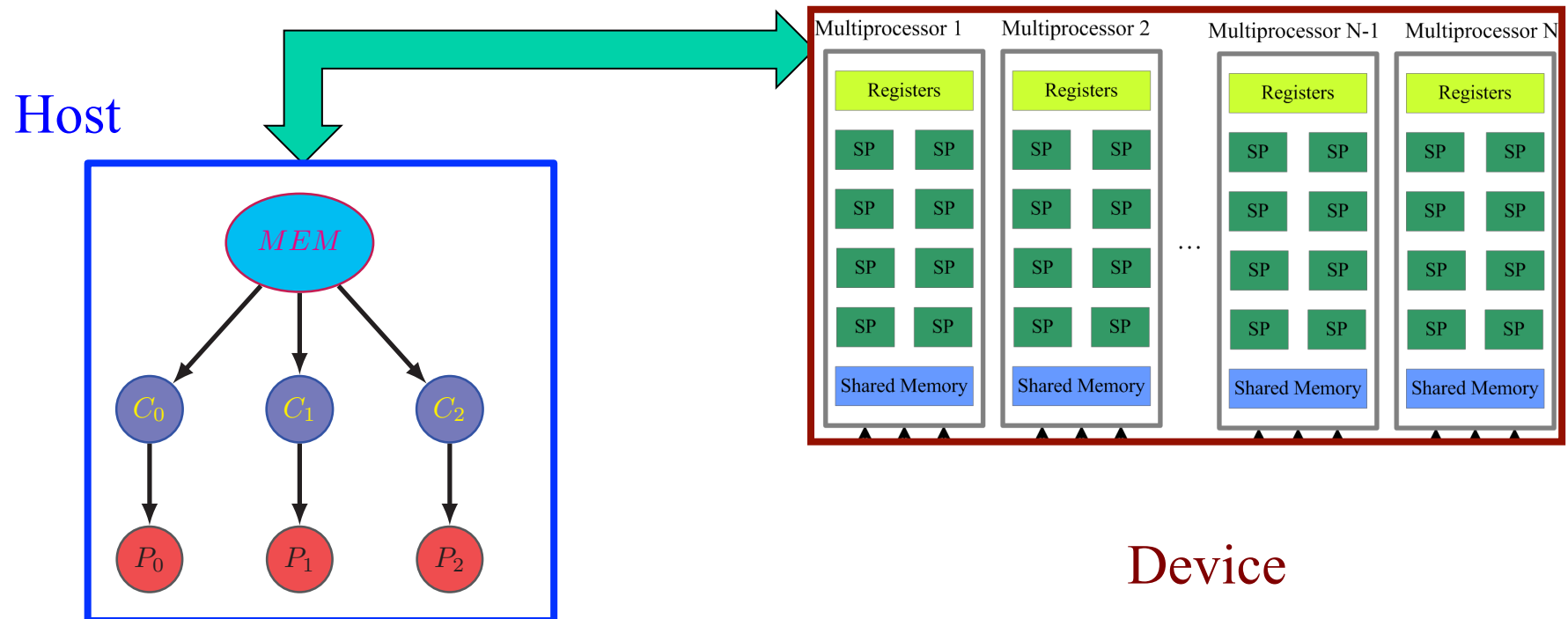
- GPU Architecture
- Programming with CUDA

Processor design trends

- No longer possible to use a growing population of transistors to boost single processor performance
 - Can no longer increase the clock speed
- Instead, we replicate the cores
 - Simplified design, reduces power, pack more onto the chip
- Simplified core
 - Remove architectural enhancements like branch caches
 - Constrain memory access and control flow
 - Partially expose the memory hierarchy

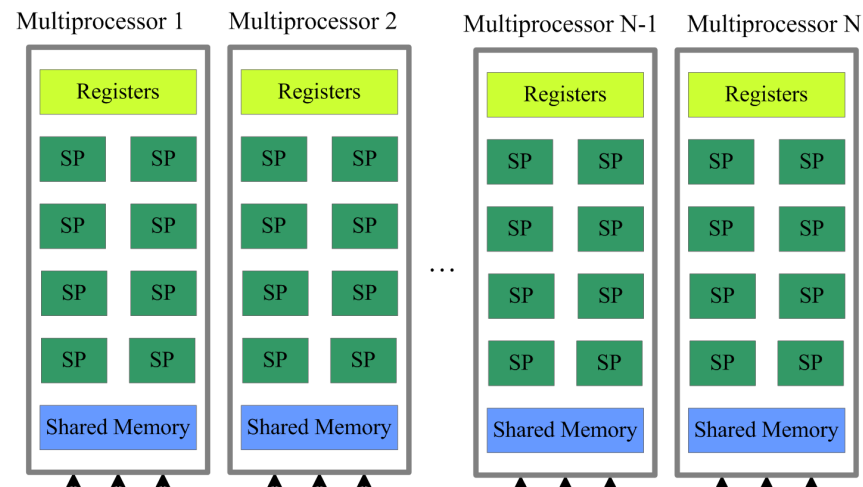
Heterogeneous processing with Graphical Processing Units

- Specialized *many-core* processor
- Explicit data motion
 - between *host* and *device*
 - inside the device



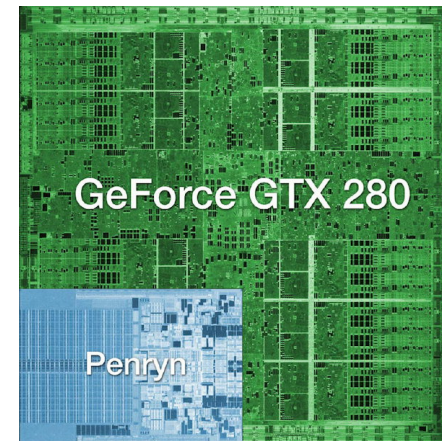
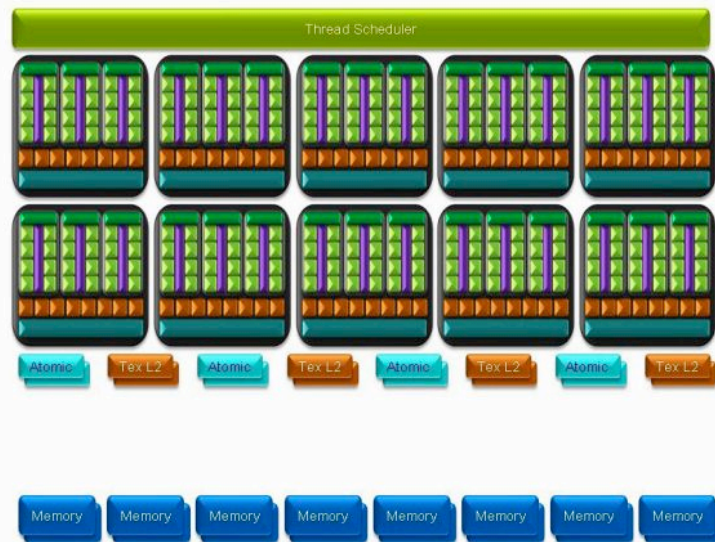
Graphical Processing Units

- Processes long vectors
- Thousand of highly specialized cores
- NVIDIA, AMD



NVIDIA GeForce GTX 280

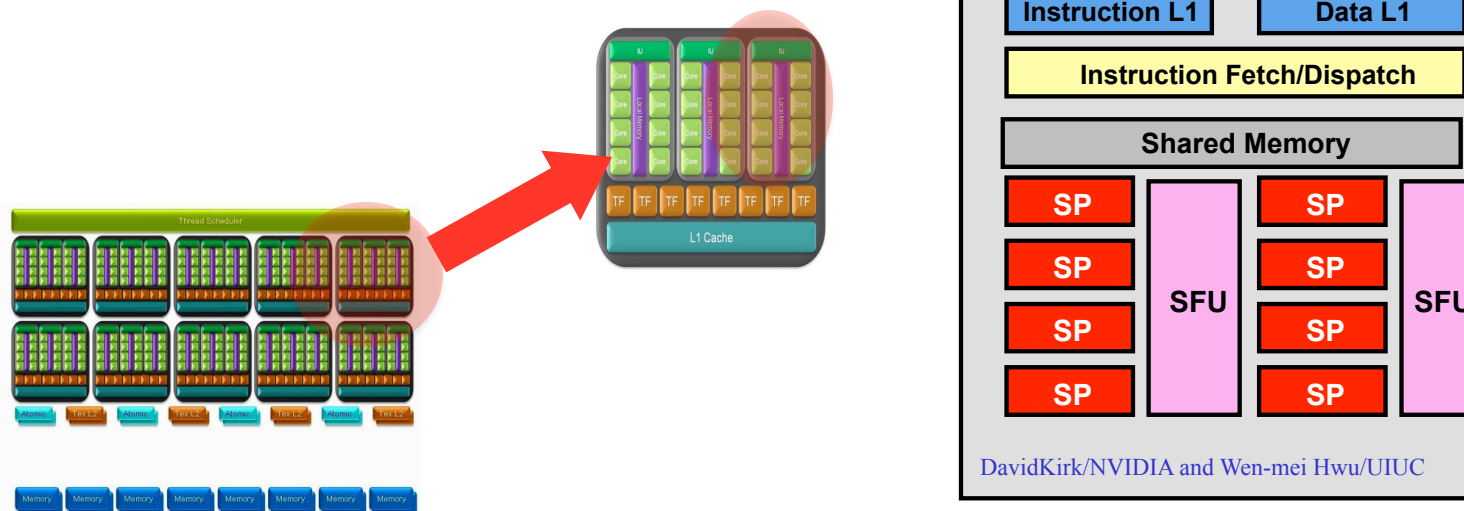
- Hierarchically organized clusters of streaming multiprocessors
 - 240 cores @ 1.296 GHz
 - Peak performance 933.12 Gflops/s
- 1 GB “device” memory (frame buffer)
- 512 bit memory interface @ 132 GB/s



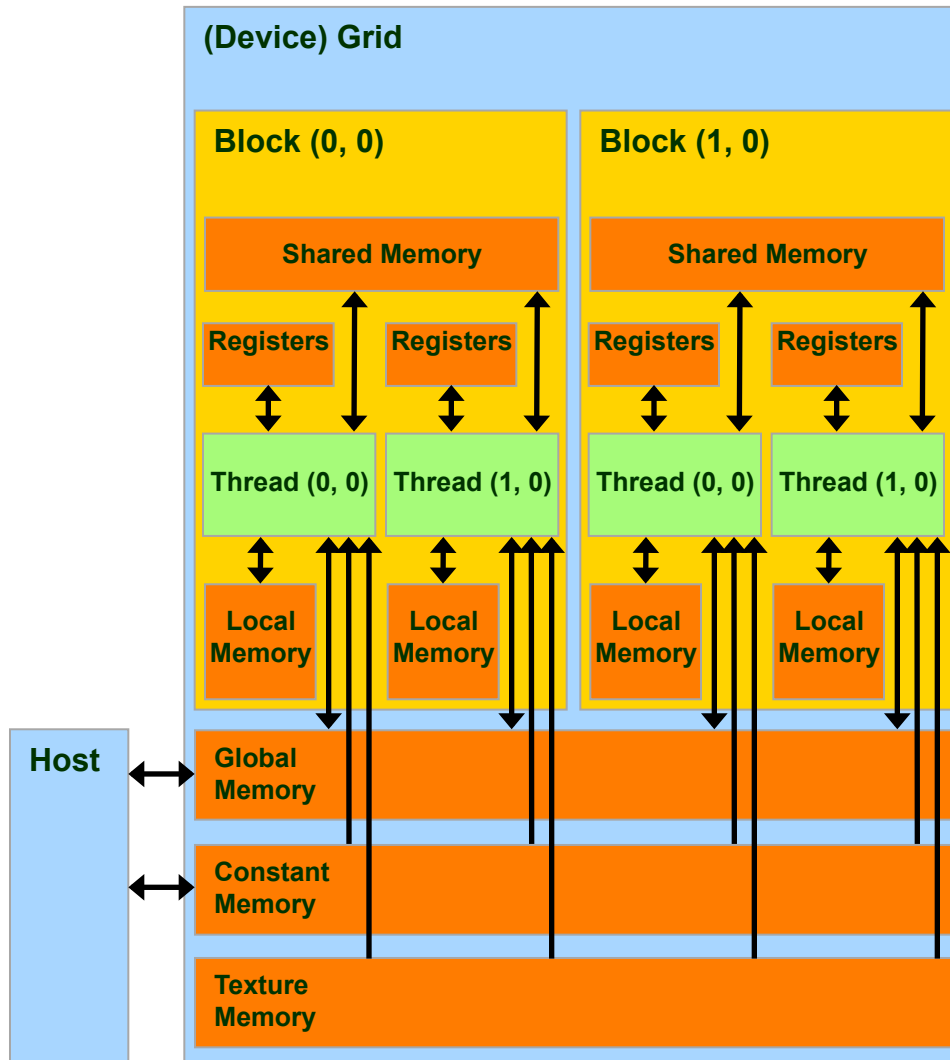
GTX 280:	1.4B transistors
Intel Penryn: (dual core)	410M (110mm ²)
Nehalem:	731M (263mm ²)

Streaming processor cluster

- GTX-280 GPU
 - 10 clusters @ 3 streaming multiprocessors or *vector cores*
- Each vector core
 - 8 scalar cores: fused multiply adder + multiplier (32 bits), truncate intermediate result
 - Shared memory (16KB) and registers (16K × 32 bits = 64KB)
 - 1 64-bit fused multiply-adder + 2 super function units (2 fused multiply-adders)
 - 1 FMA + 1 multiply per cycle = 3 flops / cycle / core * 240 cores = 720 flops/cycle @ 1.296 Ghz: 933 GFLOPS

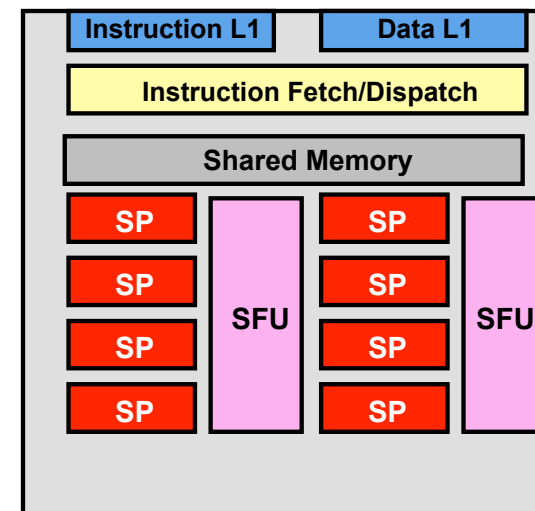


Memory Hierarchy



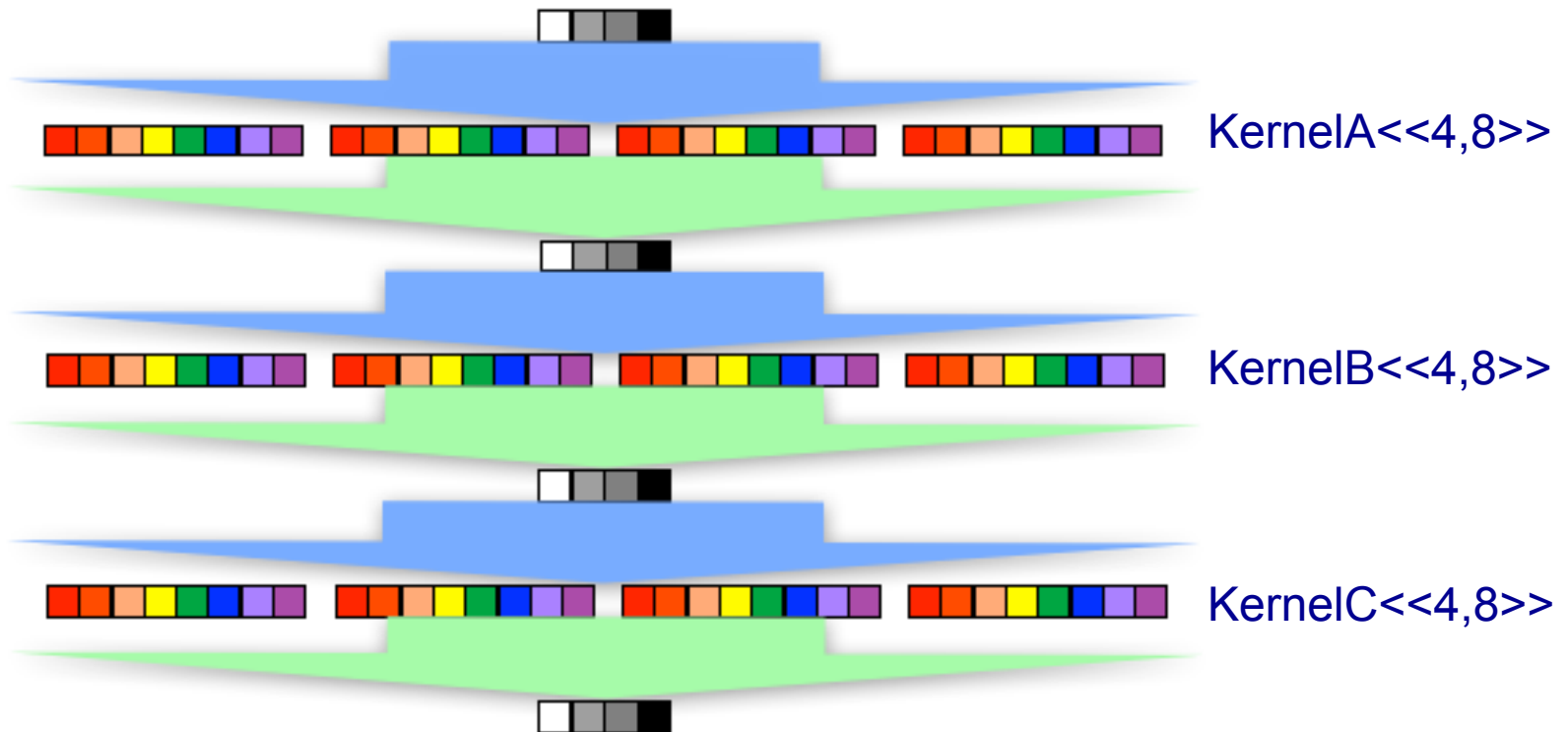
Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

Name	Latency (cycles)	Cached
Global	DRAM – 100s	No
Local	DRAM – 100s	No
Constant	1s – 10s – 100s	Yes
Texture	1s – 10s – 100s	Yes
Shared	1	--
Register	1	--



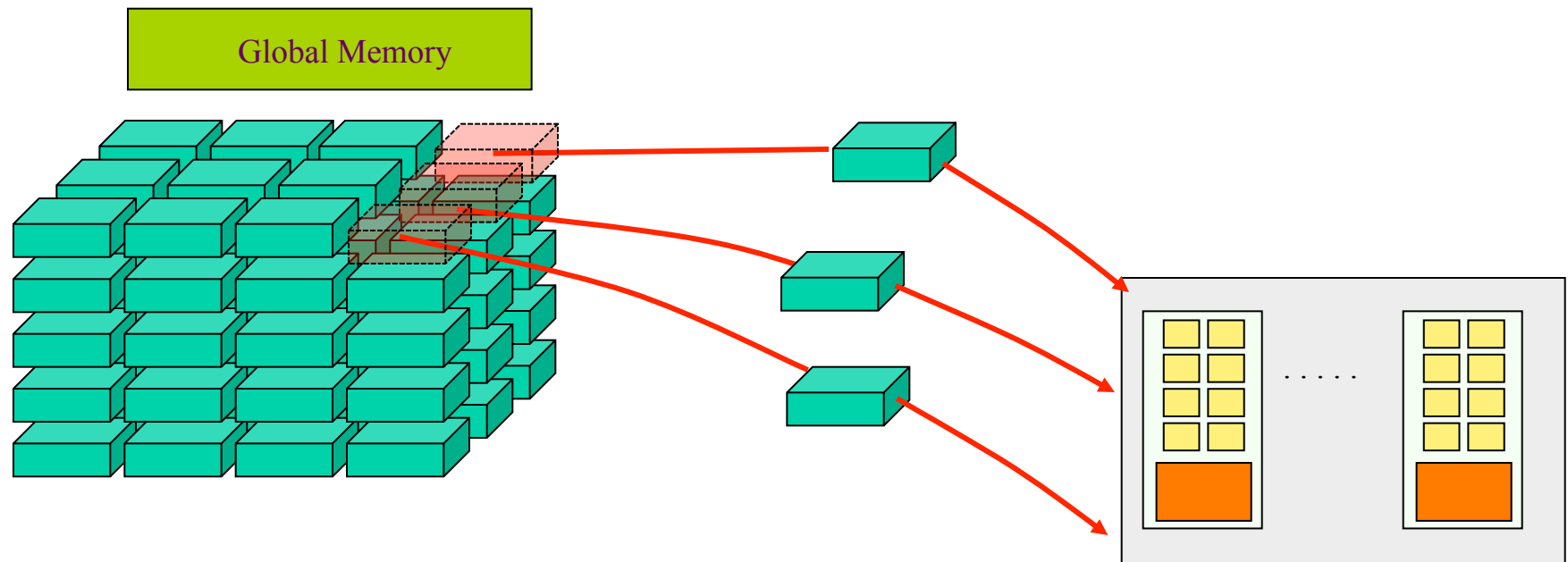
CUDA

- Programming environment with extensions to C
- Under control of the *host*, invoke sequences of multithreaded *kernels* on the *device* (GPU)
- Many lightweight *virtualized* threads
- CUDA: programming environment + C extensions



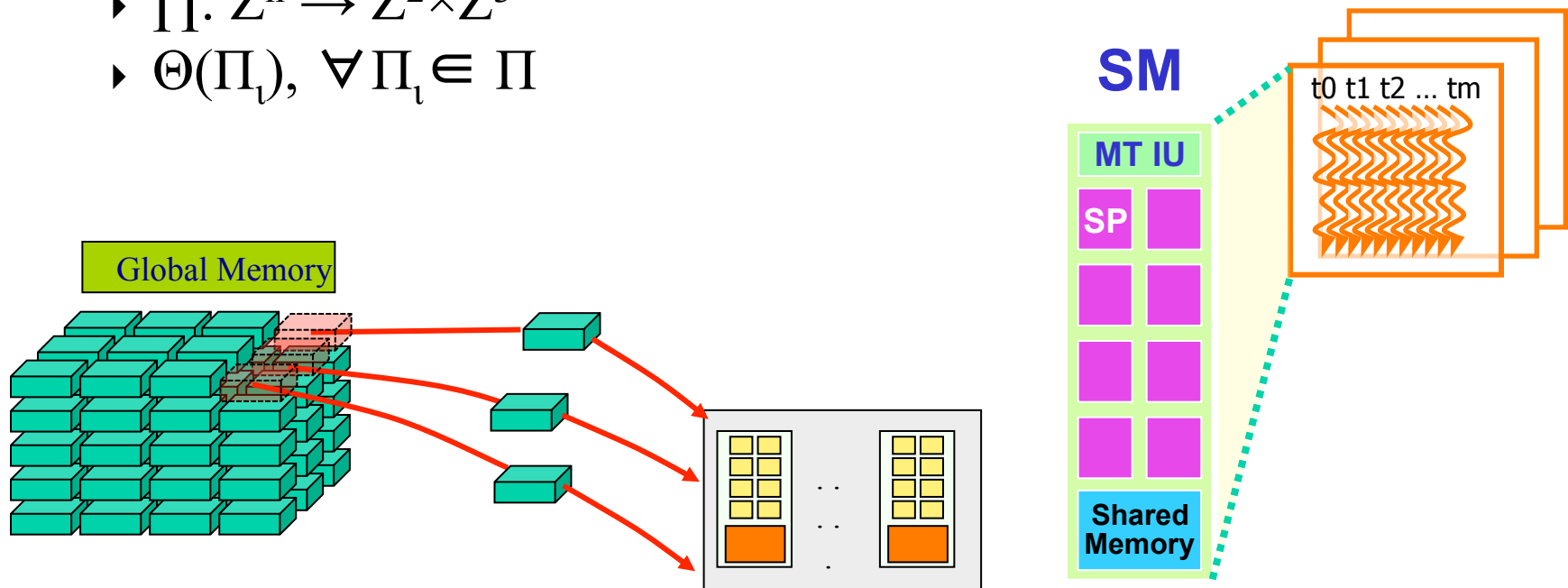
Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads
- Hardware handles dispatching, 0 overhead
- Compiler re-arranges loads to hide latencies
- Global synchronization: kernel invocation



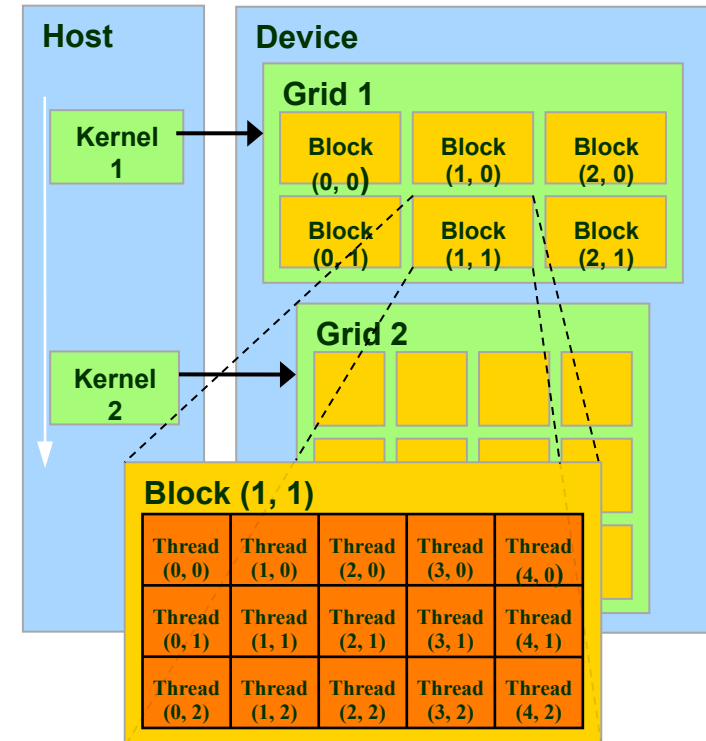
Threads organization

- Threads all execute same instruction (SIMT)
- Threads are organized into *blocks*, blocks into *grids*
- Each thread uniquely specified by block & thread ID
- Programmer determines the mapping of virtual thread IDs to global memory location
 - $\Pi: Z^n \rightarrow Z^2 \times Z^3$
 - $\Theta(\Pi_i), \forall \Pi_i \in \Pi$



Hierarchical Thread Organization

- Thread organization
 - Grid $\supset$ Block $\supset$ Thread
 - Specify number and geometry of threads in a block and similarly for blocks
- Thread Blocks
 - Unit of workload assignment
 - Subdivide a global index domain
 - Cooperate, synchronize, with access fast on-chip *shared memory*
 - Threads in different blocks communicate only through slow global memory

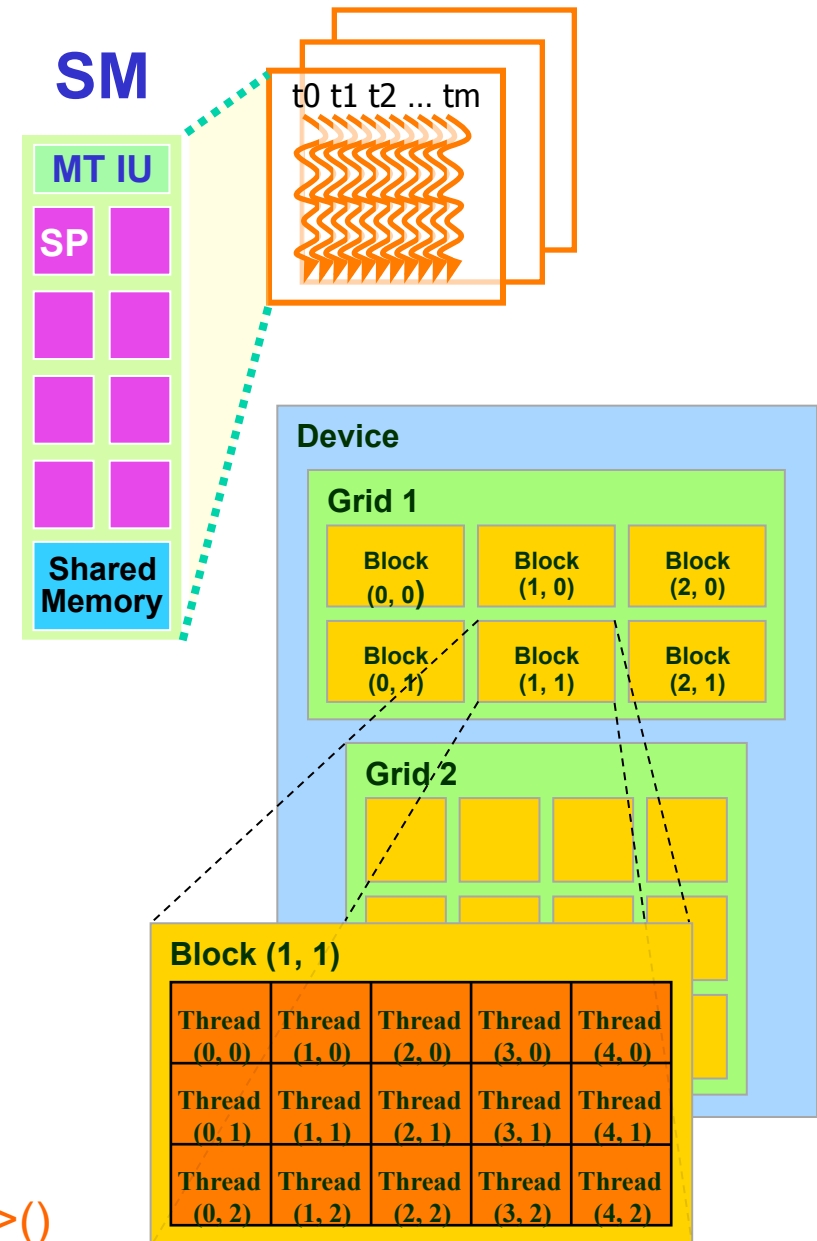


KernelA<<<2,3>,<3,5>>>()
Grid Block

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Thread execution

- Thread Blocks
 - ▶ Unit of workload assignment
 - ▶ Each thread has its own set of registers
 - ▶ All have access to a fast on-chip *shared memory*
 - ▶ Synchronization only among all threads in a block
 - ▶ Threads in different blocks communicate via slow global memory
- SIMT parallelism: all threads in a warp execute the same instruction
 - ▶ All branches followed
 - ▶ Instructions disabled
 - ▶ Divergence, serialization



KernelA<<<2,3>,<3,5>>>()
Grid Block

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Parallel Speedup

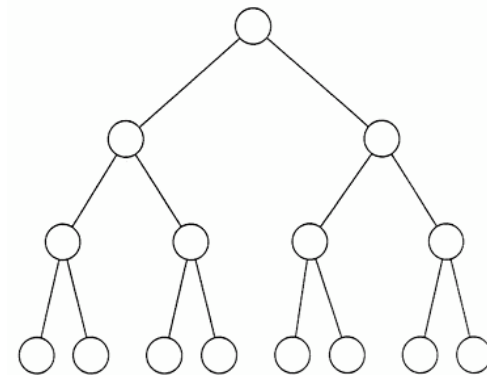
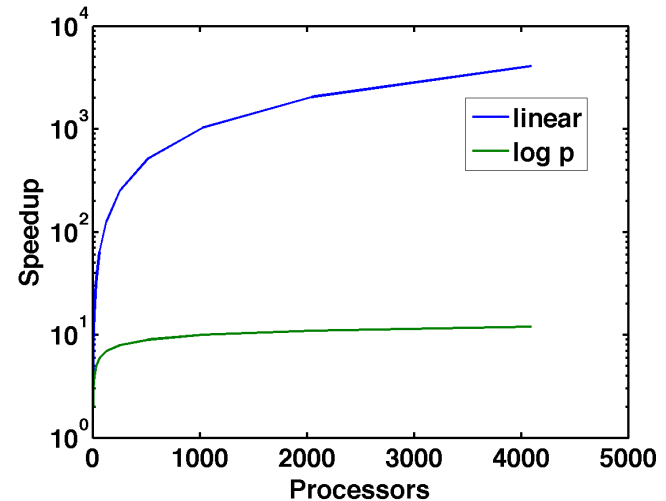
- How much did our GPU implementation improve over the traditional processor?
- *Speedup, S*

$$\frac{\text{Running time of the fastest program on conventional processors}}{\text{Running time of the accelerated program}}$$

- Baseline: a multithreaded program

How to maximize performance?

- Avoid algorithms that present intrinsic barriers to utilizing the hardware
- Hide latency of host $\leftrightarrow$ device memory transfers
- Reduce global memory accesses
 - Global memory accesses $\rightarrow$ fast on-chip accesses
 - Coalesced memory transfers
- Avoid costly branches, or render harmless
- Minimize serial sections



Consequences

- If we don't use the parallelism, we lose it
 - Amdahl's law - serial sections
 - Von Neumann bottleneck – data transfer costs
 - Workload Imbalances
- Rethink the problem solving technique
 - Hide latency of host $\leftrightarrow$ device memory transfers
 - Global memory accesses $\rightarrow$ fast on-chip accesses
 - Coalesced memory transfers
 - Avoid costly branches, or render them harmless
- Simplified processor design, but more user control over the hardware resources

Today's lecture

- GPU Architecture
- Programming with CUDA

CUDA language extensions

- Type qualifiers to declare device kernel functions
`__global__ void matrixMul( ...)`
- Kernel launch syntax
`matrixMul<<< grid, threads >>>( ...)`
- Keywords
`blockIdx, threadIdx, blockDim, gridDim`
- Runtime, e.g. storage allocation
`cudaMalloc, cudaFree, cudaMemcpy`

Coding example – Increment Array

Serial Code

```
void incrementArrayOnHost(float *a, int N){  
    int i;  
    for (i=0; i < N; i++) a[i] = a[i]+1.f;  
}
```

```
#include <cuda.h>  
  
__global__ void incrementOnDevice(float *a, int N)  
{  
    int idx = blockIdx.x*blockDim.x + threadIdx.x;  
    if (idx<N) a[idx] = a[idx]+1.f;  
}  
  
incrementOnDevice <<< nBlocks, blockSize >>> (a_d, N);
```

Managing memory

```
float *a_h, *b_h;           // pointers to host memory
float *a_d;                 // pointer to device memory

cudaMalloc((void **) &a_d, size);

for (i=0; i<N; i++) a_h[i] = (float)i; // init host data

cudaMemcpy(a_d, a_h, sizeof(float)*N,
           cudaMemcpyHostToDevice);
```

Computing and returning result

```
int bSize = 4;
int nBlocks = N/bSize + (N%bSize == 0?0:1);
incrementOnDevice <<< nBlocks, bSize >>> (a_d, N);

// Retrieve result from device and store in b_h
cudaMemcpy(b_h, a_d, sizeof(float)*N,
           cudaMemcpyDeviceToHost);

// check results
for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);

// cleanup
free(a_h); free(b_h);
cudaFree(a_d);
```

Experiments - increment benchmark

- Total time: timing taken from the host, includes copying data to the device
- Device only: time taken on device only

N = 8388480, block size = 128, times in milliseconds, Nehalem

Reps = 10 100 1000 10000

8.5 83 830 8300 Device time

29 100 850 8300 Kernel launch + data xfer

77 770 7700 Host

16 103 a[i] = 1 + sin(a[i]) : Device)

7000 23.6 sec Sine function (Host)

Summary - Programming issues

- Branches serialize execution within a warp
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
 - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
 - Register consumption
 - Scheduling: hide latency
- Shared memory and registers do not persist across kernel invocations
- Next time: using shared memory

Summary - Programming issues

- Branches serialize execution within a warp
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
 - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
 - Register consumption
 - Scheduling: hide latency
- Shared memory and registers do not persist across kernel invocations
- Next time: using shared memory