

Lecture 19

Computing with Graphical Processing Units

Announcements

- Evaluate your TAs
 - ▶ Complete by March 11: <http://goo.gl/forms/Q17MRKRhqq>
 - ▶ You are automatically entered into a drawing for a \$100 UCSD Bookstore gift card, a \$50 Triton Cash card or a \$10 Triton Cash card. See the Terms and Rules for details:
<https://academicaffairs.ucsd.edu/Modules/Evals/Prizes.aspx>
- Peer Review Survey
 - ▶ Worth 1.5% of your final exam grade
 - ▶ Separate from CAPE
 - ▶ Run by Center for Teacher Development
 - ▶ https://www.surveymonkey.com/r/Baden_CSE160_Wi16
 - ▶ The survey will close
Sunday March 13th at 9 PM

What makes a processor run faster?

- Registers and cache
- Vectorization (SSE)
- Instruction level parallelism
- *Hiding data transfer delays*
- *Adding more cores*

Today's Lecture

- Computing with GPUs

Technology trends

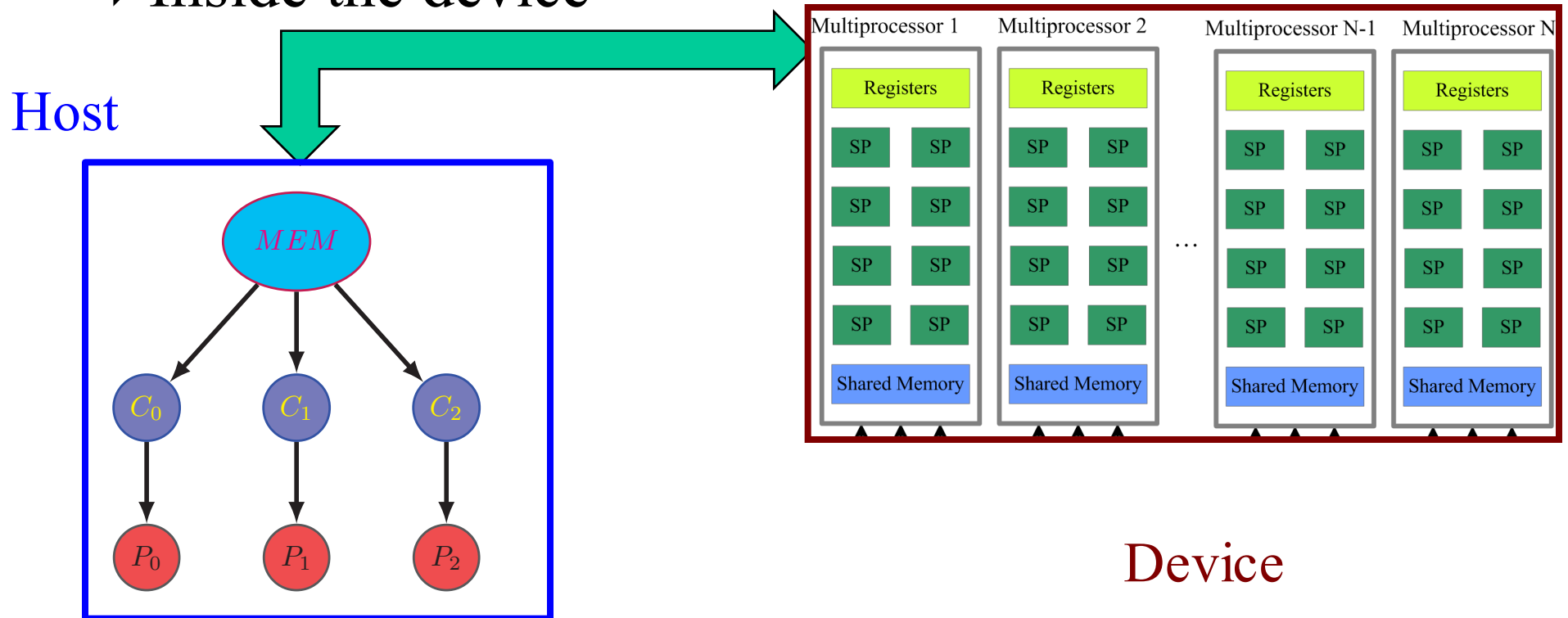
- No longer possible to use a growing population of transistors to boost single processor performance
 - Cannot dissipate power, which grows linearly with clock frequency f
 - Can no longer increase the clock speed
- Instead, we replicate the cores
 - Reduces power consumption, pack more performance onto the chip
- In addition to multicore processors we have “many core” processors
- Not a precise definition, and there are different kinds of many-cores

Many cores

- We'll look at one member of the family—
Graphical Processing Units—made by one
manufacturer—NVIDIA
- Simplified core, replicated on a grand
scale: 1000s of cores
- Removes certain power hungry features of
modern processors
 - Branches are more expensive
 - Memory accesses must be aligned
 - Explicit data motion involving on-chip memory
 - Increases performance:power ratio

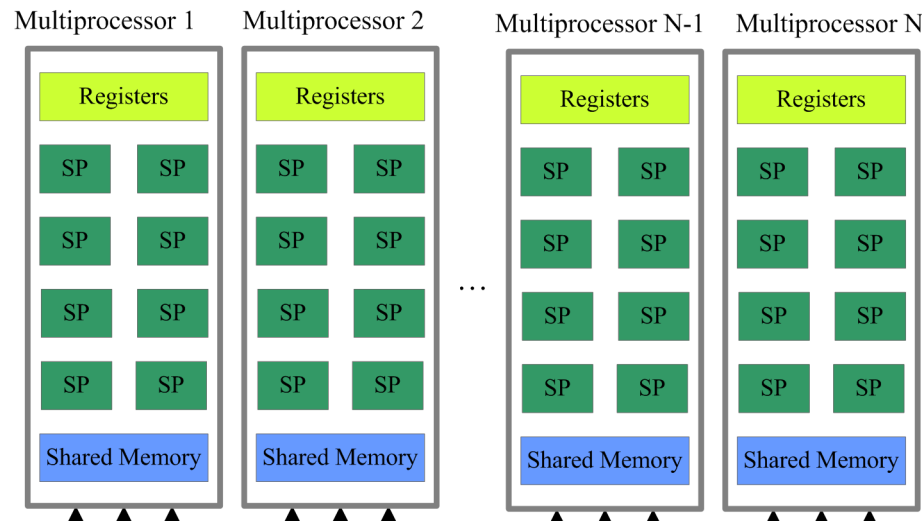
Heterogeneous processing with Graphical Processing Units

- Specialized *many-core* processor (the *device*) controlled by a conventional processor (the *host*)
- Explicit data motion
 - Between *host* and *device*
 - Inside the device



What's special about GPUs?

- Process long vectors on 1000s of specialized cores
- Execute 1000s of threads to hide data motion
- Some regularity involving memory accesses and control flow

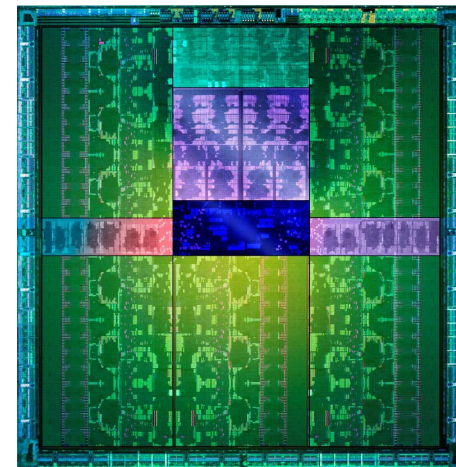


Stampede's NVIDIA Tesla K20m (GK110)

- Hierarchically organized clusters of streaming multiprocessors
 - 13 streaming processors @ 705 MHz
(down from 1.296 GHz on GeForce 280)
 - Peak performance: 1.17 Tflops/s Double Precision, fused multiply/add
- SIMT parallelism
- 5 GB “device” memory (frame buffer) @ 208 GB/s
- See international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf
www.techpowerup.com/gpudb/2029/tesla-k20m.html

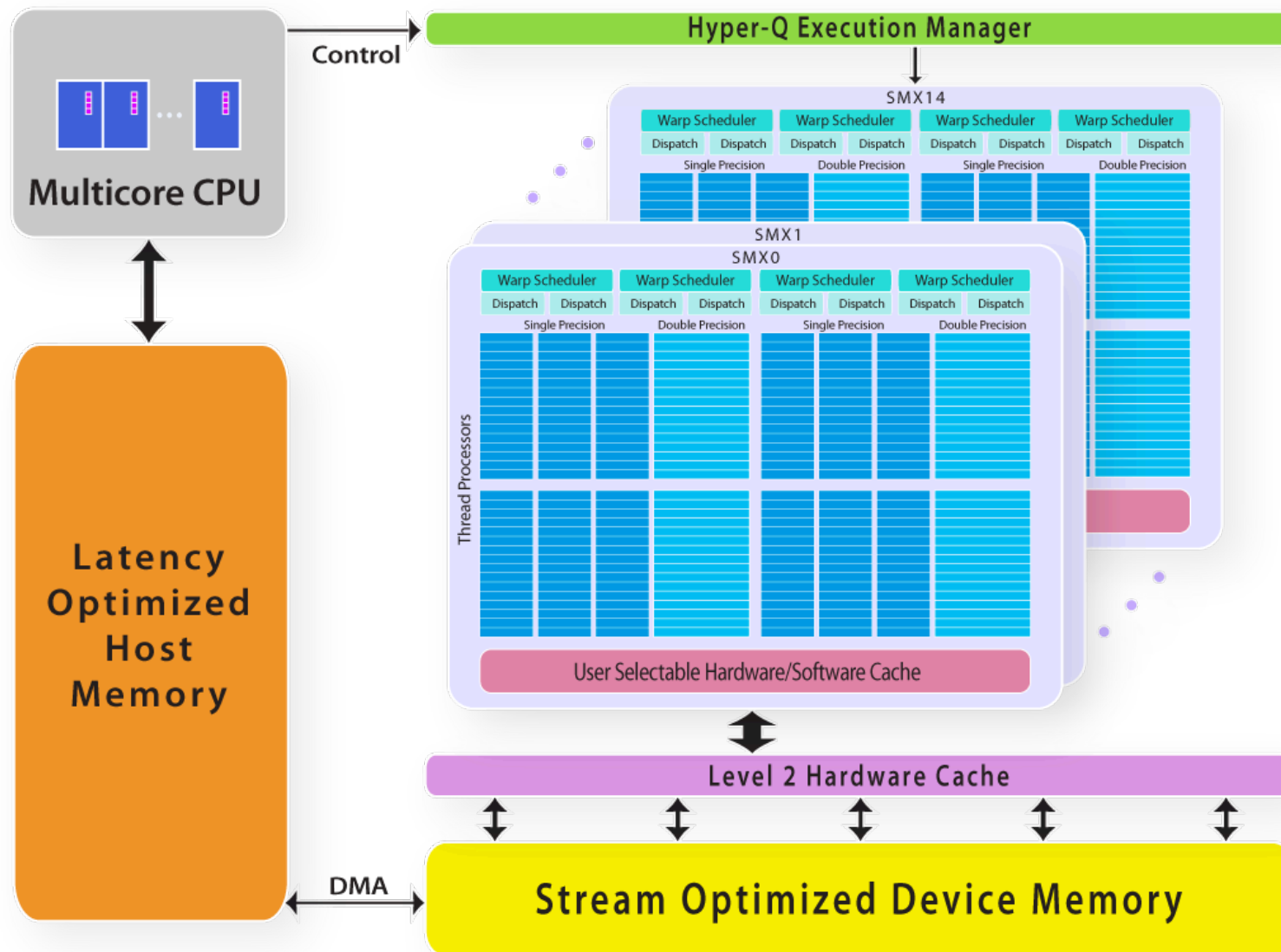


Nvidia



7.1B transistors

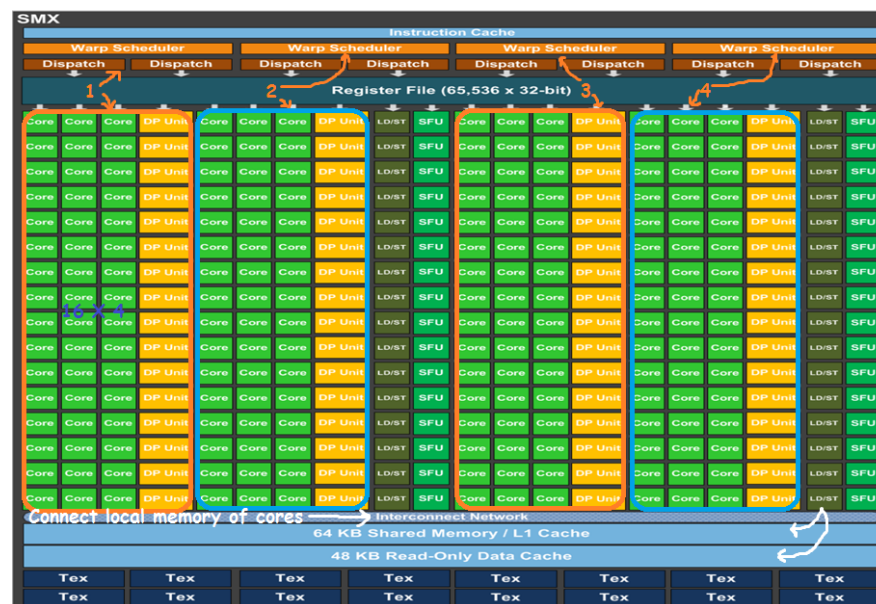
Overview of Kepler GK110

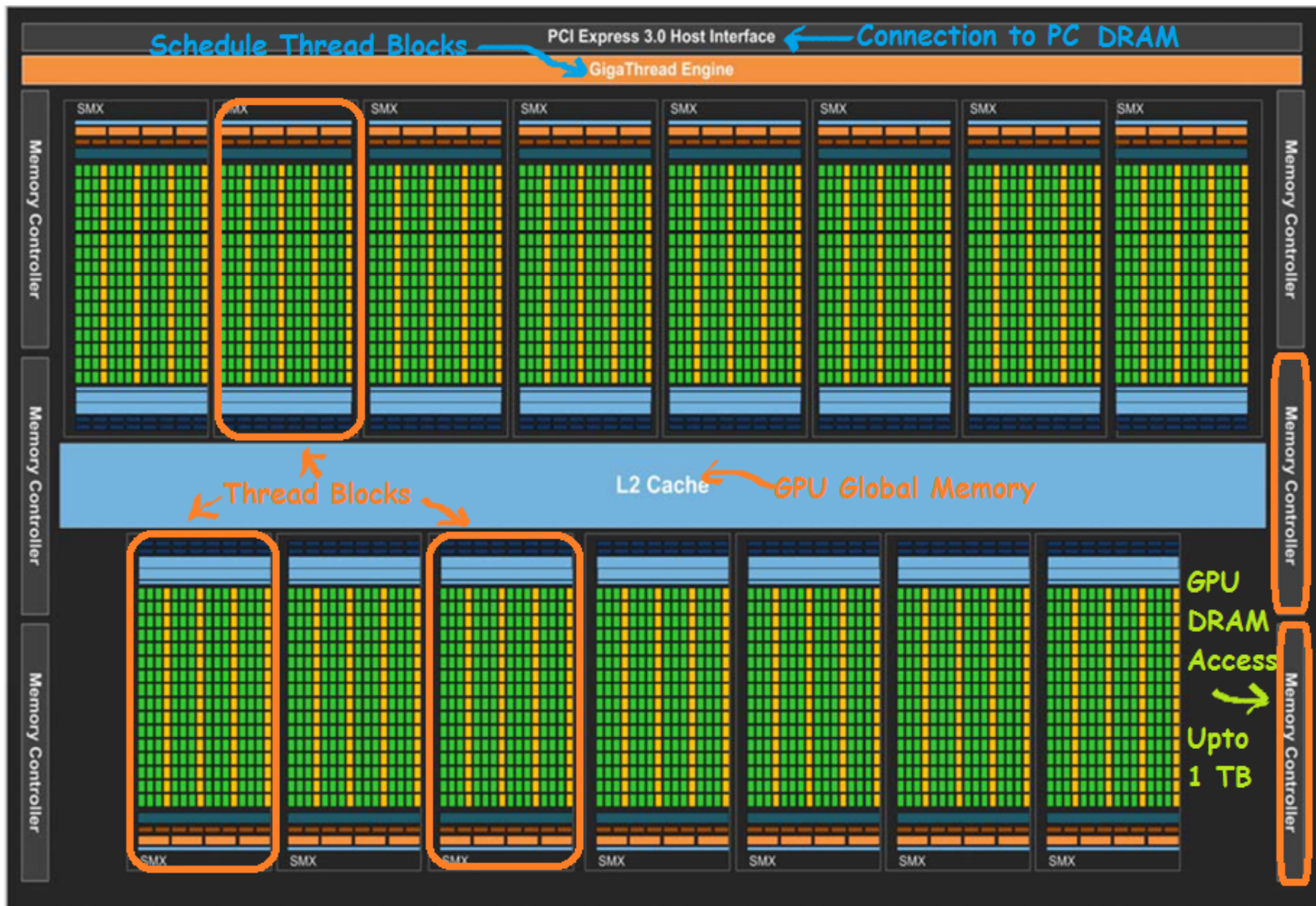


SMX Streaming processor

- Stampede's K20s (GK110 GPU) have 13 SMXs (2496 cores)
- Each SMX
 - 192 SP cores, 64 DP cores, 32 SFUs, 32 Load/Store units
 - Each scalar core: fused multiply adder, truncates intermediate result
 - 64KB on-chip memory configurable as scratchpad memory + L1 \$
 - 64K x 32-bit registers (256 (512) KB) up to 255/thread
 - $1 \text{ FMA /cycle} = 2 \text{ flops / cyc / DP core} * 64 \text{ DP/SMX} * 13 \text{ SMX} = 1664 \text{ flops/cyc}$
 $@0.7006 \text{ Ghz} = 1.165 \text{ TFLOPS per processor (2.33 for K80)}$

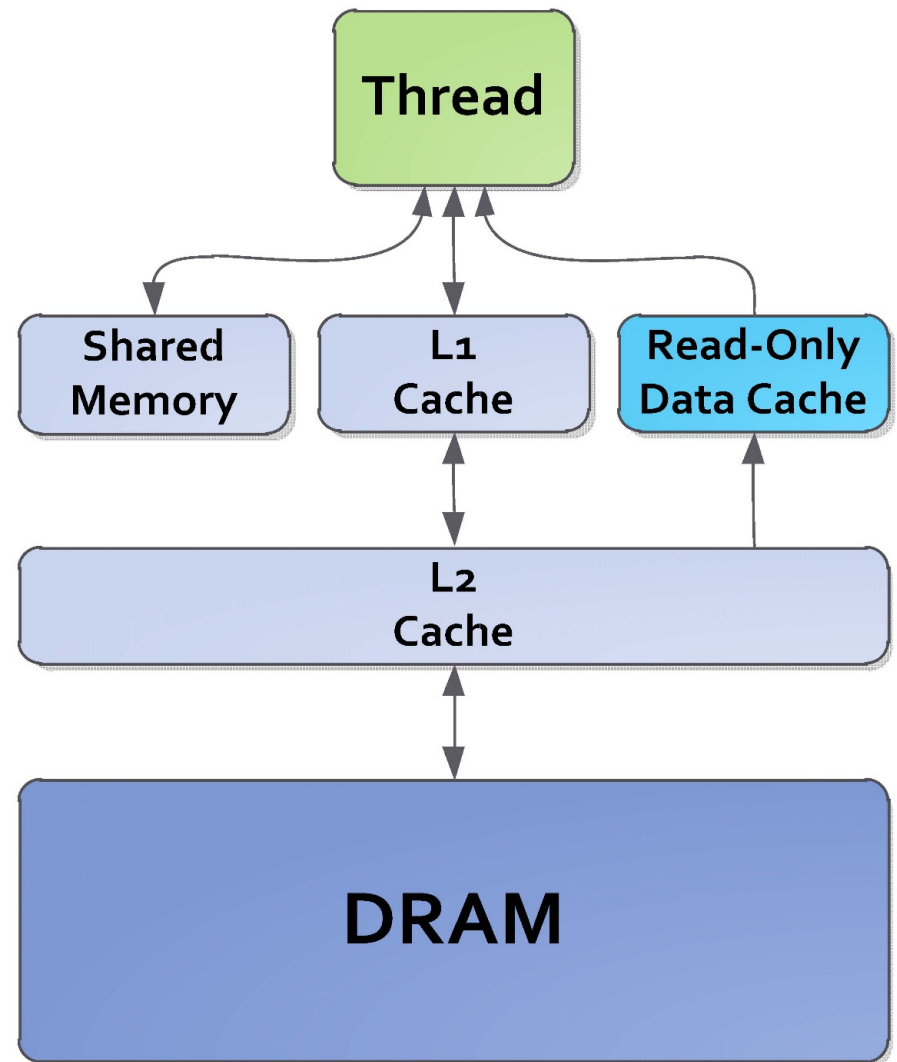
Nvidia





Kepler's Memory Hierarchy

- DRAM takes hundreds of cycles to access
- Can partition the on-chip Shared memory L1 cache
 - $\{3/4 + 1/4\}$
 - $\{3/4 + 1/4\}$
 - $\{1/2 + 1/2\}$
- L2 Cache (1.5 MB)



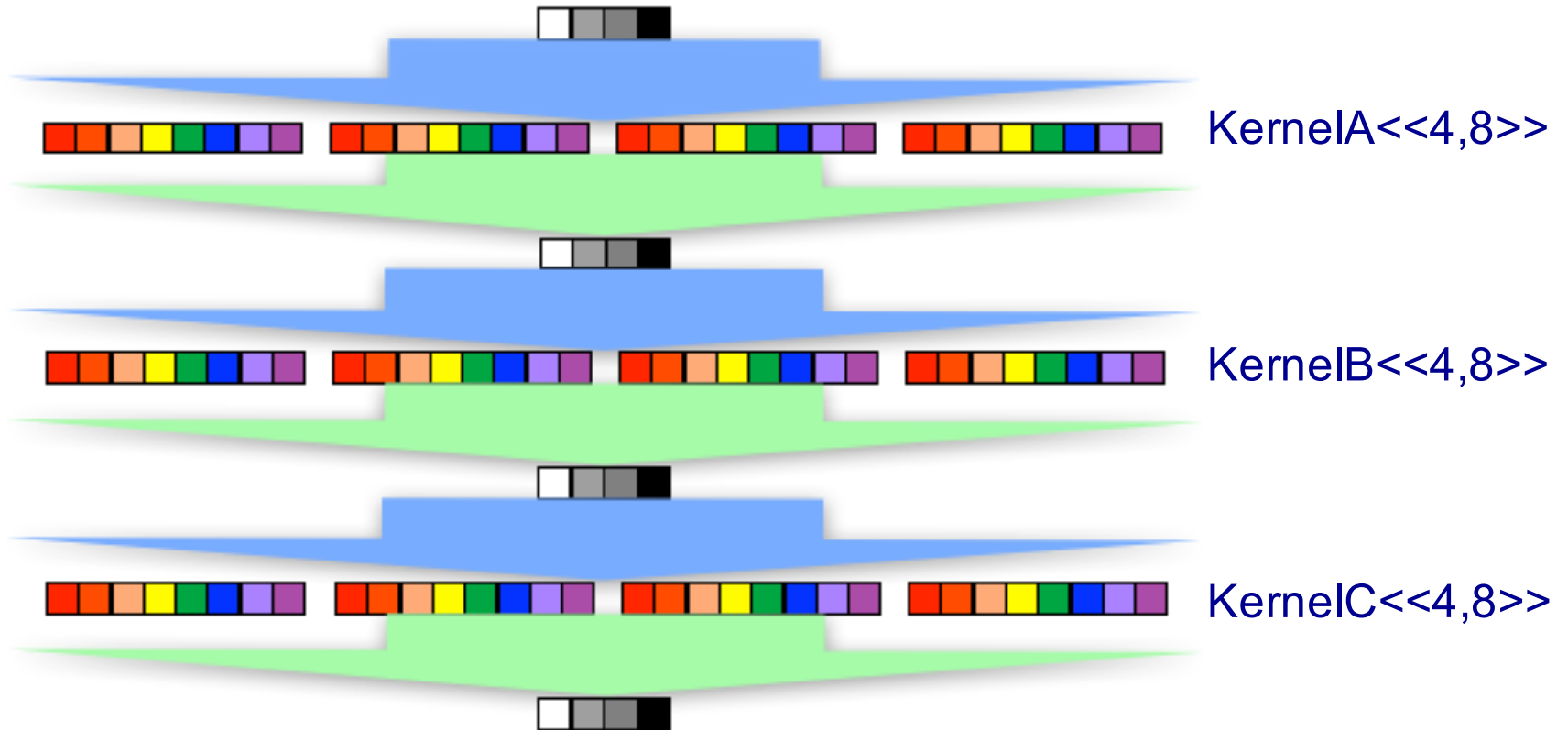
B. Wilkinson

Which of these memories are on chip and hence fast to access?

- A. Host memory
- B. Registers
- C. Shared memory
- D. A & B
- E. B & C

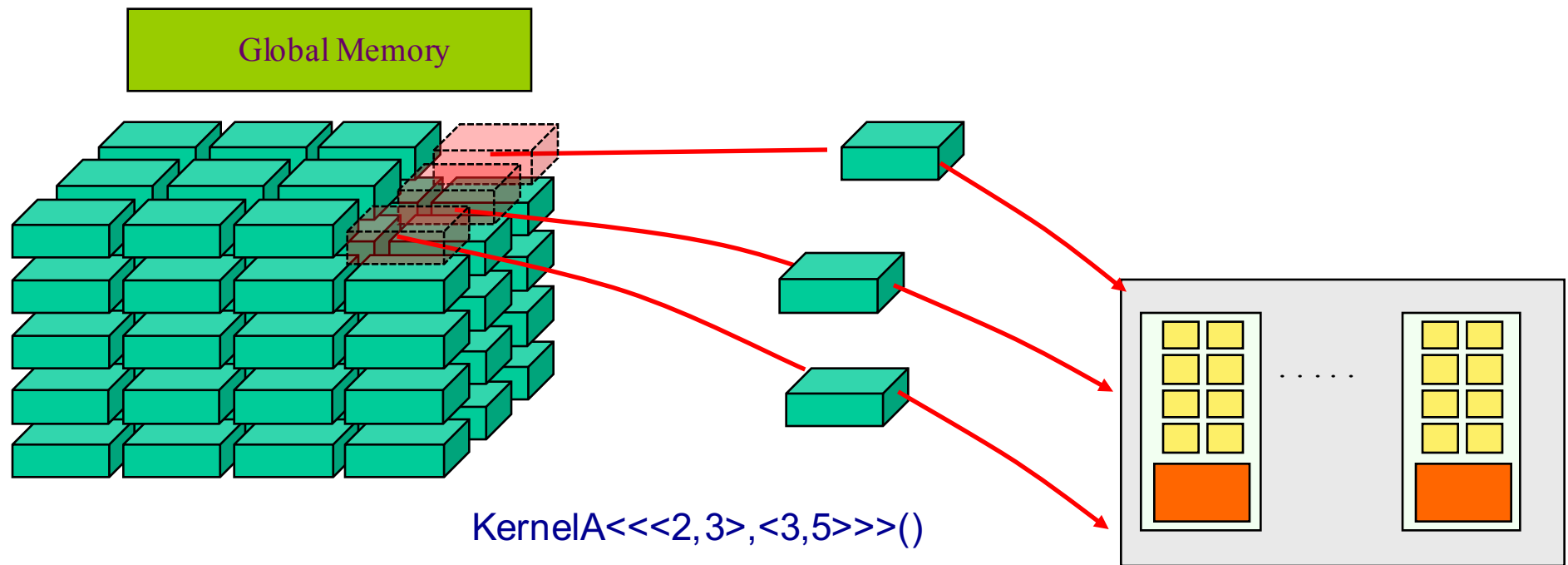
CUDA

- 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



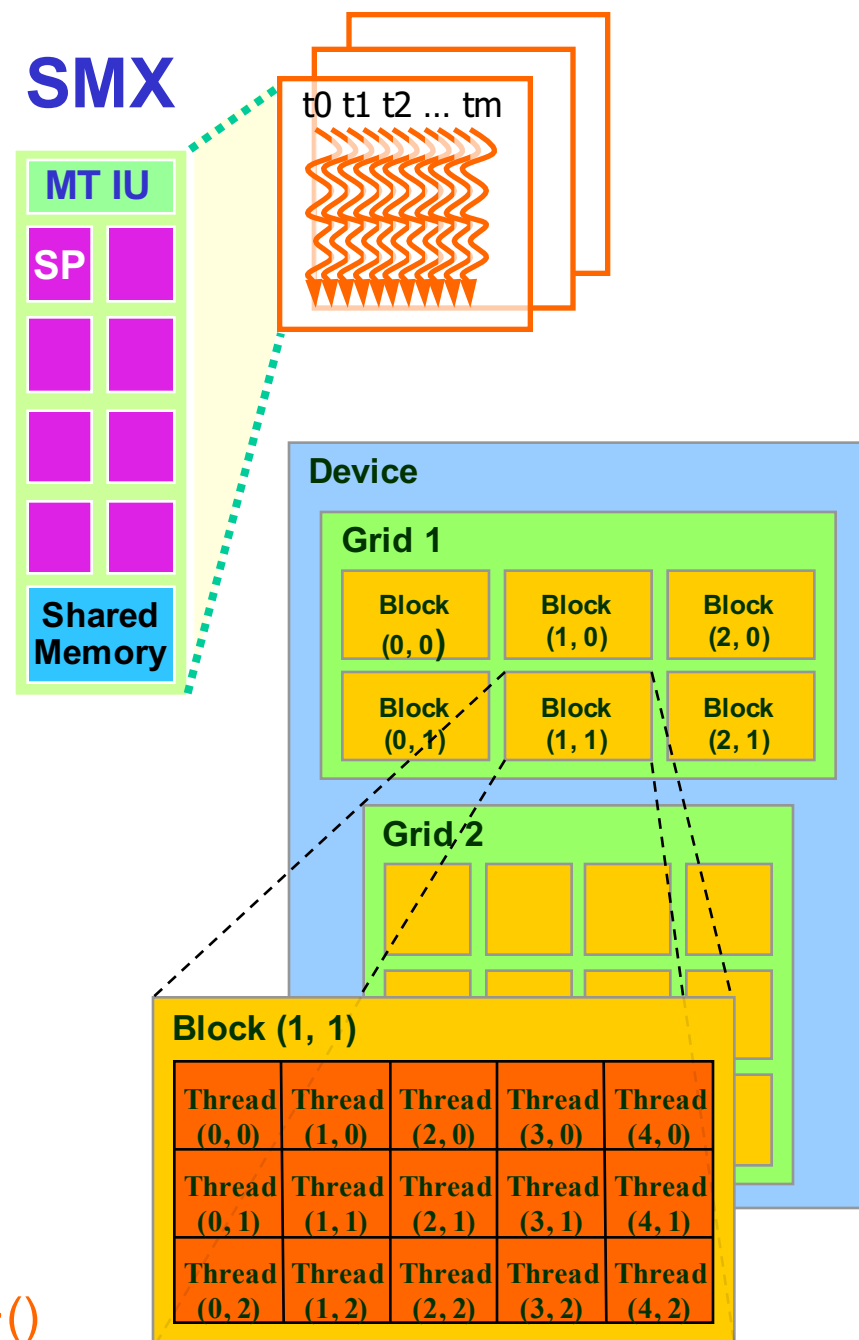
Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads **Grid** $\supset$ **Block** $\supset$ **Thread**
- Hardware dispatches blocks to cores, 0 overhead
- Compiler re-arranges loads to hide latencies



Thread block 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
 - ▶ Global synchronization also via kernel invocation
- 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

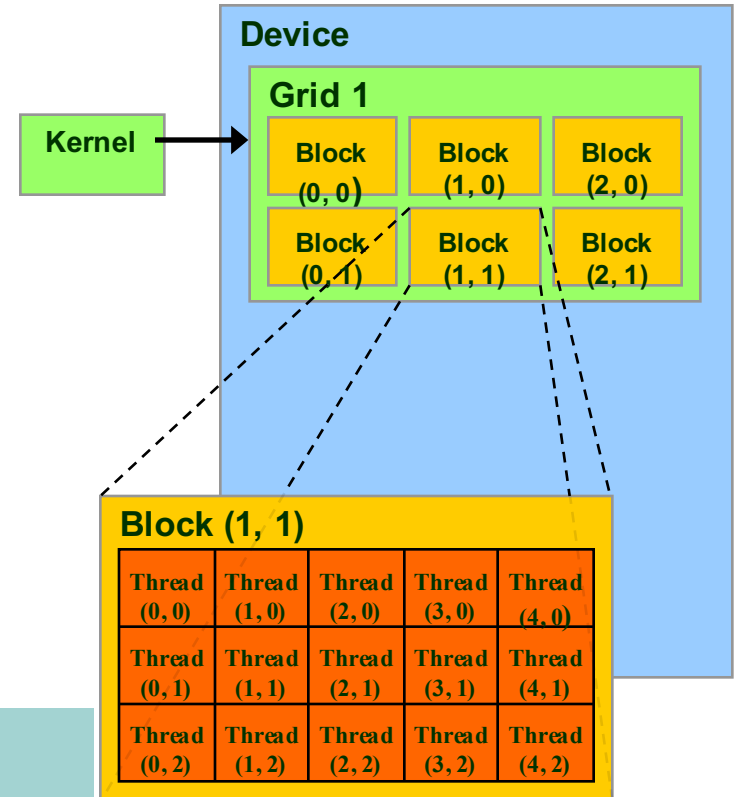
DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Which kernel call spawns 1000 threads?

- A. `KernelA<<<10,100>,<10,10>>>()`
- B. `KernelA<<<100,10>,<10,10>>>()`
- C. `KernelA<<<2,5>,<10,10>>>()`
- D. `KernelA<<<10,10>,<10,100>>>()`

Execution Configurations

- Grid $\supset$ Block $\supset$ Thread
- Expressed with *configuration variables*
- Programmer sets the thread block size, maps threads to memory locations
- Each thread uniquely specified by block & thread ID



```
__global__ void Kernel (...);
```

```
dim2 DimGrid(2,3); // 6 thread blocks
```

```
dim2 DimBlock(3,5); // 15 threads /block
```

```
Kernel<<< DimGrid, DimBlock, >>>(...);
```

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

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; }
```

CUDA

```
// Programmer determines the mapping of virtual thread IDs  
// to global memory locations
```

```
#include <cuda.h>
```

```
__global__ void incrementOnDevice(float *a, int N)
```

```
{ // Each thread uniquely specified by block & thread ID
```

```
    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

- Data must be allocated on the device
- Data must be moved between host and the device explicitly

```
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
- Loop repeats the computation inside the kernel – 1 kernel launch and 1 set of data transfers in and out of device

N = 8388480 (8M ints), block size = 128, times in *milliseconds*,

Repetitions				
10	100	1000	10 ⁴	
1.88	14.7	144	1.44s	Device time
19.4	32.3	162	1.46s	Kernel launch + data xfer

What is the cost of moving the data and launching the kernel?

- A. About 1.75 ms $((19.4-1.88)/10)$
- B. About 0.176 ms $(32.3-14.7)/100$
- C. About 0.018 ms $((162-144)/1000)$
- D. About 17.5 ms $(19.4-1.88)$

N = 8 M block size = 128, times in *milliseconds*

Repetitions

10	100	1000	10 ⁴	
1.88	14.7	144	1.44s	Device time
19.4	32.3	162	1.46s	Kernel launch + data xfer