Introduction to GPUs

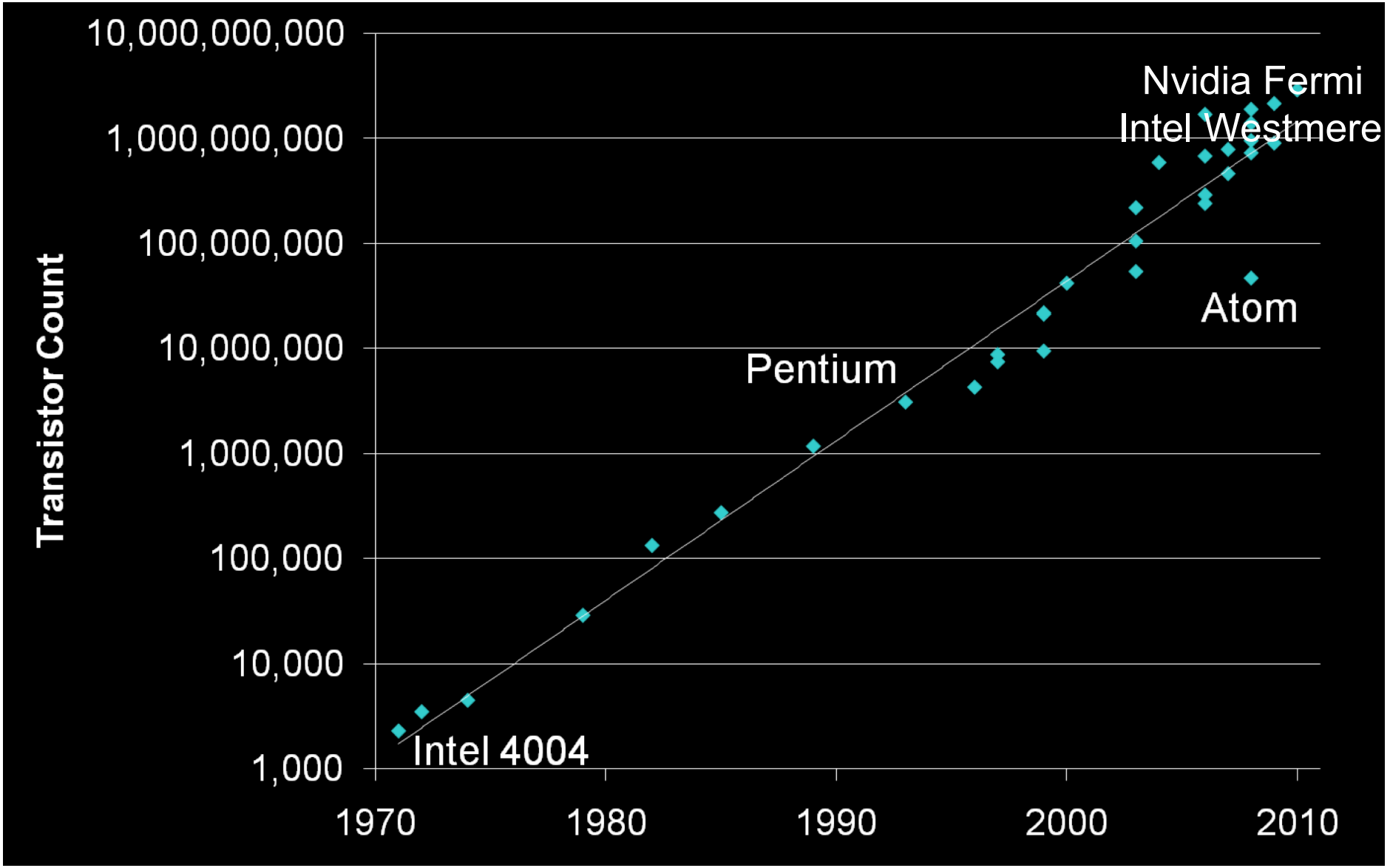
CPSC 424/524
Lecture #11
November 5, 2018

Moore's Law (paraphrased)

"The number of transistors on an integrated circuit doubles approximately every two years at constant cost."

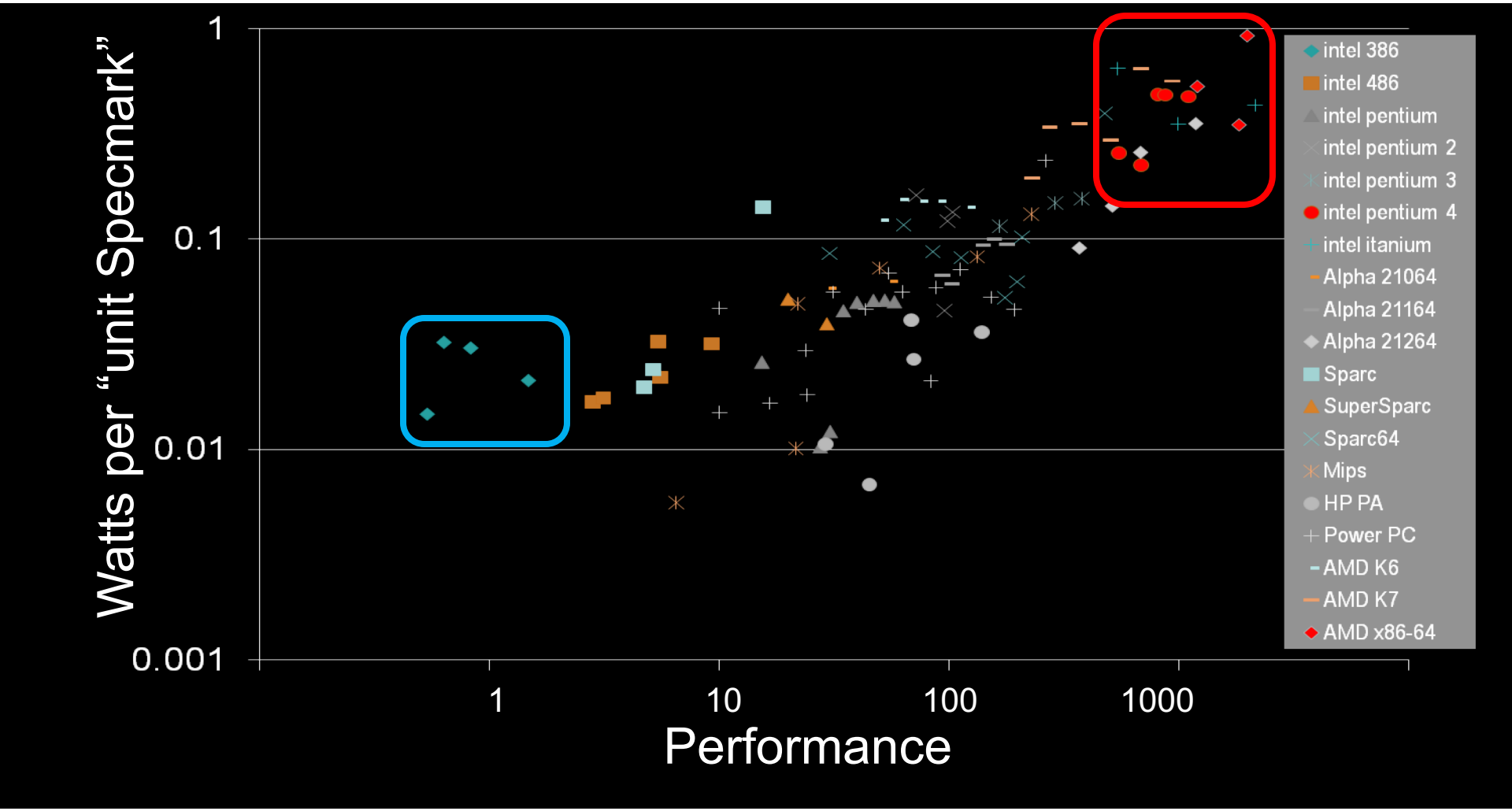
– Gordon E. Moore (ca. 1965)

Moore's Law (Visualized)



Data credit: Wikipedia

Buying Performance with Power



(courtesy Nvidia/Mark Horowitz and Kevin Skadron)

Options for Serial Performance are Limited

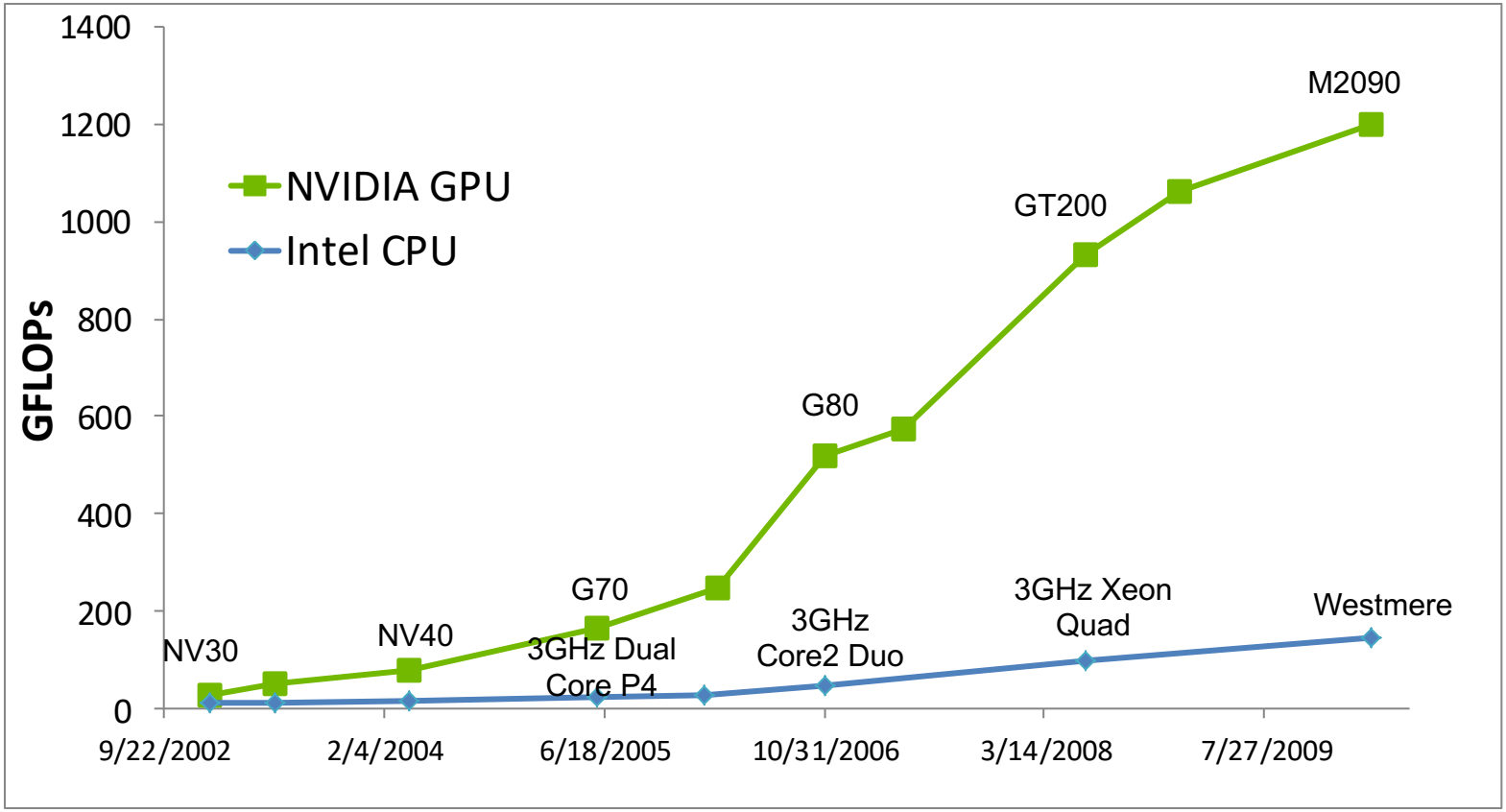
- Processor frequencies and cycle times can't continue to scale
  - no 10 GHz chips
- Power consumption can't grow
  - can't melt chip
- Moore's law still applies
  - Transistor density can continue to increase

How to Use More Transistors?

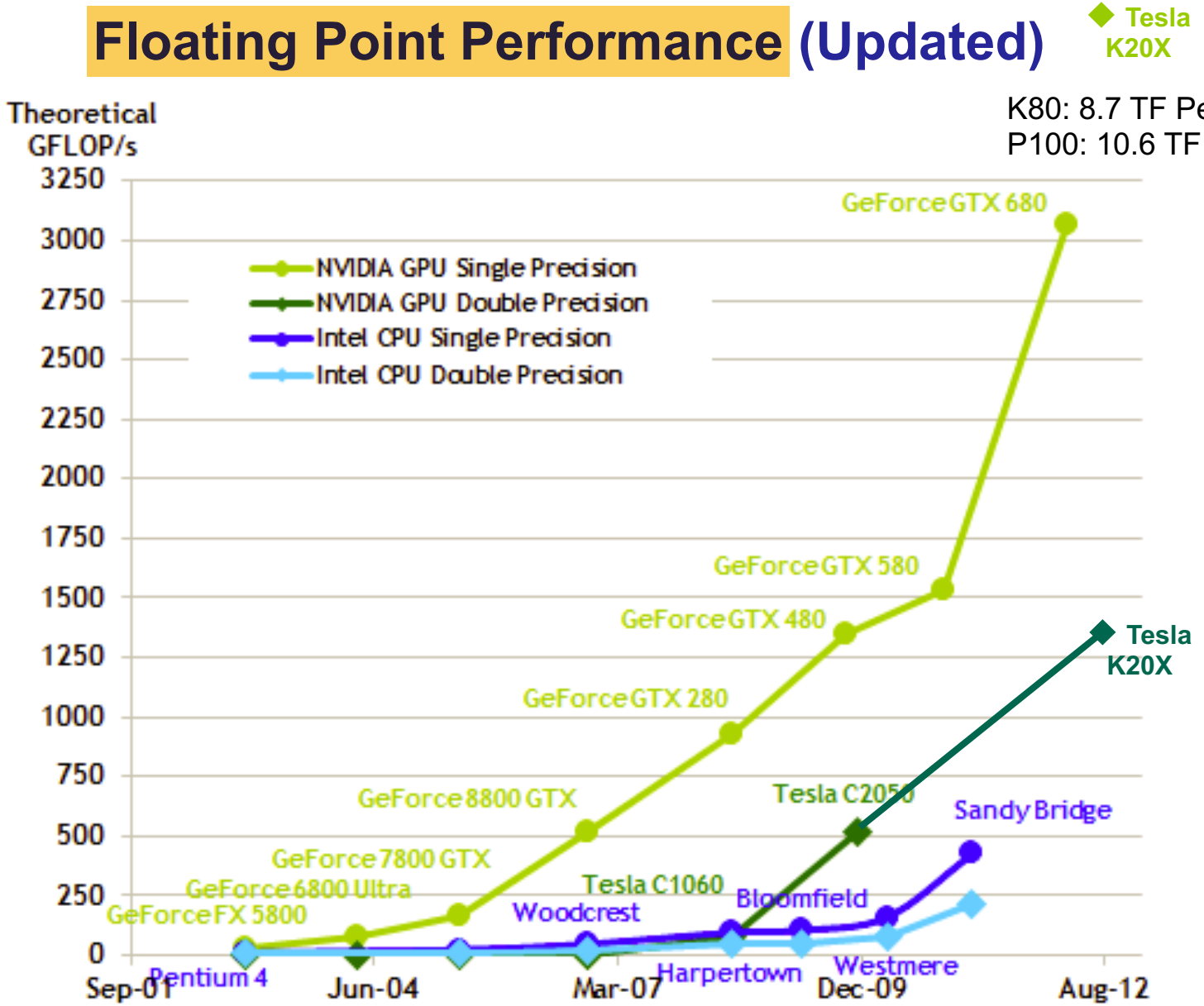
- Instruction-level parallelism
  - out-of-order execution, speculation, …
  - vanishing opportunities in power-constrained world
- Data-level parallelism
  - vector units, SIMD execution, …
  - increasing … SSE, AVX, Cell SPE, Clearspeed, GPU
- Thread-level parallelism
  - increasing … multithreading, multicore, manycore
  - Intel Core2, AMD Phenom, Sun Niagara, STI Cell, NVIDIA Fermi, Intel Knight's Corner, Intel Knight's Landing, …

Why Choose Data-Level Parallelism?

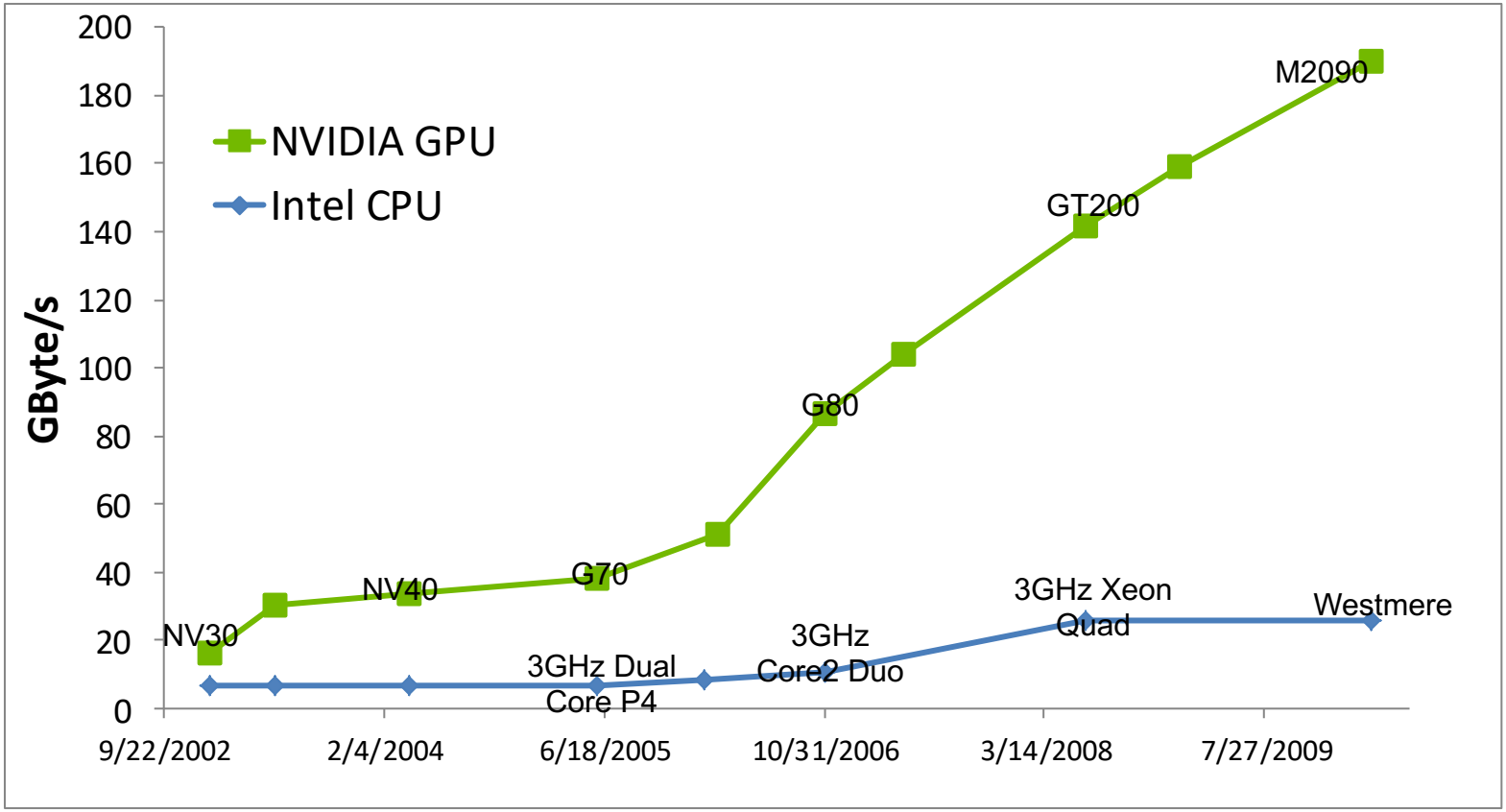
More (Single Precision) FLOPs per transistor---for the right problems



Floating Point Performance (Updated)

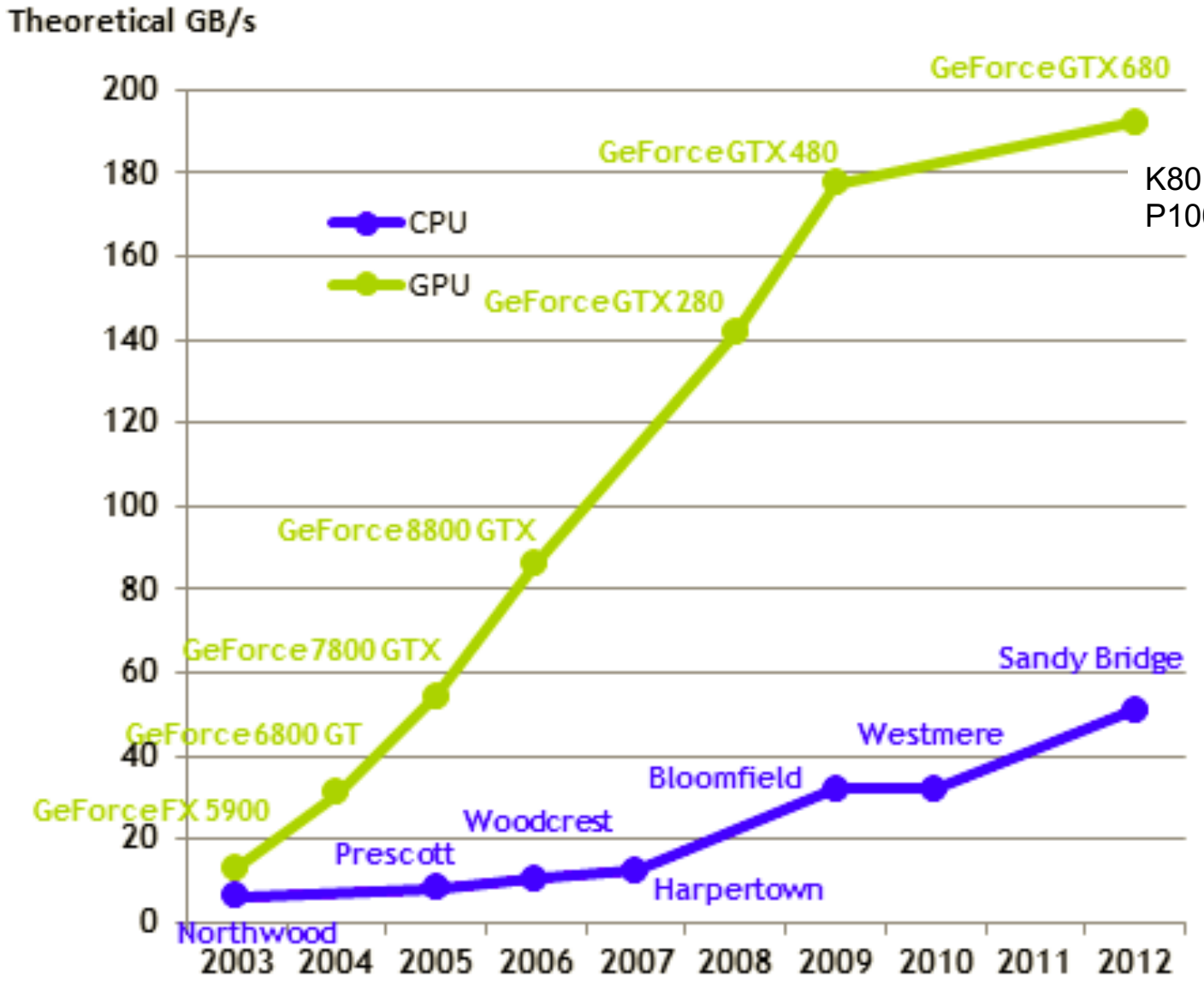


Memory Bandwidth is Crucial



Memory Bandwidth (Updated)

◆ K20X
250 GB/s

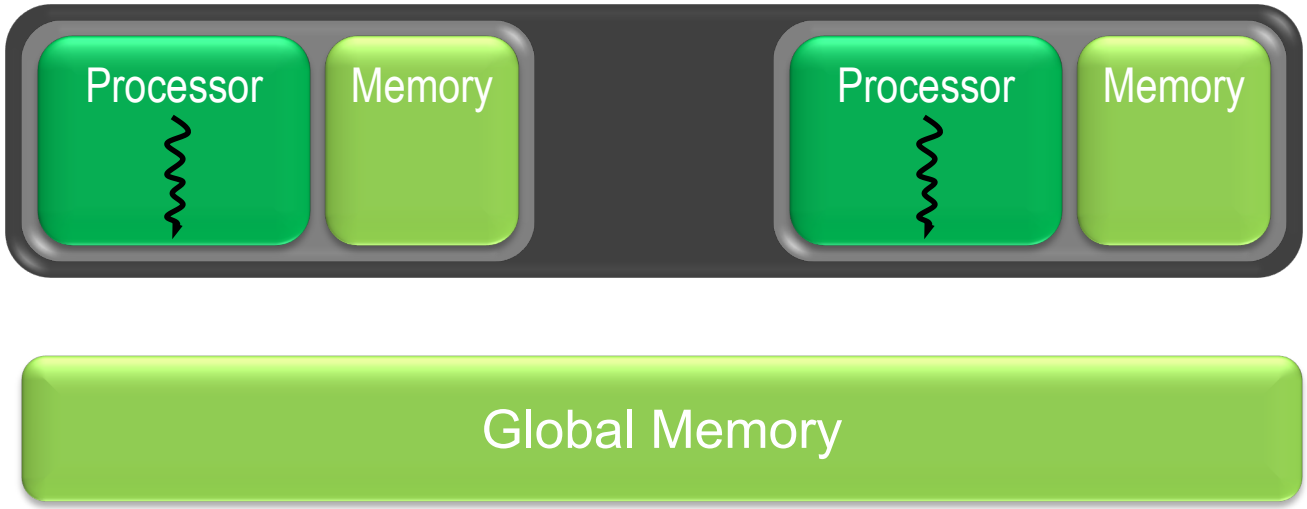


K80: 480 GB/s Peak
P100: 720 GB/s Peak

The New Rules (according to NVIDIA)

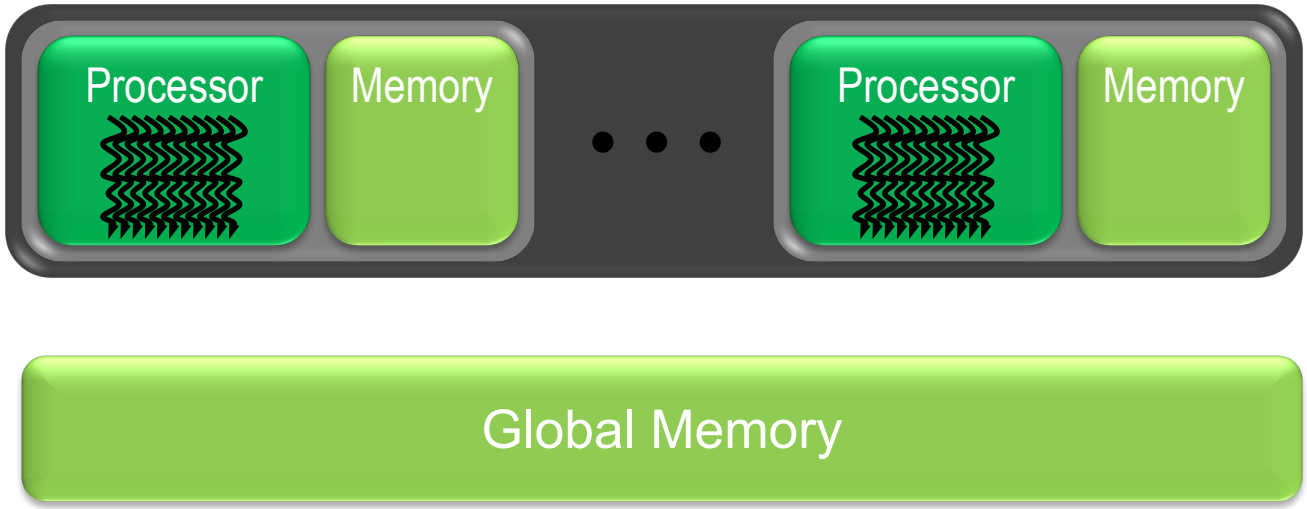
- Computers no longer get faster, just wider
- You must re-think your algorithms to be parallel !
- Data-parallel computing is most scalable solution
  - Otherwise: continually refactor code for a growing number of cores
  - You will always have more data than cores – build the computation around the data

Generic Multicore Chip



- Handful of CPUs/cores, each supporting ~1 hardware thread
- On-chip memory near CPUs (cache, RAM, or both)
- Shared global memory space (external DRAM on-board, not on-chip)

Generic Manycore Chip



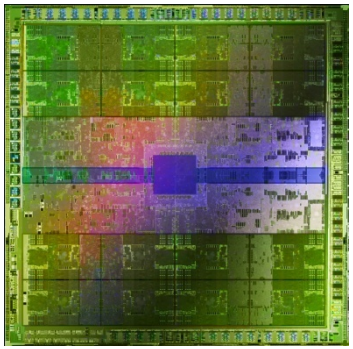
- Many CPUs/cores each supporting multiple hardware threads
- On-chip memory near CPUs (cache, RAM, or both)
- Shared global memory space (on-board DRAM)

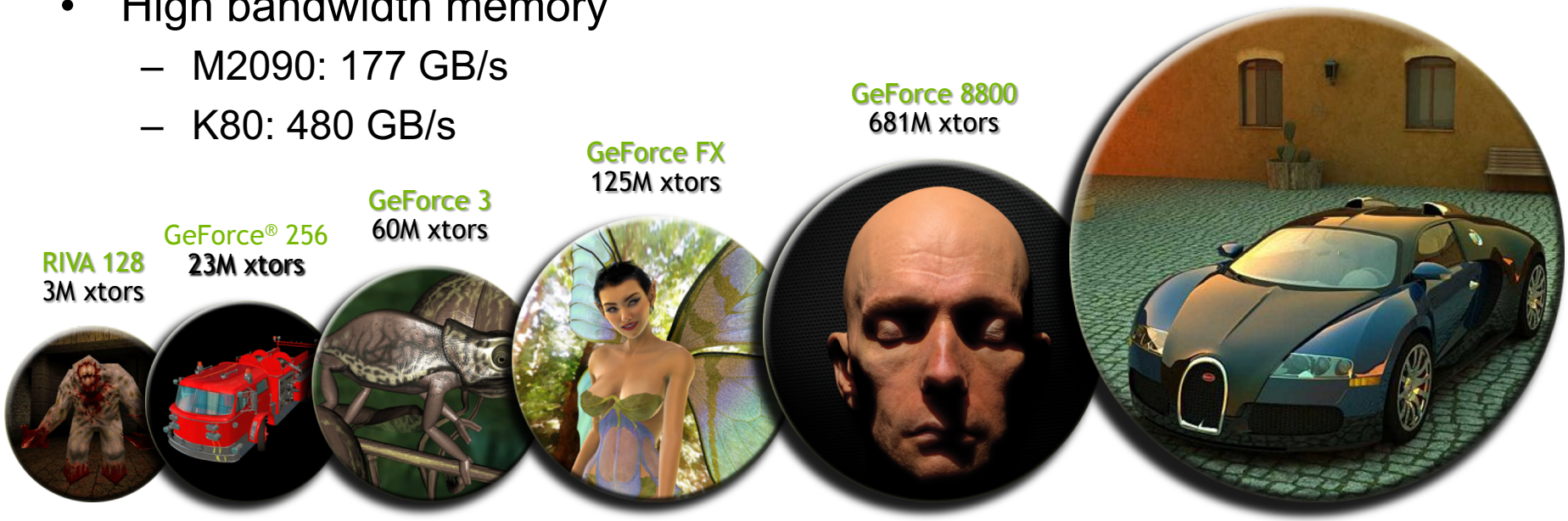
GPUs

- Massive economies of scale
- Massively parallel
- Driven by commodity market (Gaming, Video, etc.)

GPU Evolution

- High performance computation
  - Tesla M2090: 1331 SP GFLOP/s (665 DP)
  - Tesla K80: 8730 SP GFLOP/s (2910 DP)
- Large memories
  - M2090: 6 GB global, caches, “nearby” Shared Mem.
  - K80: 24 GB global (plus similar hierarchy)
- High bandwidth memory
  - M2090: 177 GB/s
  - K80: 480 GB/s





Lessons from Graphics Pipeline

- Throughput is paramount
  - must paint every pixel within frame time
  - scalability

- Create, run, & retire lots of threads very rapidly
  - Min: 2 Mpixels * 60 fps *2 = 240 Mthreads/sec for gaming
  - Measured: 14.8 Gthread/s or better

- Use multithreading to hide latency
  - 1 stalled thread is OK if 100 are ready to run

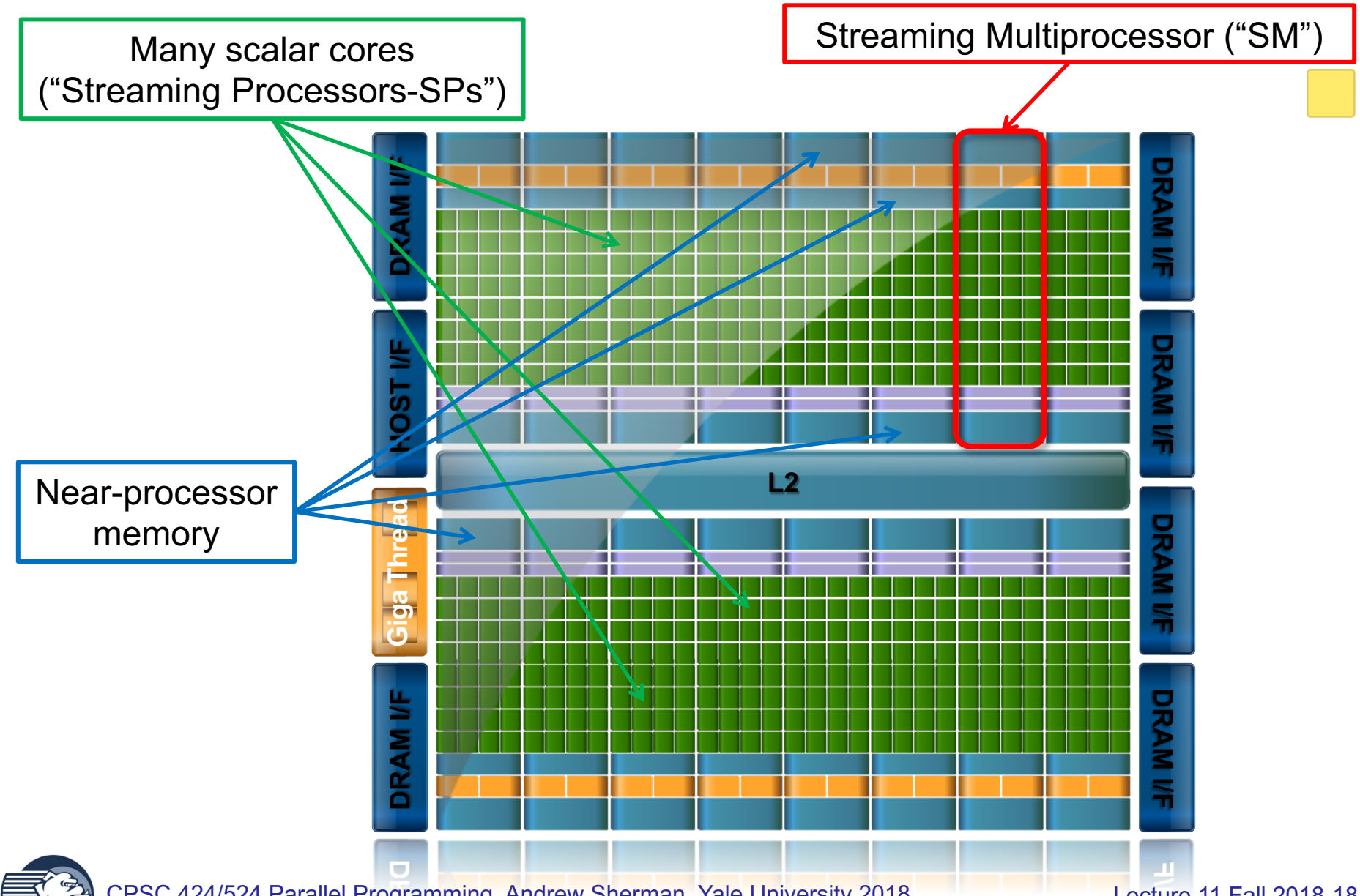
Why is this different from a CPU?

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not

- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic

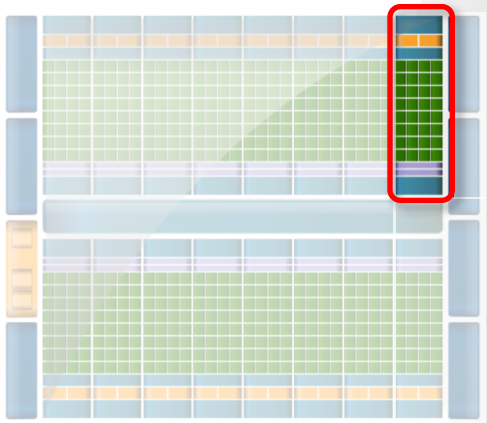
- GPU: maximize throughput of all threads
  - # threads in flight limited by resource availability
    => need lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches
  - share control logic across many threads

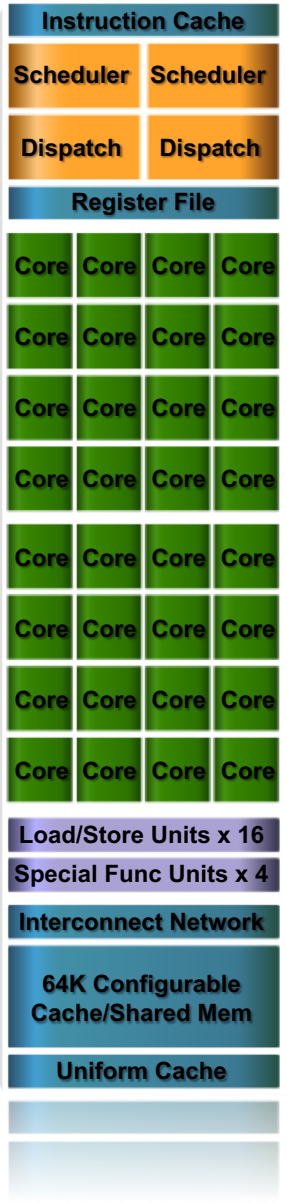
NVIDIA Fermi GPU Architecture



Fermi SM Multiprocessor

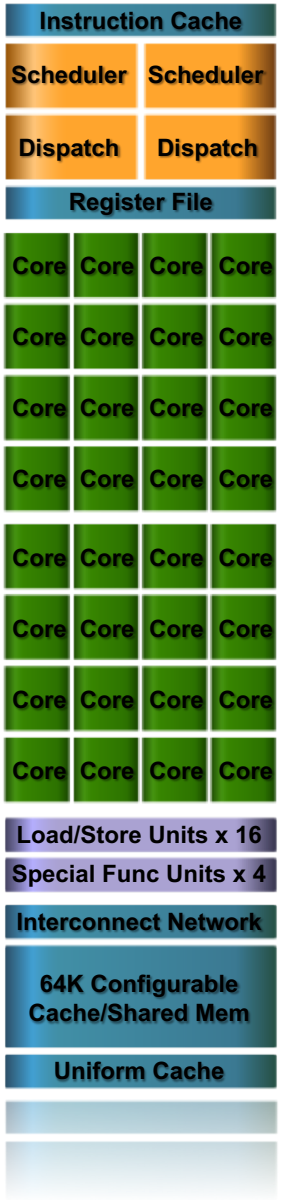
- 32 hardware cores per SM
  (16 SMs/chip give 512 total cores)
- High-performance 64-bit FP
  - 50% of peak FP32 performance
  - 8x over previous generation
- Direct load/store to memory
  - High bandwidth (177 GB/sec for M2090 in aggregate)
- 64KB of fast, on-chip RAM
  - Software or hardware-managed
  - Shared by the SPs in 1 SM
  - Enables thread communication
  - Essential to performance





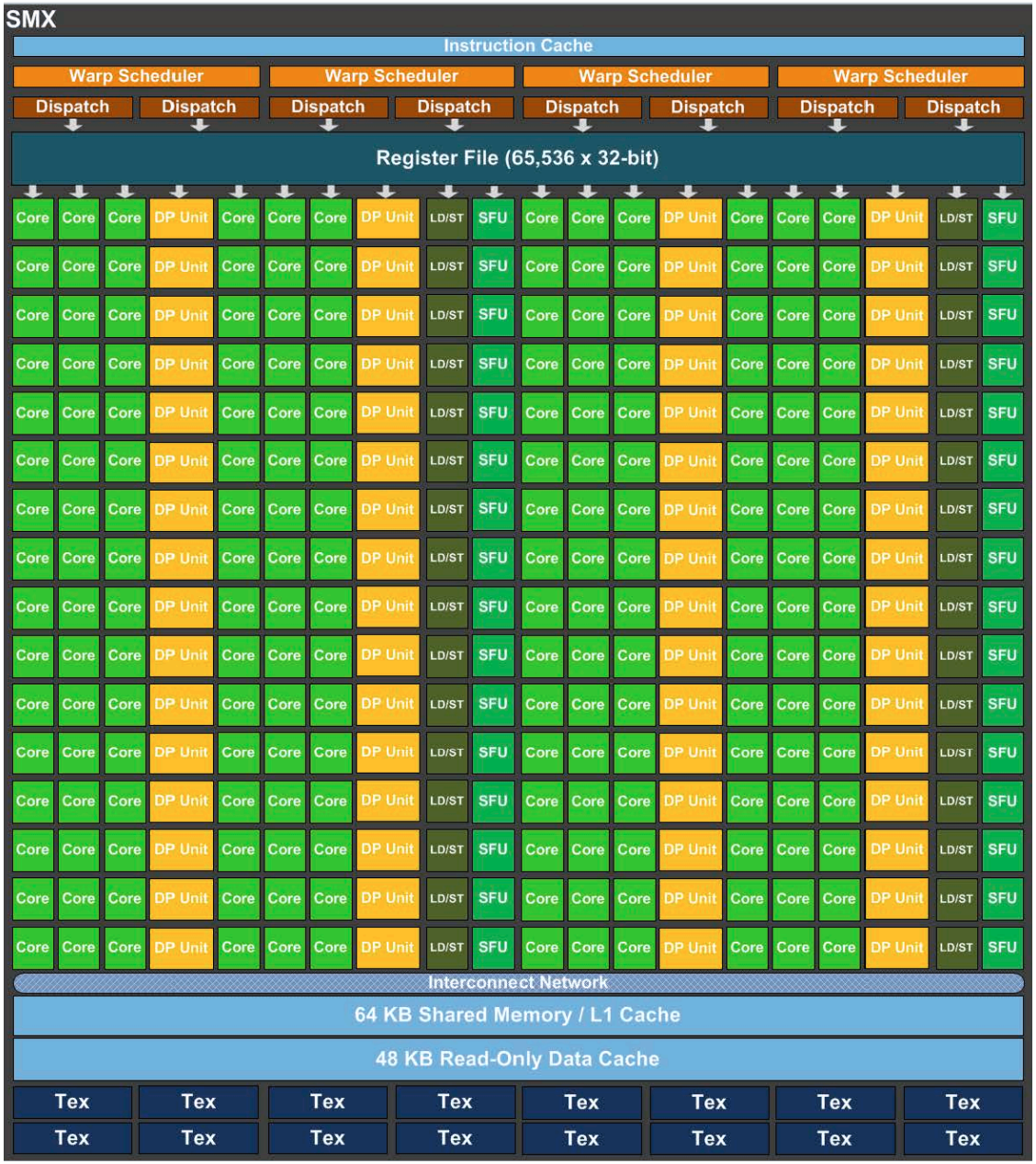
Many-threaded Computation

- SIMT (Single Instruction Multiple Thread) execution
  - Many threads per SM
  - Threads run in groups of 32 called “warps”
  - “Ready-to-run” warps can run in any order
  - Threads in a warp run in SIMD mode with “divergence”
  - HW automatically handles thread 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
  - Context switching is (nearly) free in SM



NVIDIA Kepler GPU Architecture

Kepler SMX Multiprocessor



CUDA: A way to program GPUs

- Scalable parallel programming model
- Minimal extensions to familiar C/C++ environment
- CUDA runs on both CPUs and GPUs

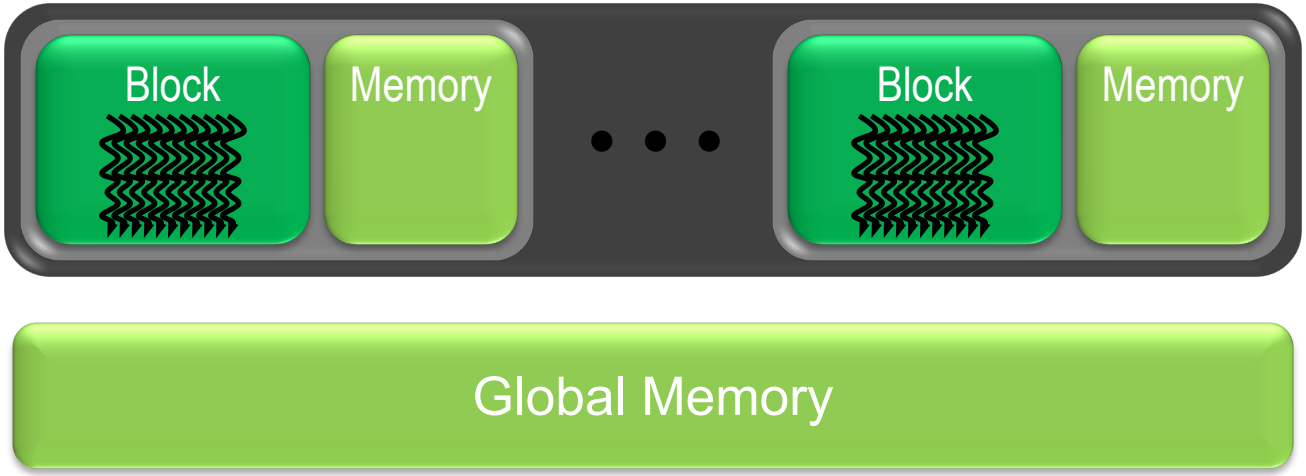
CUDA: Scalable parallel programming

- Augments C/C++ with minimalist abstractions
  - Lets programmers focus on parallel algorithms, not mechanics of a parallel programming language

- Maps straightforwardly onto hardware
  - Good fit to GPU architecture
    - GPU threads are very lightweight — create / switch is (almost) free
  - Maps well to multi-core CPUs (useful for debugging)

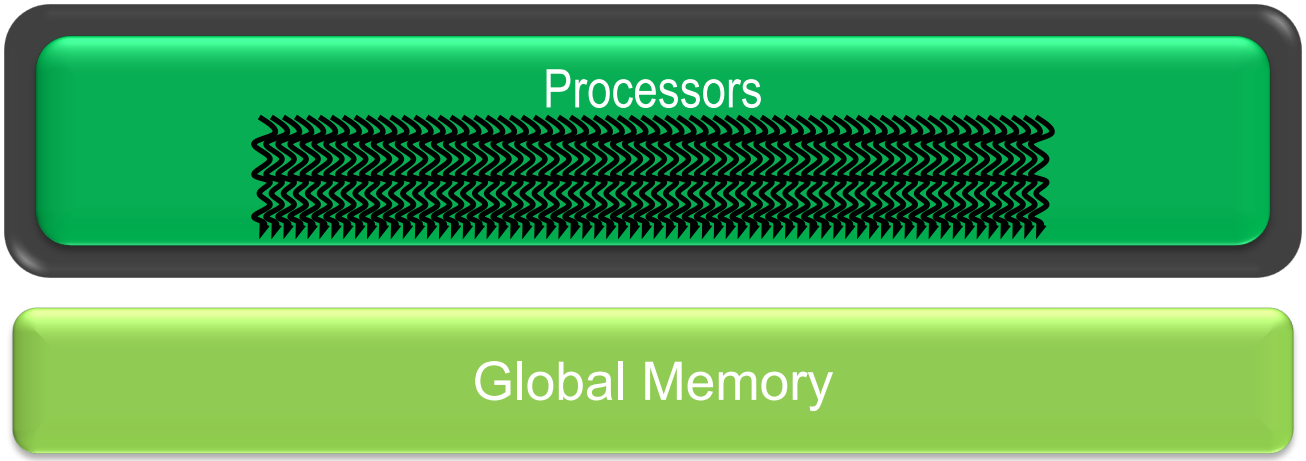
- Scales extremely well
  - Hundreds of cores & thousands of parallel threads
  - GPU needs 1000s of threads for full utilization

CUDA Model of Parallelism



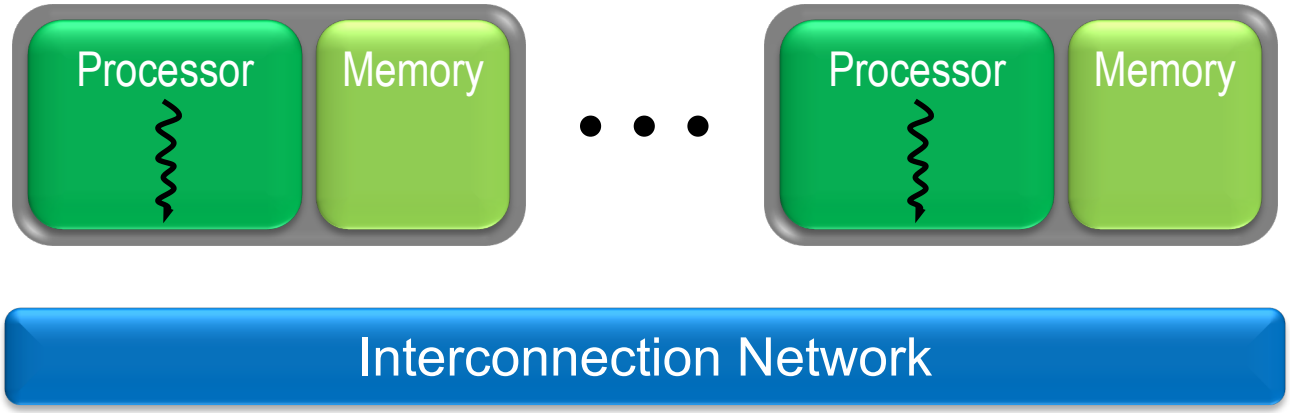
- CUDA virtualizes the physical hardware
  - Thread is like a virtualized scalar processor (registers, PC, state)
  - Block is like a virtualized multiprocessor (threads, shared mem.)
- Execution is scheduled onto physical hardware without pre-emption
  - Threads/blocks launch & run to completion
  - Blocks must be independent since they may run in any order

NOT: Flat Multiprocessor



- Global synchronization isn't cheap
- Global memory access times are expensive

NOT: Distributed Processors



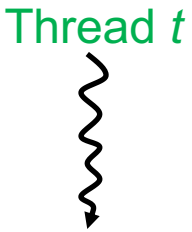
- Distributed computing is a different setting
  - May be more flexible or applicable
  - Depends on network characteristics

Key Parallel Abstractions in CUDA

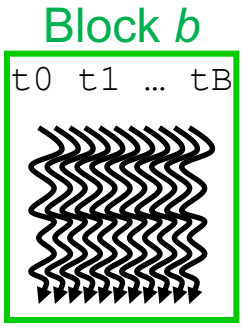
- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads

Hierarchy of concurrent threads

- Parallel kernels are composed of many threads
  - All threads execute the same sequential program
  - Threads may “diverge” via program logic

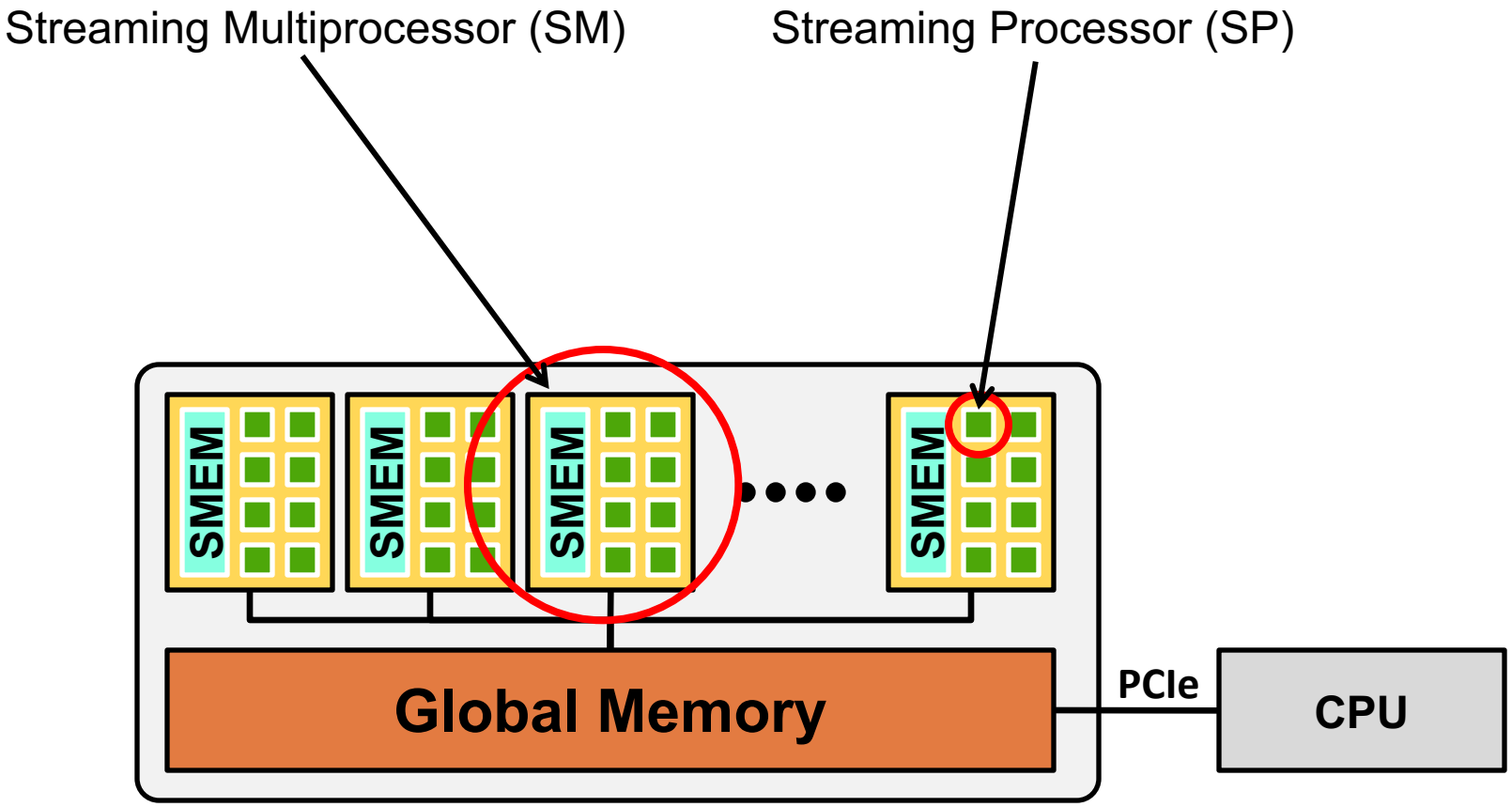


- Threads are grouped into “thread blocks”
  - Blocks are multidimensional groups of threads
  - Blocks must be independent and may run in any order
  - Threads have unique IDs based on positions in blocks
  - Threads in the same block may cooperate via shared memory and/or synchronization

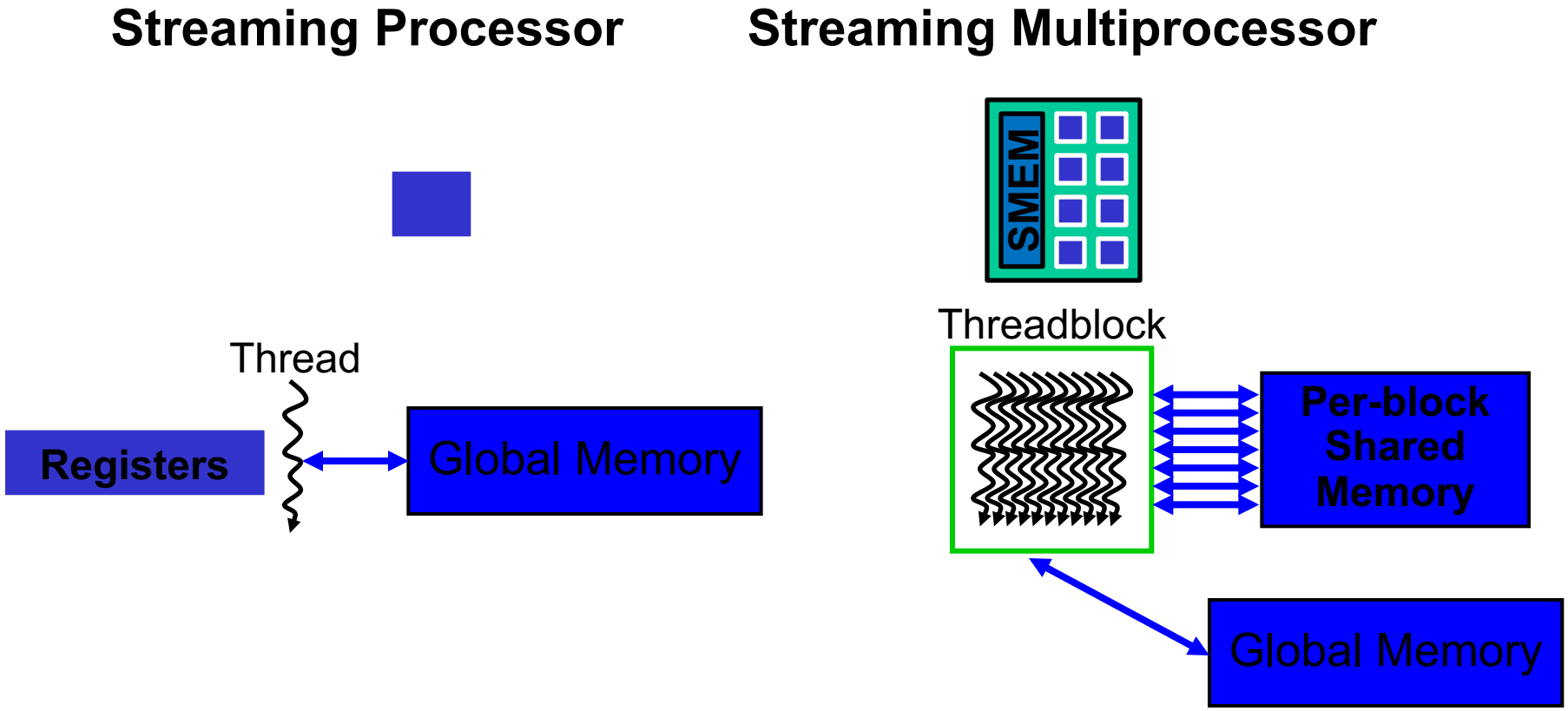


- Blocks are grouped into a “grid”
  - A grid is the multidimensional group of blocks in a single “kernel”
  - Threads in different blocks cooperate via global memory
  - Blocks have unique IDs based on position in the grid

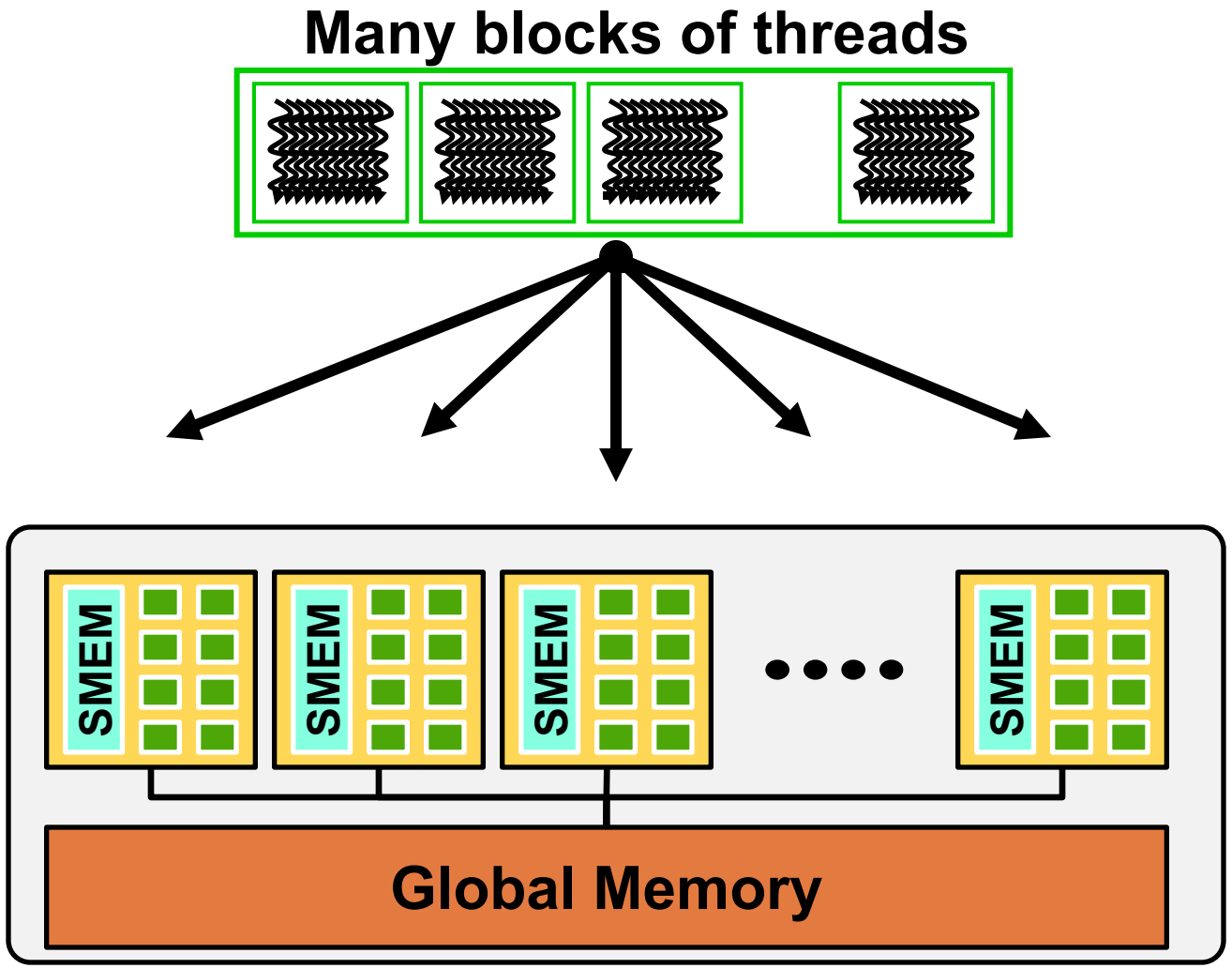
High-level view of GPU hardware



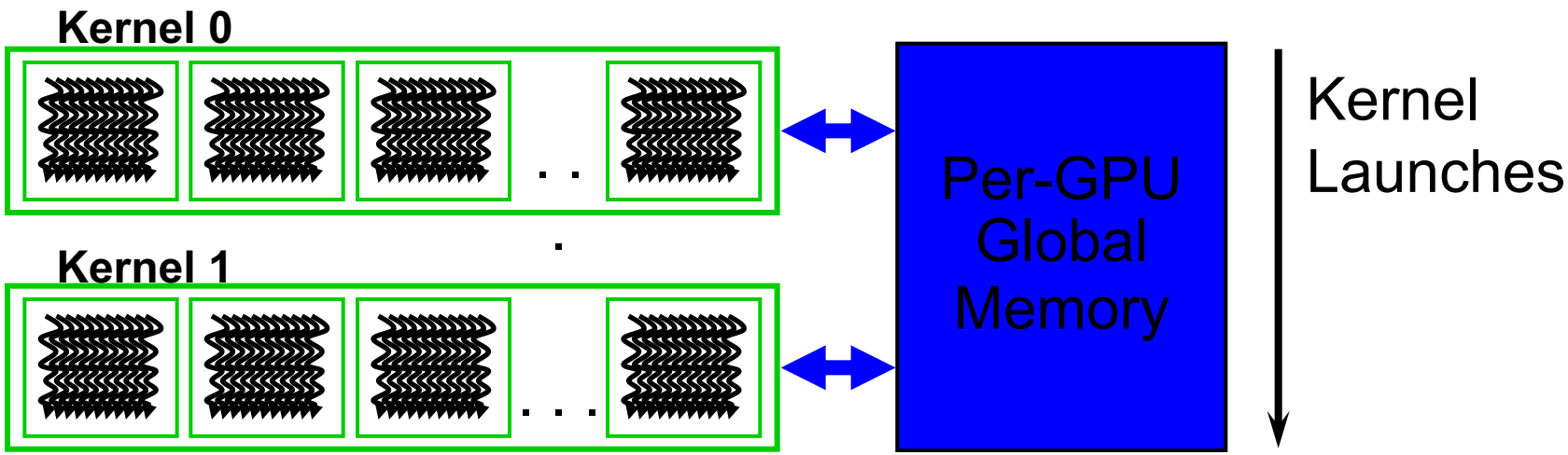
Blocks of threads run on an SM



Whole grid runs on GPU as a single kernel launch



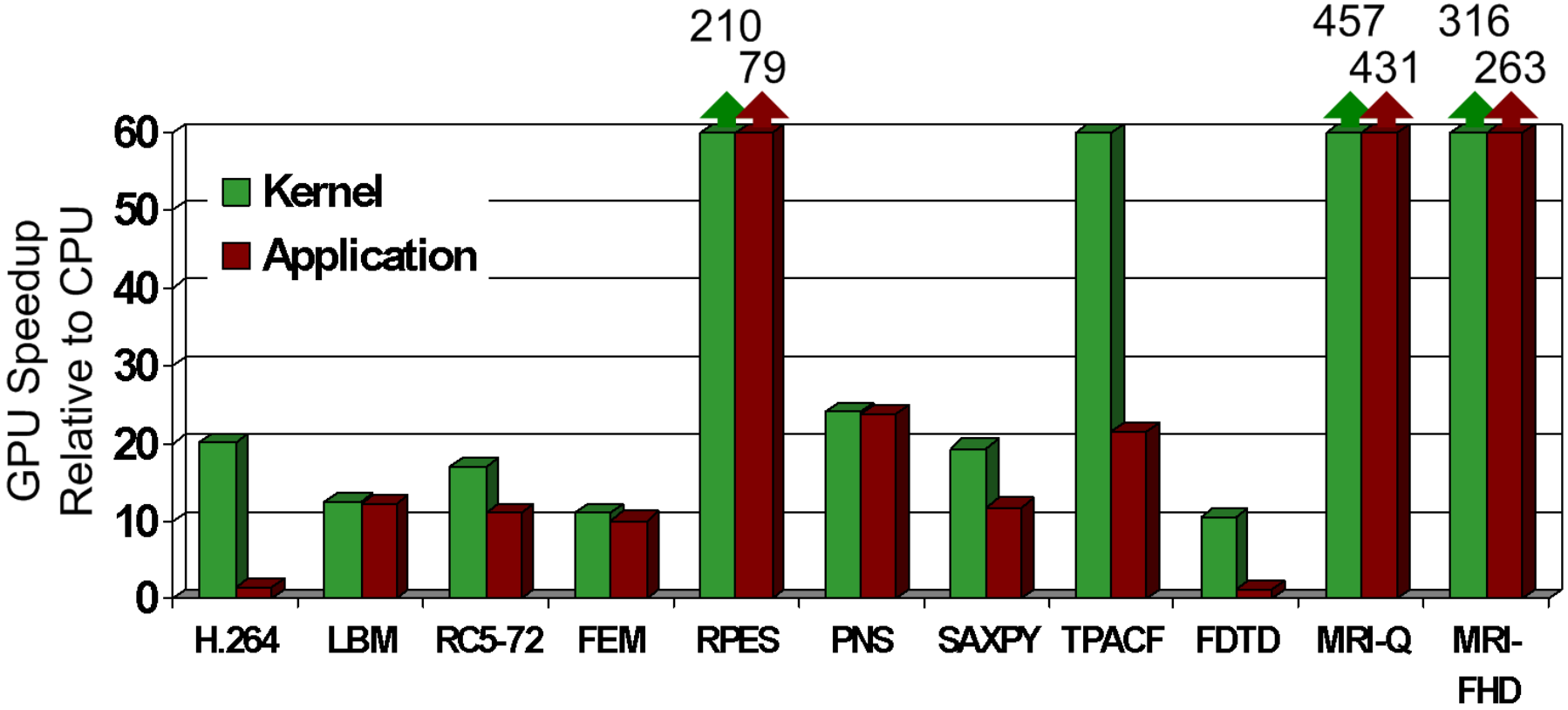
Memory Model



Sample Performance (Student Projects in UIUC Course)

Application	Description	Source	Kernel	% time
H.264	SPEC '06 version, change in guess vector	34,811	194	35%
LBM	SPEC '06 version, change to single precision and print fewer reports	1,481	285	>99%
RC5-72	Distributed.net RC5-72 challenge client code	1,979	218	>99%
FEM	Finite element modeling, simulation of 3D graded materials	1,874	146	99%
RPES	Rye Polynomial Equation Solver, quantum chem, 2-electron repulsion	1,104	281	99%
PNS	Petri Net simulation of a distributed system	322	160	>99%
SAXPY	Single-precision implementation of saxpy, used in Linpack's Gaussian elim. routine	952	31	>99%
TPACF	Two Point Angular Correlation Function	536	98	96%
FDTD	Finite-Difference Time Domain analysis of 2D electromagnetic wave propagation	1,365	93	16%
MRI-Q	Computing a matrix Q, a scanner's configuration in MRI reconstruction	490	33	>99%

Speedup of Applications



- **GeForce 8800 GTX vs. 2.2GHz Opteron 248**
- **At least 10x speedup in a kernel is typical, so long as the kernel can occupy enough parallel threads**
- **Application speedup depends on portion that can be run on GPU and other factors (like I/O, etc.)**

Outline of CUDA Basics

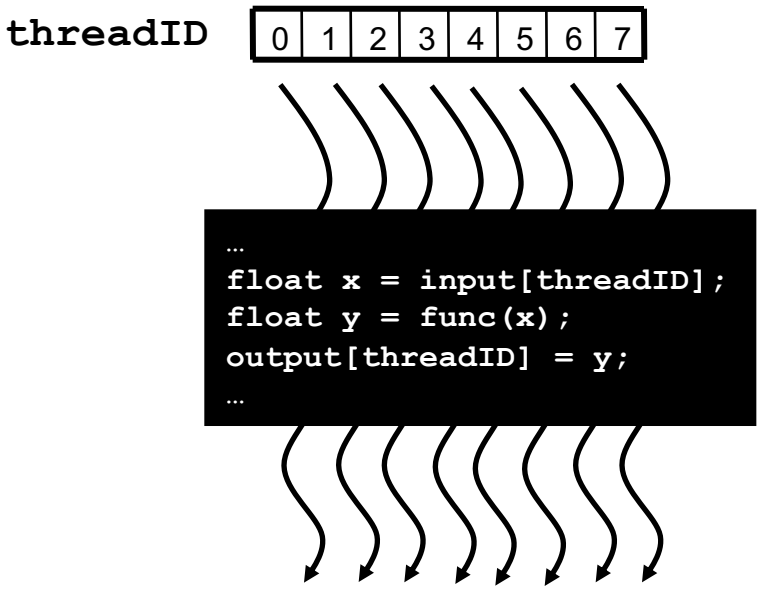
- Basic Kernels and Execution on GPU
- Basic Memory Management
- Coordinating CPU and GPU Execution

- See the Programming Guide (to be posted on Canvas) for the full CUDA API

Note: We will not address many of the latest features in CUDA. Our goal in this class is to provide a basic introduction and conceptual understanding of the GPU computational model.

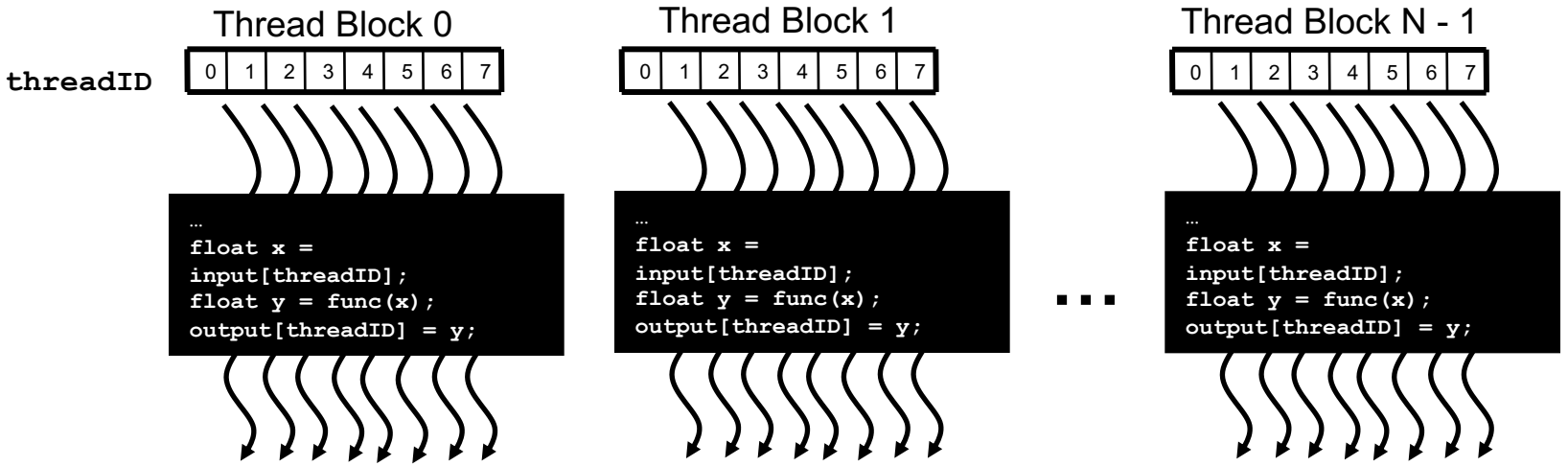
Simple Kernel

- A program uses GPU(s) by defining "kernels" that run on a GPU (similar to parallel regions)
- A CUDA kernel is executed by an array of threads
  - All threads run same code (Single Program, Multiple Threads)
  - Each thread has an ID that it uses to differentiate its activities, such as computing memory addresses, making control decisions, etc.



Threads, Blocks, Grids

- Threads in a kernel are arranged in a hierarchy
  - Individual threads are arranged in 3-D thread blocks of ≤1024 threads
    - All blocks in a kernel have same shape; max dims: 1024(x or y); 64 (z)
    - Threads within a single block cooperate/coordinate via shared memory, atomic operations, and barrier synchronization
  - Blocks are arranged into a 3-D grid (1 per kernel; ind. dims ≤ 65536)
    - Threads in different blocks can only cooperate via global memory
  - Each thread may compute its "thread ID" by using its position in its block, and its block's position in the grid

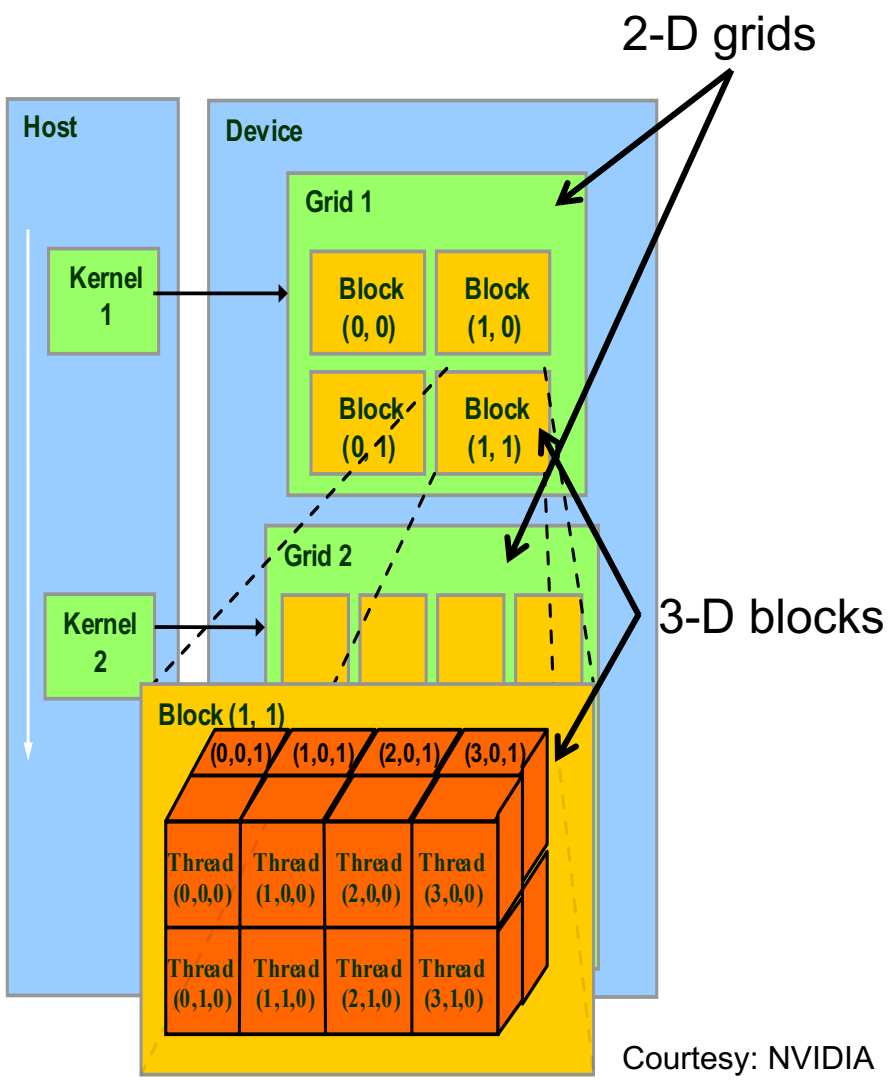


Block and Thread IDs

- Each thread uses positional indices to decide what to do:
  - Block Index: Position in the grid (up to 3-D)
  - Thread Index: Position in its block (up to 3-D)
- Mapping between problem and threads is based on indices:

```
id = threadIdx.x + blockDim.x * blockIdx.x;
```

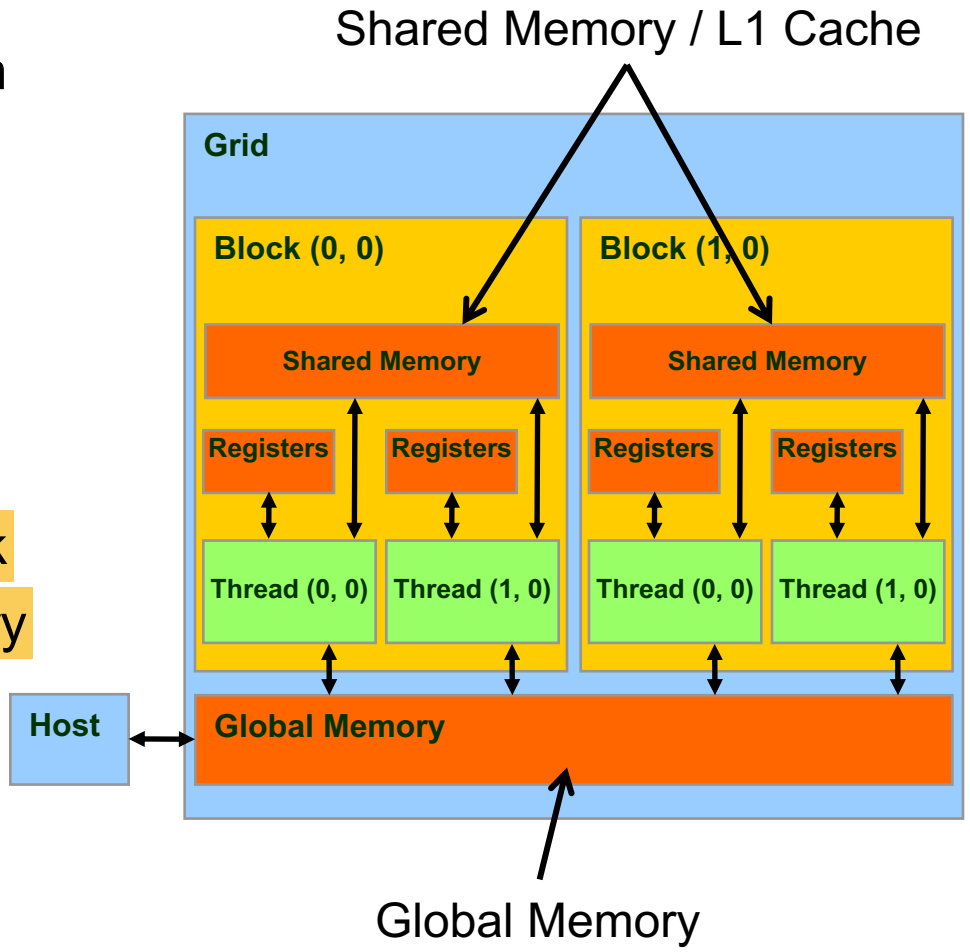
```
col = threadIdx.x + blockDim.x * blockIdx.x;
row = threadIdx.y + blockDim.y * blockIdx.y;
```



Courtesy: NVIDIA

CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W data between host & GPU
  - Visible to all kernels in program and all threads in a kernel
- Shared memory/L1 cache
  - 64 KB configurable
    - Max shared memory: 48 KB
    - Max L1 cache: 48 KB
  - Visible only within current block
  - Much faster than global memory
  - Local to each SM



Block-level Scalability

- Blocks must be independent of one another
  - Blocks are presumed to run to completion without pre-emption
  - Any possible interleaving of blocks must be valid
    - may run in any order
    - may run concurrently OR sequentially

- Blocks may coordinate, but not synchronize, with one another using the global memory. But be careful:
  - Shared queue pointer: OK
  - Shared lock: BAD … can easily deadlock or give poor performance

- Independence requirement gives “automatic scalability”

Code executed on GPU

- C/C++ with some restrictions:
  - Can only use GPU memory (note: newer GPUs have unified memory)
  - No variable number of arguments
  - Static variables must be “shared”
  - Limited recursion
  - No dynamic polymorphism

- Functions must be declared with a qualifier:
  - __global__ : launched by CPU,
    - May not be called from GPU; must return void
  - __device__ : called from other GPU functions,
    - May not be called on the CPU
  - __host__ : may be called/run only by CPU, not GPU
  - __host__ and __device__ qualifiers can be combined
    - Code is built twice: once for CPU, once for GPU

C for CUDA

Philosophy: Provide minimal set of extensions necessary to expose power, such as:

- Function qualifiers:

```
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

Called on CPU; executes on GPU; "main()"
Called on GPU; executes on GPU

- Variable qualifiers:

```
__shared__   float my_shared_array[32];
```

Lives in shared memory of thread block

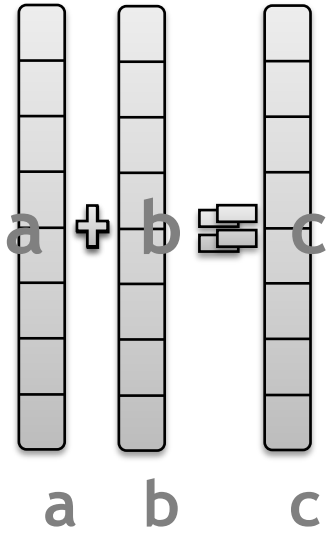
- Execution configuration:

```
dim3 grid_dim(100, 50, 1);  // grid with 5000 thread blocks
dim3 block_dim(4, 8, 8);    // 256 threads per block
my_kernel <<< grid_dim, block_dim >>> (...); // Launch kernel
    (for 1-D blocks/grids, args may be scalar instead of dim3 type)
```

- Built-in variables and functions valid in device code (referenced via ".x", ".y", ".z"):

```
dim3 gridDim;   // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index
dim3 threadIdx; // Thread index
void __syncthreads(); // Thread barrier synchronization
```

Vector Addition in CUDA



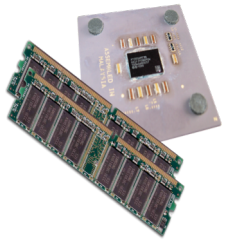
- Start by adding two integers and build up to vector addition
- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {
                    *c = *a + *b;
}
```

- Key syntax/semantics:
  - `__global__` means `add()` is called on host, executed on GPU
  - Arguments are pointers (To what?)
  - Need to address memory allocation

Memory Management (Pre-Unified Memory)

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code



- Simple CUDA API for handling device memory
  - **`cudaMalloc()`**, **`cudaFree()`**, **`cudaMemset()`, `cudaMemcpy()`**
  - Similar to the C equivalents **`malloc`**, **`free`**, **`memset`, `memcpy`**
- Note: For performance, may want to use page-locked (aka "pinned") memory (advanced topic)

Photos ©Nvidia 2011

GPU Memory Allocation / Release

- Host (CPU) manages device (GPU) memory:
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)

```
int n = 1024;
size_t nbytes = 1024*sizeof(int);
int *d_a = 0;

cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Simple Data Copies

- `cudaMemcpy( void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn't start copying until previous CUDA calls complete
- `enum cudaMemcpyKind`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`
- Non-blocking copies are also available
  - Require use of page-locked memory and streams

Scalar Addition on the GPU: `main()`

```
int main(void) {
        int a, b, c;                    // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        size_t size = sizeof(int);

// Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

// Setup input values
        a = 2;
        b = 7;
```

Scalar Addition on GPU: `main()` (cont)

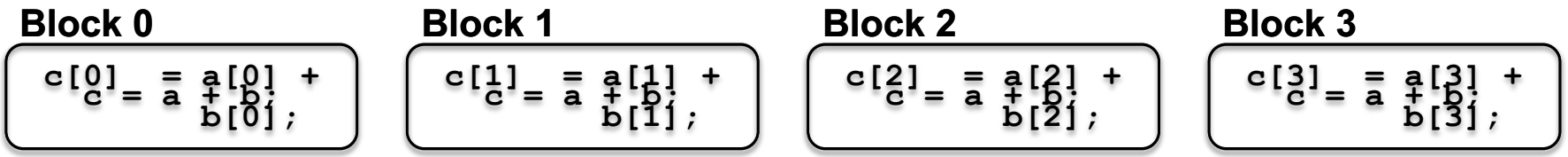
```
// Copy inputs to device
        cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
        add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
        cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Moving to Parallel Vector Addition

- Step 1: Run code in parallel on the GPU:
  - Execute **`add()`** N times in parallel
    - Instead of: **`add<<< 1, 1 >>>(d_a, d_b, d_c);`**
    - Use: **`add<<< N, 1 >>>(d_a, d_b, d_c);`**
    - This runs a 1-D grid containing **N blocks**, with **1 thread per block**
- Step 2: Get threads to add different vector elements, not scalars:
  - Do this by indexing the vectors using the block index **`blockIdx.x`**

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

  - For **`N=4`**:



Vector Addition on the Device: main()

```
#define N 512
int main(void) {
        int *a, *b, *c;             // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b, c
        size_t size = N * sizeof(int);

// Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

// Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

Vector Addition on the Device: `main()` (cont)

```
// Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks with 1 thread each
        add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Alternative Parallel Vector Addition

- So far: multiple blocks with 1 thread per block
- Could use 1 block with multiple threads (up to 1024 threads):

```
__global__ void add(int *a, int *b, int *c) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

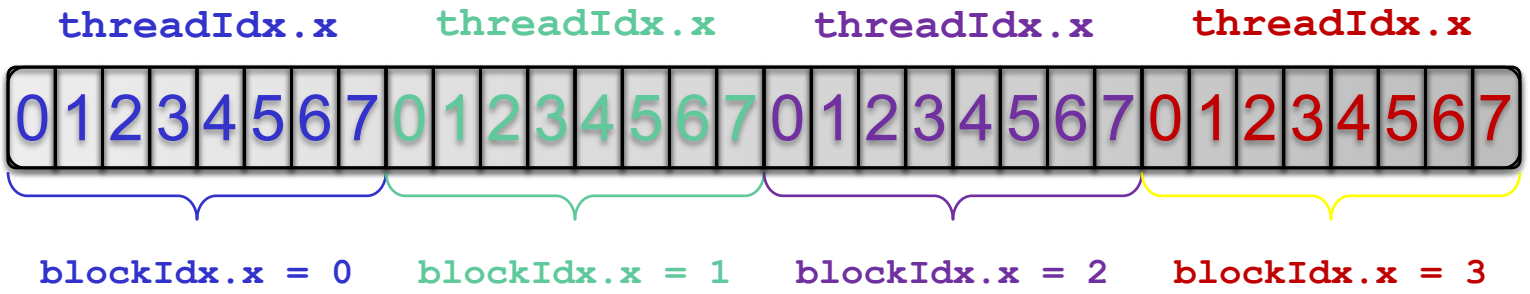
- Only change to **`main()`** is the kernel launch:

```
// Launch add() kernel on GPU with 1 block containing N threads
    add<<<1,N>>>(d_a, d_b, d_c);
```

- Even better: multiple blocks with multiple threads in each
  - Need to map block and thread index values to vector subscripts

Indexing Arrays with Both Block and Thread Indices

- Not as simple as using **`blockIdx.x`** or **`threadIdx.x`** alone
  - Consider assigning 1 array element per thread
  - Using 8 threads per block and 4 blocks gives:

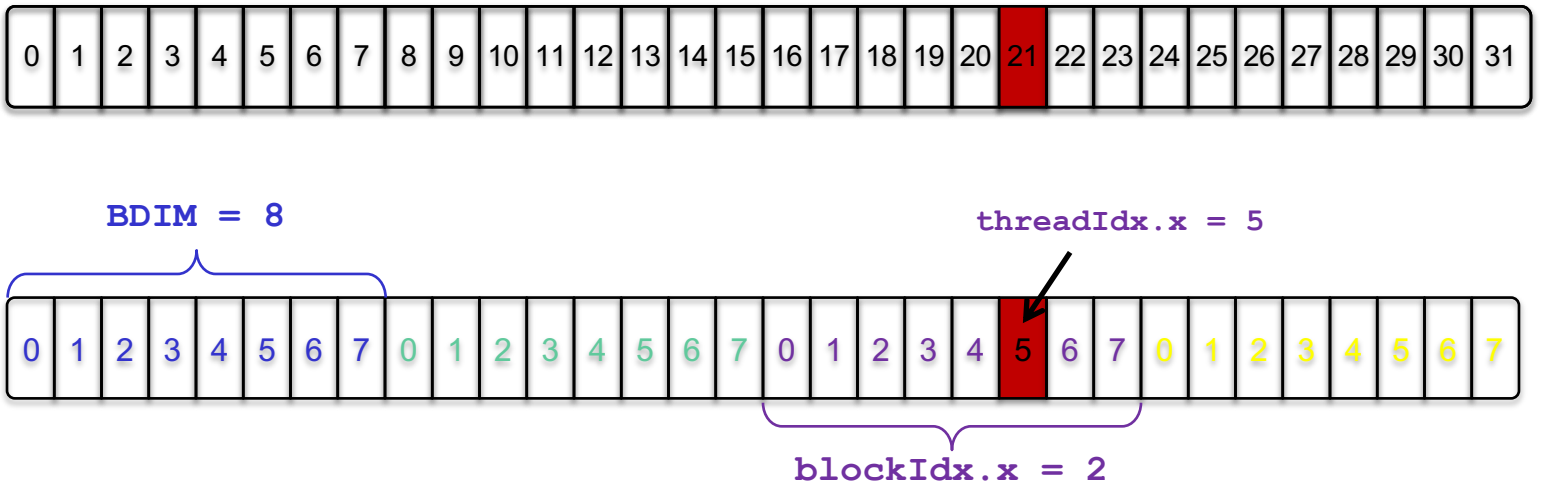


- With BDIM threads/block, the array element assigned to a thread is:

```
int index = threadIdx.x + blockIdx.x * BDIM;
```

Indexing 1-D Arrays: Example (8 threads/block)

- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * BDIM;
          =       5       +      2       *  8
          = 21
```

Vector Addition with Blocks and Threads

- Use the built-in variable **`blockDim.x`** (**`== BDIM`** at runtime)

```
int index = threadIdx.x + blockDim.x * blockIdx.x
```

- Version of **`add()`** using parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
        int index = threadIdx.x + blockDim.x * blockIdx.x;
        c[index] = a[index] + b[index];
}
```

- Only changes in **`main()`** relate to kernel launch

Vector Addition on the Device: main()

```
#define N (2048*2048)
#define BDIM 512
int main(void) {
        int *a, *b, *c;             // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b, c
        size_t size = N * sizeof(int);

// Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

// Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
// Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N/BDIM blocks
        add<<<N/BDIM,BDIM>>>(d_a, d_b, d_c);

// Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Handling Arbitrary Vector Sizes

- In many problems, N is not an integer multiple of `blockDim.x`
- Must avoid accessing beyond the end of the arrays:

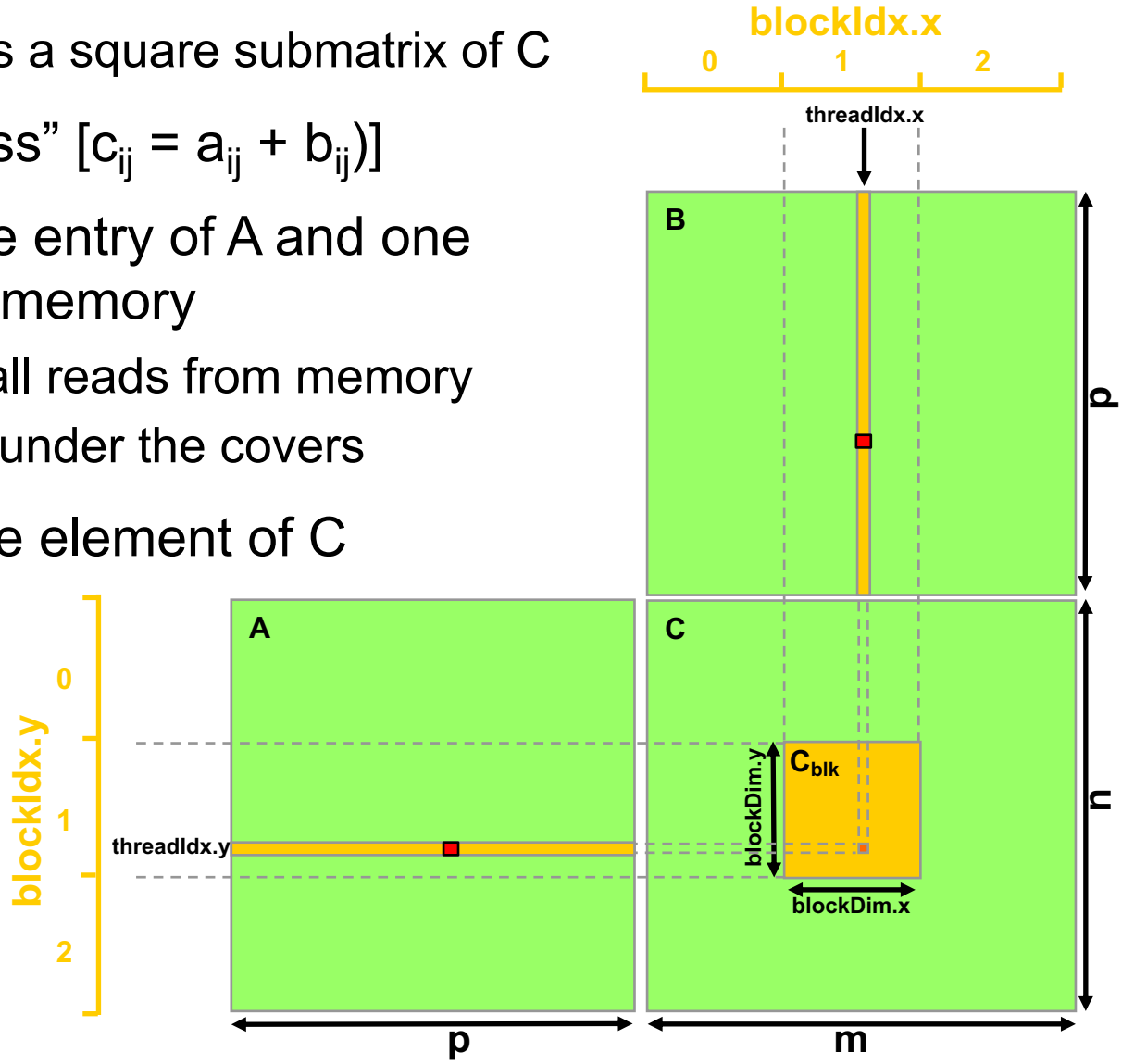
```
__global__ void add(int *a, int *b, int *c, int n) {
        int index = threadIdx.x + blockDim.x * blockIdx.x;
        if (index < n) c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N+BDIM-1)/BDIM,BDIM>>>(d_a, d_b, d_c, N);
```

Moving from Vectors to Matrices - Concepts

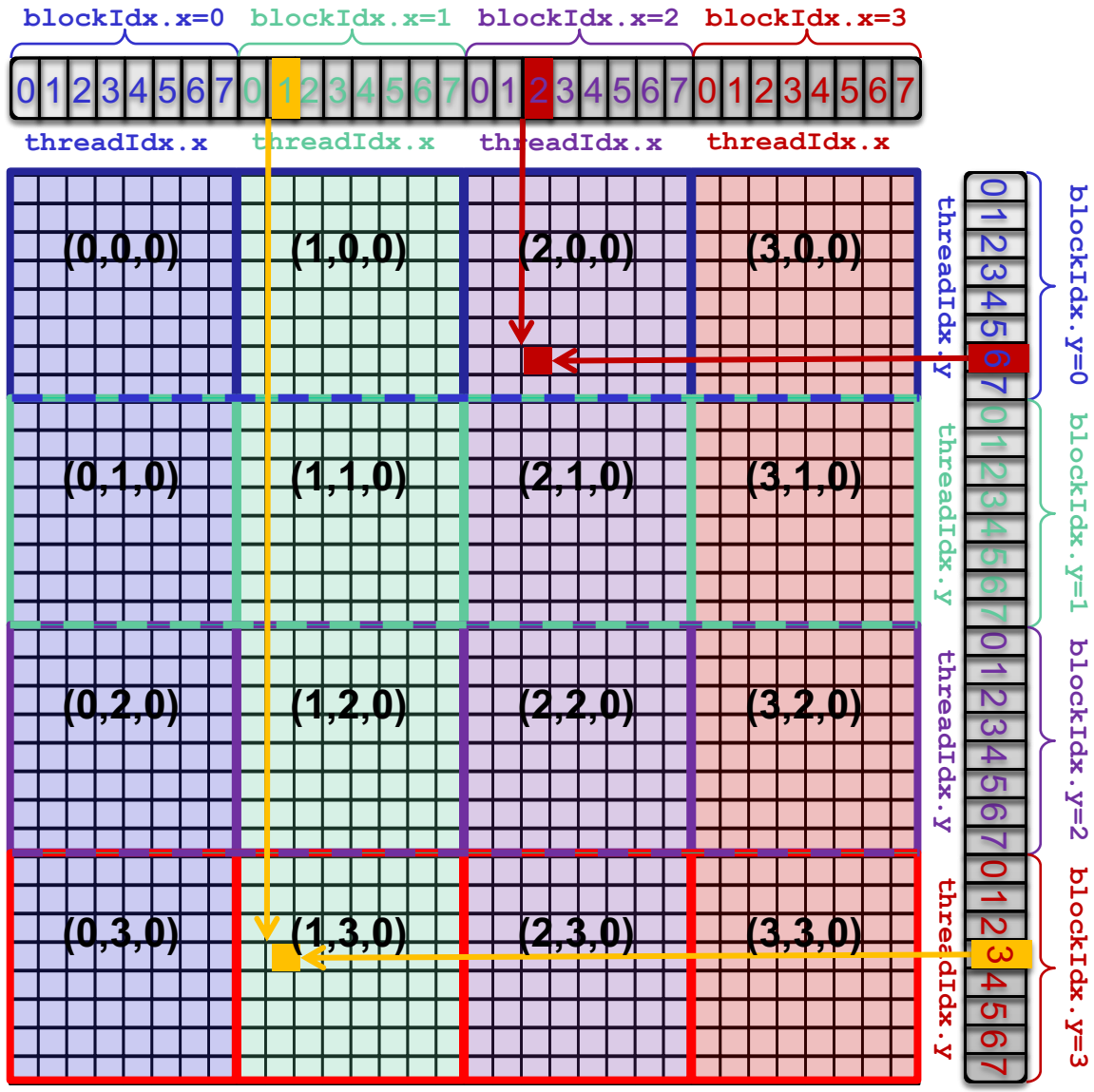
- Each thread computes one element of C
  - Each block computes a square submatrix of C
- "Scalar thought process" [$c_{ij} = a_{ij} + b_{ij}$)]
- Each thread reads one entry of A and one entry of B from global memory
  - Inefficient: many small reads from memory
  - Cache may be used under the covers
- Each thread writes one element of C



2-D Indexing for Matrices (4x4 grid, 8x8 blocks)

col = threadIdx.x +
blockDim.x*blockIdx.x

row = threadIdx.y +
blockDim.y*blockIdx.y



Matrix Addition with Blocks and Threads

- Use the built-in variables **`blockDim.x`** and **`blockDim.y`**

```
int col = threadIdx.x + blockDim.x * blockIdx.x
int row = threadIdx.y + blockDim.y * blockIdx.y
```

- Version of **`add()`** using 2-D grid *and* 2-D blocks

```
__global__ void add(int *a, int *b, int *c, int n) {
        int col = threadIdx.x + blockDim.x * blockIdx.x;
        int row = threadIdx.y + blockDim.y * blockIdx.y;
        int index = row * n + col;
        if (col<n && row<n) c[index] = a[index] + b[index];
}
```

- Only modest changes need to be made in **`main()`**
- Note use of 1-D array indexing for multidimensional array

Matrix Addition on the GPU: main()

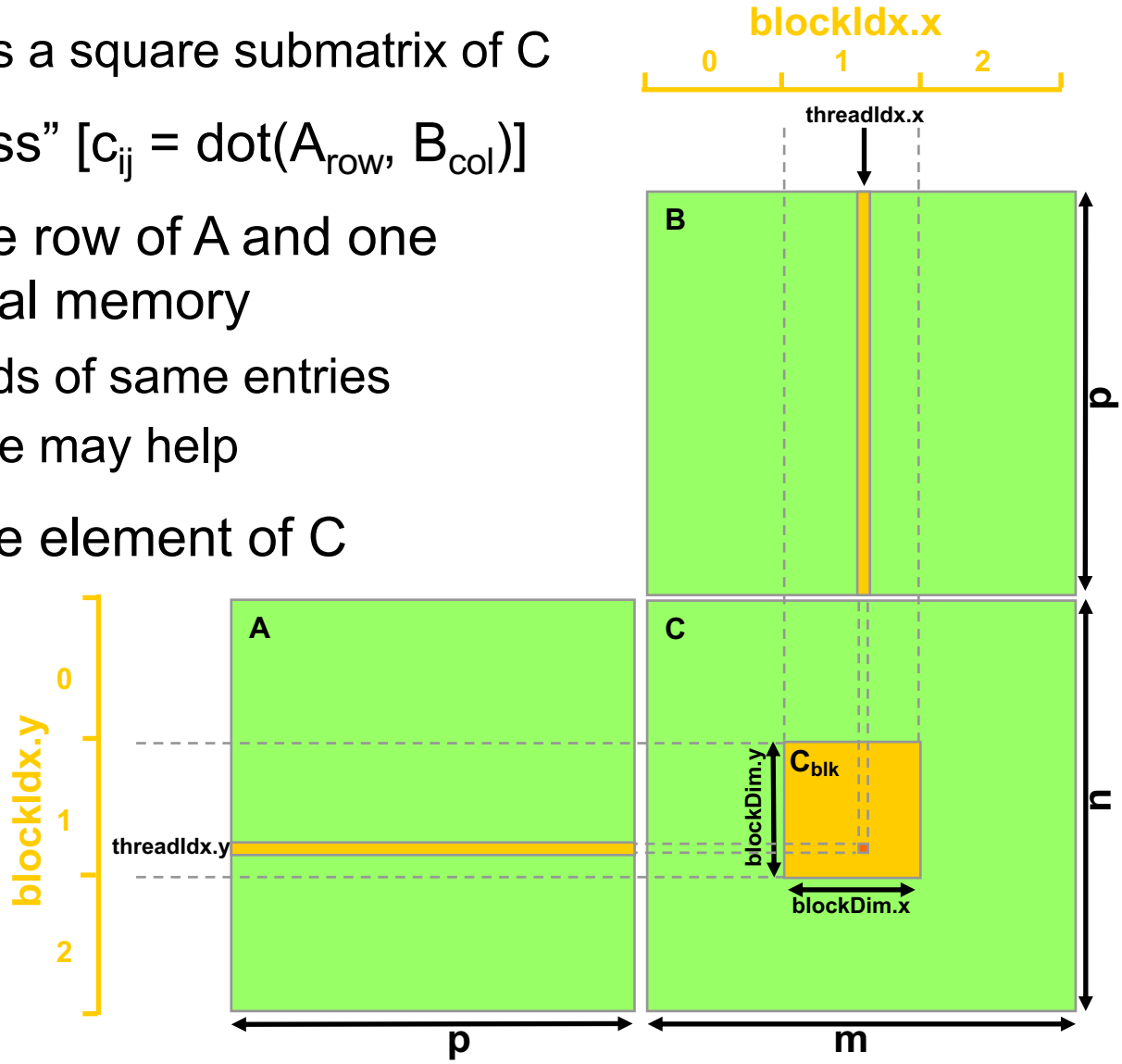
```
#define N 2048
#define BDIM 32
int main(void) {
        int *a, *b, *c;             // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b, c
        size_t size = N * N * sizeof(int); // N x N matrix

// Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

// Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N*N);
        b = (int *)malloc(size); random_ints(b, N*N);
        c = (int *)malloc(size);
```

Matrix Addition on the GPU: main()

```
// Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with enough threads
// (This assumes that BDIM divides N evenly)
        dim3 Block(BDIM, BDIM); // BDIM x BDIM thread blocks;
        dim3 Grid(N/BDIM, N/BDIM); // N threads in x & y directions;
        add<<<Grid,Block>>>(d_a, d_b, d_c, N);

// Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

From Addition to Naïve Matrix Multiplication – Concepts

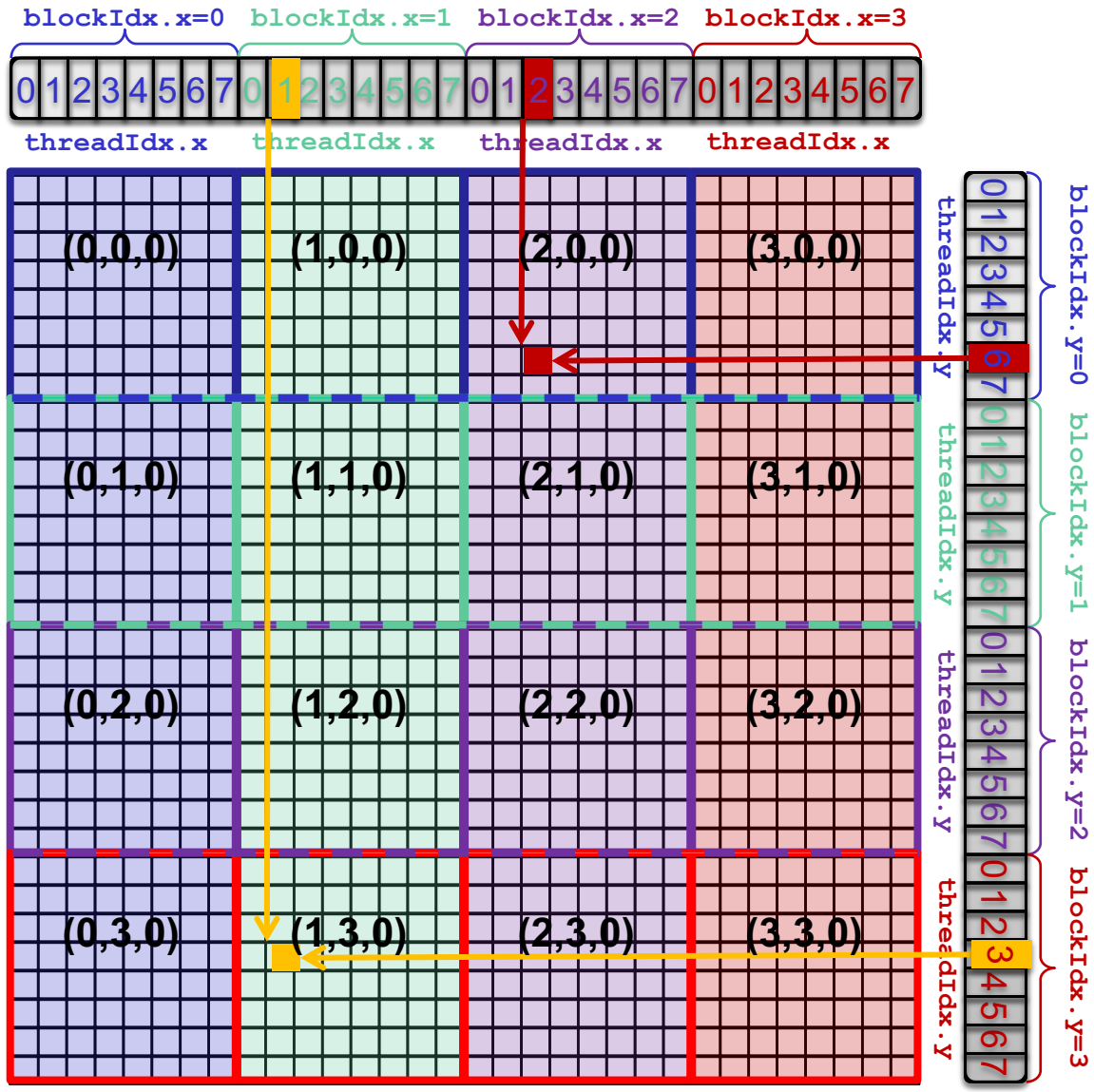
- Each thread computes one element of C
  - Each block computes a square submatrix of C
- "Vector thought process" [c_{ij} = dot(A_{row}, B_{col})]
- Each thread reads one row of A and one column of B from global memory
  - Inefficient: many reads of same entries
  - GM is slow, but cache may help
- Each thread writes one element of C



2-D Indexing for Matrices (4x4 grid, 8x8 blocks)

col = threadIdx.x +
blockDim.x*blockIdx.x

row = threadIdx.y +
blockDim.y*blockIdx.y



Moving from Matrix Addition to Matrix-Matrix Product

- All we really have to do is replace "+" with a dot product:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    int index = row * n + col;
    int cvalue = 0;

    if (col<n && row<n)
        for (int k=0; k<n; k++) cvalue += a[row*n+k] * b[k*n+col];
    c[index] = cvalue;
}
```

- No changes needed in the main program!

Naïve Matrix Multiplication Kernel – Code

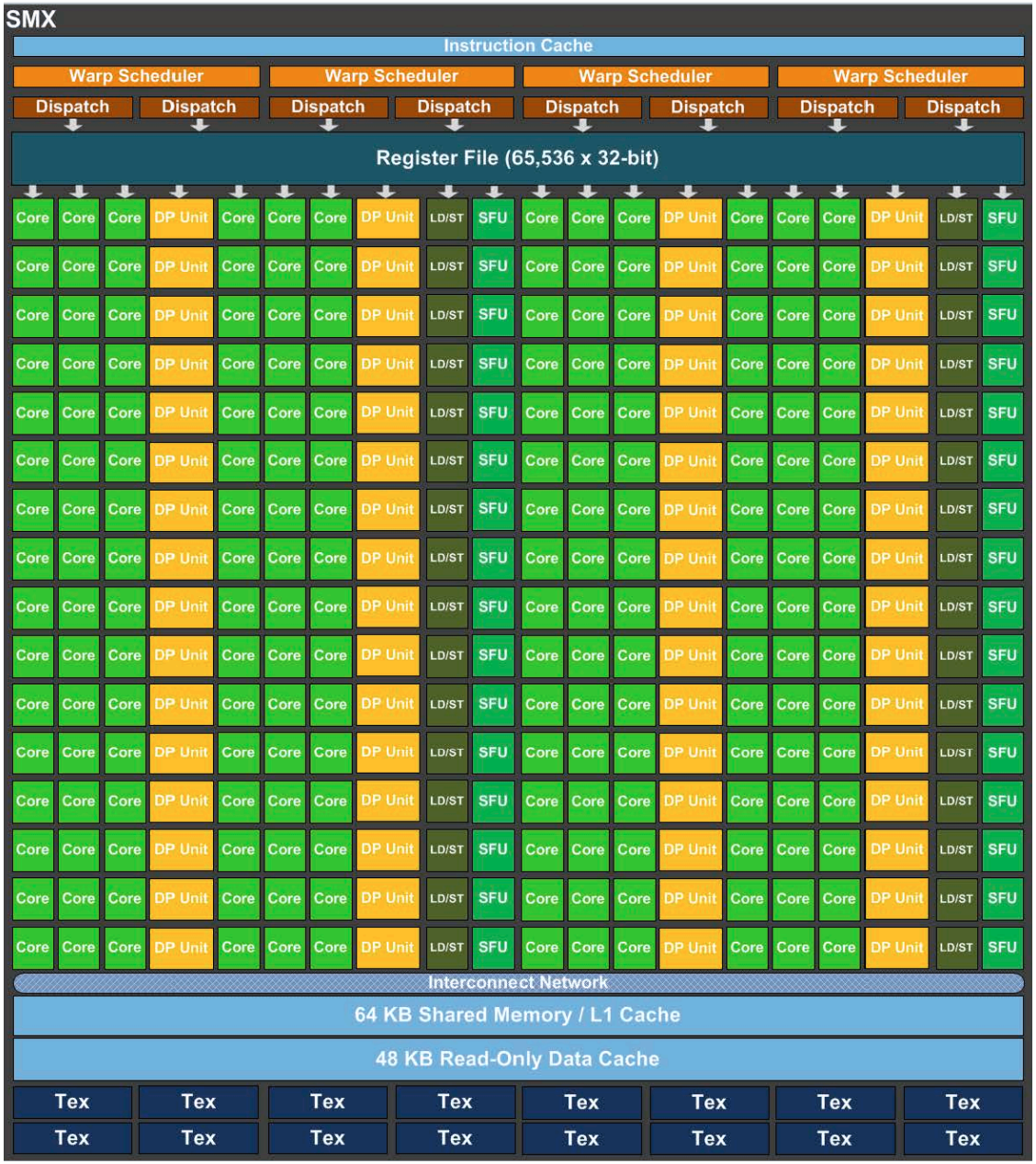
```
__global__ void matmul(int *a, int *b, int *c, int n) {

   int tx = threadIdx.x; int ty = threadIdx.y;
   int col = tx + blockDim.x * blockIdx.x;
   int row = ty + blockDim.y * blockIdx.y;
   int index = row*n + col;
   int cvalue = 0;

   if (row<n && col<n)
      for (int k=0; k<n; k++) cvalue += a[row*n+k] * b[k*n+col];

   c[index] = cvalue;
}
```

Topics

- Thread Scheduling & Divergence
- GPU Memory Hierarchy
- Improving matrix multiplication performance
  - Loop unrolling
  - Multi-tiled kernel
- Memory optimizations
  - Global memory
  - Shared memory
- More general performance optimization process

NVIDIA Kepler GPU Architecture

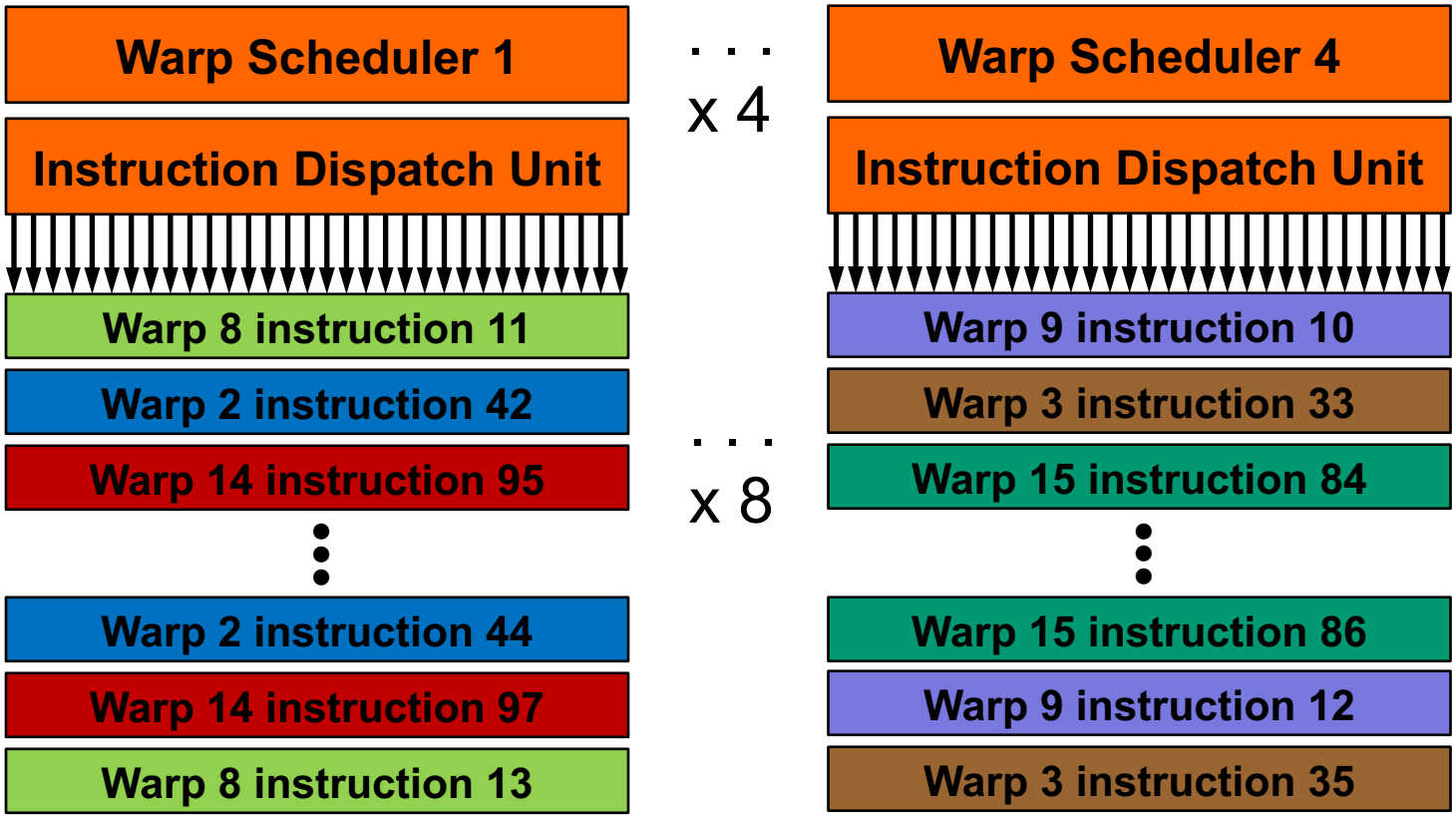
Kepler SMX Multiprocessor



Kernel Execution

- Thread blocks are assigned to SMs
  - Done at run-time, so don't assume any particular order
  - Once assigned, a TB stays resident until all its threads complete
    - Not migrated to another SM; Not swapped out for another TB
- Instructions are issued/executed per warp
  - Warp = 32 consecutive threads
    - Think of it as a "vector" of 32 threads. (How are threads numbered?)
    - The same instruction is issued to the entire warp. In older GPUS, this may take 2 cycles, with instructions issued to 16 threads each cycle
- Scheduling within a single SM
  - Warps are scheduled at run-time (details depend on hardware)
    - Except for DP operations, most instructions can be "multi-issued" up to about 7-8 warps per cycle (max of 64 warps are resident per SMX)
  - Hardware picks from warps that have an instruction ready to execute
  - Instruction/memory latency is hidden by executing other warps

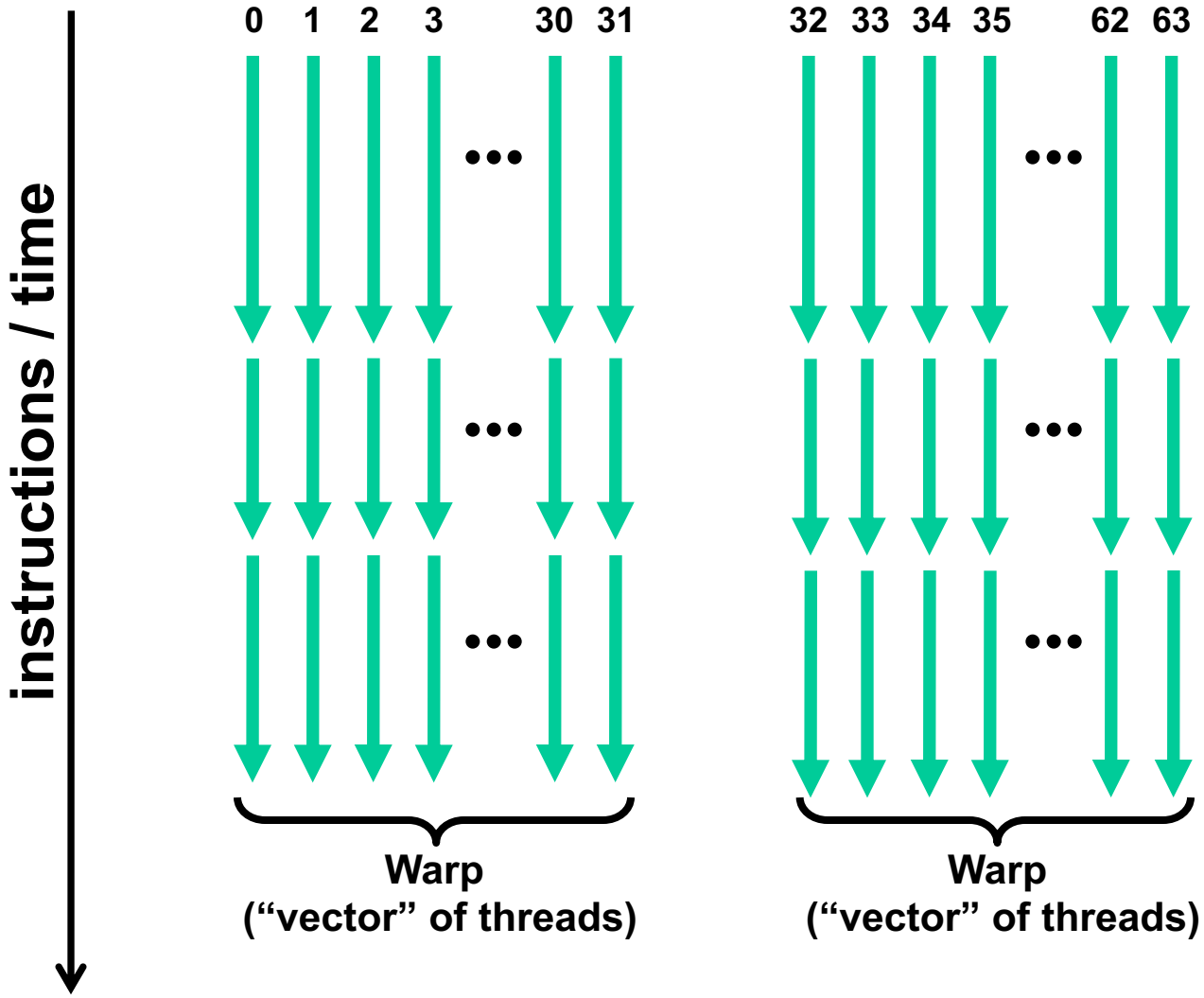
Instruction Scheduling



Each scheduler can issue 2 instructions from 1 warp at a time. However, instructions from different warps may not complete in the order they are issued.

- 4 Warp Schedulers per SMX; each can issue 2 instructions at a time
- Up to 64 warps "in flight" at once on each SMX on K40 or K80 GPU
- Instructions (which may take multiple cycles) are internally pipelined

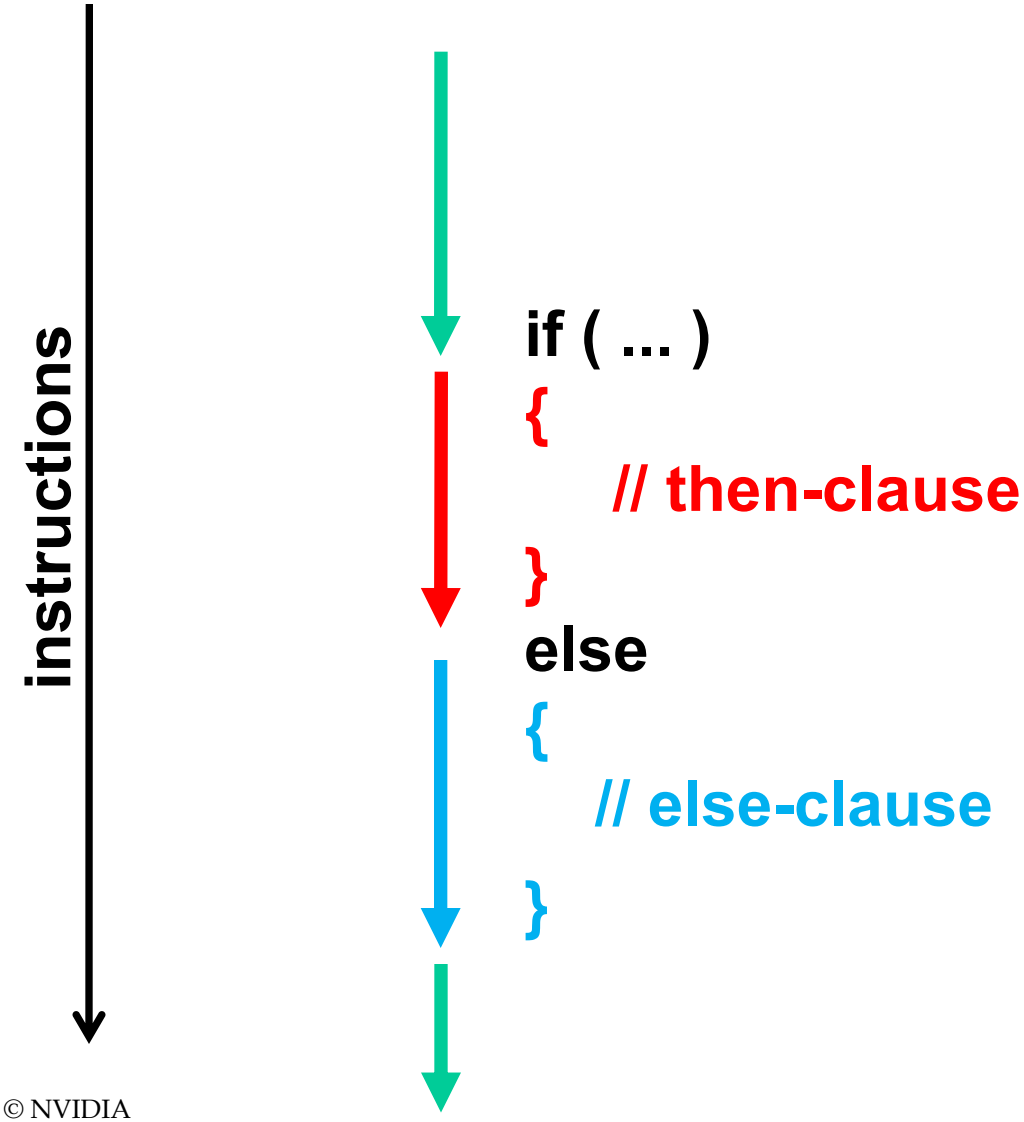
Execution scheduled by warps



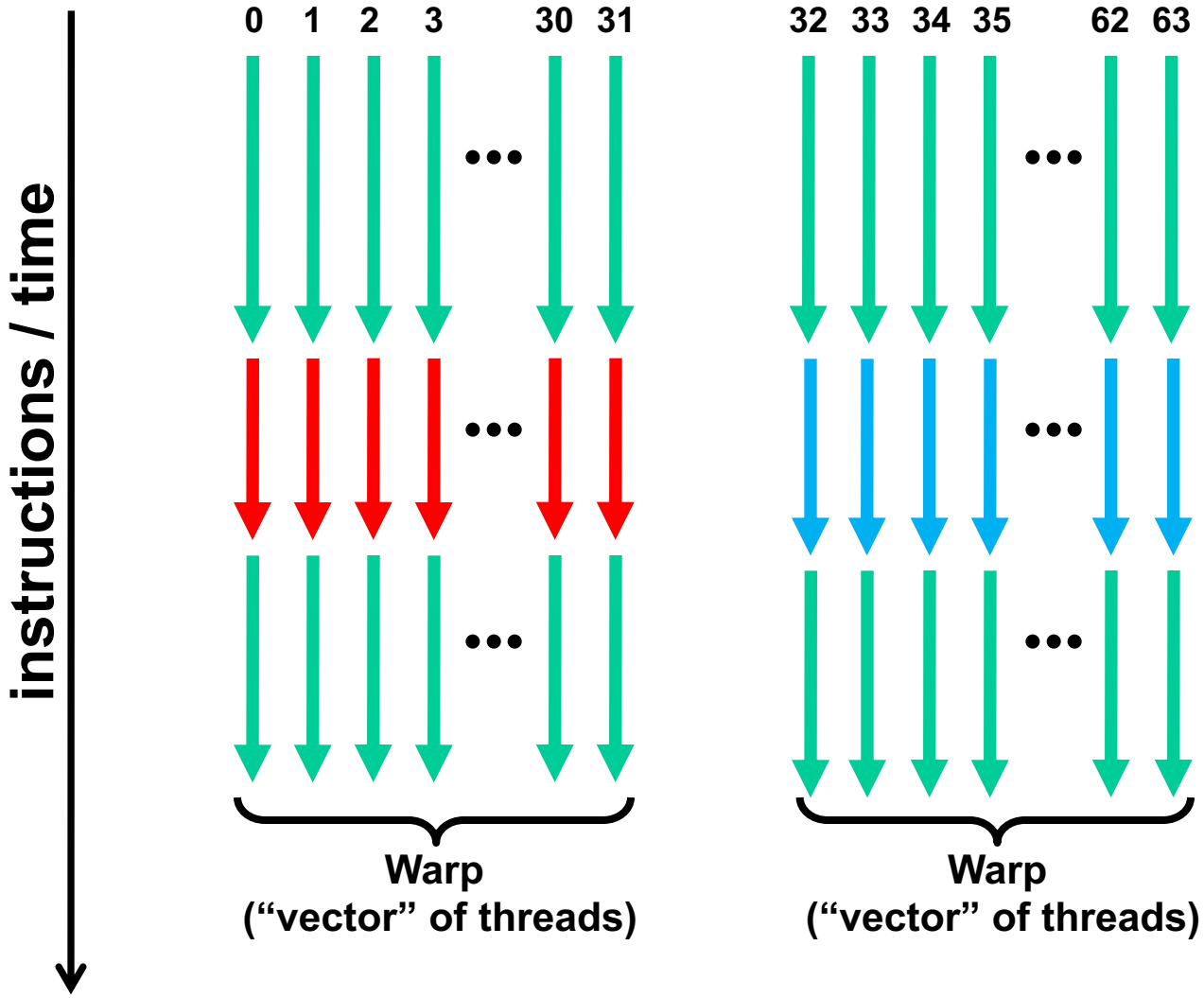
Control Flow

- "Divergence":
  - Threads within a single warp take different paths
    - if-else, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code w/o performance impact
- But… Avoid divergence within a warp:
  - Example with divergence:
    - `if (threadIdx.x > 2) {...} else {...}`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) {...} else {...}`
    - Branch granularity is a whole multiple of warp size

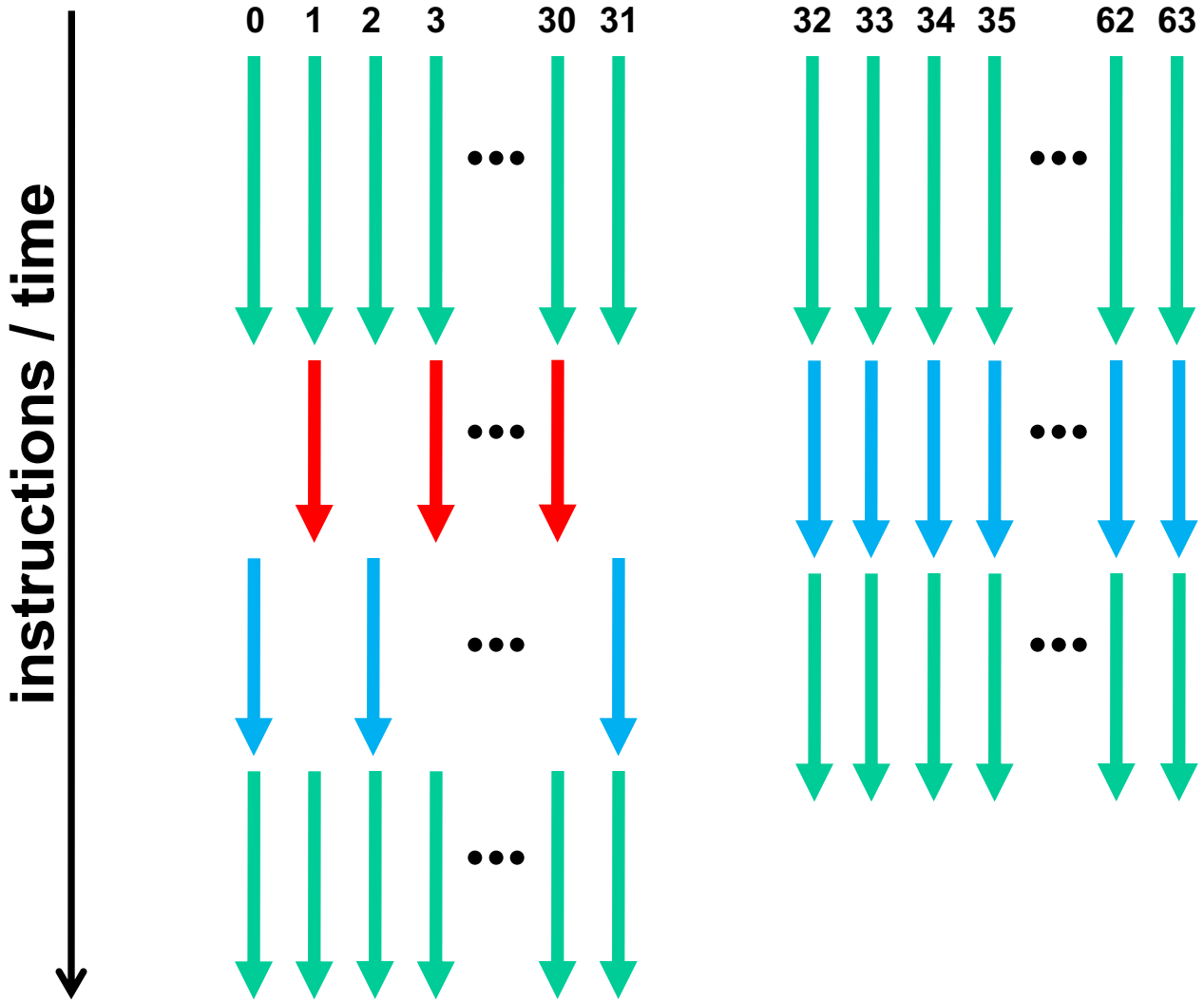
Control Flow



Execution with entire warps taking same path



Execution with divergence within a warp



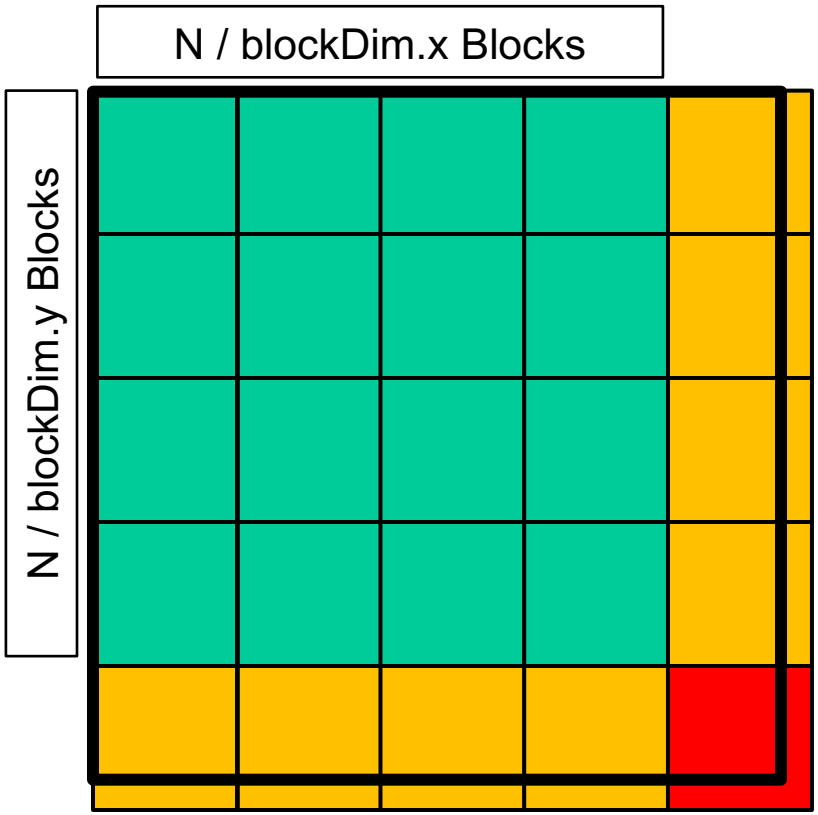
Divergence in the Naïve Matrix Multiplication Kernel

```
__global__ void matmul(int *a, int *b, int *c, int n) {

    int tx = threadIdx.x; int ty = threadIdx.y;
    int col = tx + blockDim.x * blockIdx.x;
    int row = ty + blockDim.y * blockIdx.y;
    int index = row*n + col
    int cvalue = 0;

    if (row<n && col<n)
        for (int k=0; k<n; k++) cvalue += a[row*n+k] * b[k*n+col];

    c[row*n + col] = cvalue;
}
```

Program logic, where divergence within warps could possibly occur

Divergence for Matrix Multiplication Example

- Only real source of divergence for this kernel: test of row and col
- "Good" blocks = Those entirely within the result matrix
  - Most others only have divergence in 1 dimension, which may be a non-issue (Why?) or fixable in some cases.



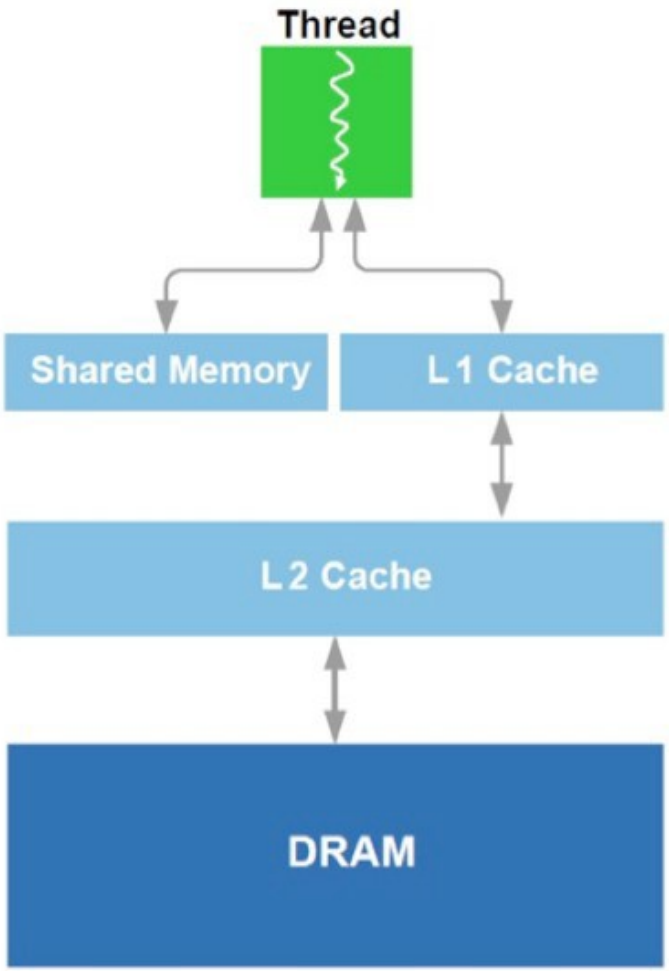
Topics

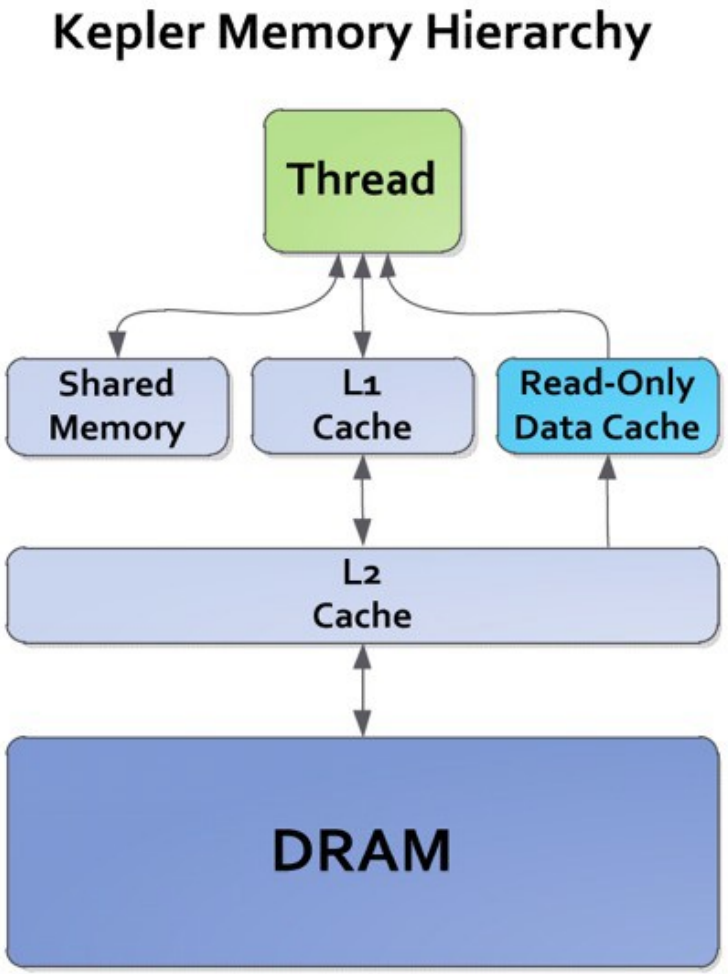
- Thread Scheduling & Divergence
- GPU Memory Hierarchy
- Improving matrix multiplication performance
  - Loop unrolling
  - Multi-tiled kernel
- Memory optimizations
  - Global memory
  - Shared memory
- More general performance optimization process

The memory in Tesla cards: Fermi vs. Kepler

Tesla card	M2075	M2090	K20	K20X	K40
32-bit register file / multiprocessor	32768	32768	65536	65536	65536
L1 cache + shared memory size	64 KB.	64 KB.	64 KB.	64 KB.	64 KB.
Width of 32 shared memory banks	32 bits	32 bits	64 bits	64 bits	64 bits
SRAM clock freq. (same as GPU)	575 MHz	650 MHz	706 MHz	732 MHz	745,810,875 MHz
L1 and shared memory bandwidth	73.6 GB/s.	83.2 GB/s.	180.7 GB/s	187.3 GB/s	216.2 GB/s.
L2 cache size	768 KB.	768 KB.	1.25 MB.	1.5 MB	1.5 MB
L2 cache bandwidth (bytes/cycle)	384	384	1024	1024	1024
L2 on atomic ops. (shared address)	1/9 per clk	1/9 per clk	1 per clk	1 per clk	1 per clk
L2 on atomic ops. (indep. address)	24 per clk	24 per clk	64 per clk	64 per clk	64 per clk

Differences in memory hierarchy: Fermi vs. Kepler





Kepler Memory Hierarchy

- Local storage
  - Thread-private memory, mainly registers (64K x 32-bit / blk on K80)
    Arrays/data not in regs are in “local memory” (priv. part of global mem)
- Shared memory / L1 (128KB per SMX on K80)
  - Same physical HW with very low latency & very high throughput
    - ~216 GB/s per SMX (2.8 TB/s aggregate) on K40
  - Configurable: 16/32/48KB Smem per block; Remainder is L1 cache
  - Shared mem accessible only by threads in the same thread block
- L2 (1.5 MB on K40)
  - Used for all global memory accesses (even copies to/from CPU host)
- Global memory (~12 GB on K40 or K80 GPUs)
  - Accessible by all threads as well as host (CPU)
  - High latency and modest throughput
    - ~288 GB/s bandwidth on K40

CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic scalar variables (not arrays)	Register	Thread	Kernel
Automatic array variables	“Local” (Private Global)	Thread	Kernel
[__device__] __shared__ …	Shared	Block	Kernel
__device__	Global	Grid	Application
[__device__] __constant__ …	Constant (WORM Global)	Grid	Application

Programming for L1 and L2 Caches

- Short answer: DON'T
  - GPU caches are not intended for the same use as CPU caches
    - Smaller size (especially per thread), so not aimed at temporal reuse
    - Intended to smooth out certain access patterns, help with spilled registers, etc.
  - Don't try to block for L1/L2 like you would on CPU
    - 100s to 1,000s of run-time scheduled threads will hit the caches
    - If it is possible to block for L1 then block for SM instead
      - Same size, same bandwidth, hw will not evict behind your back
- Bottom Line: Optimize as if no caches were there

Topics

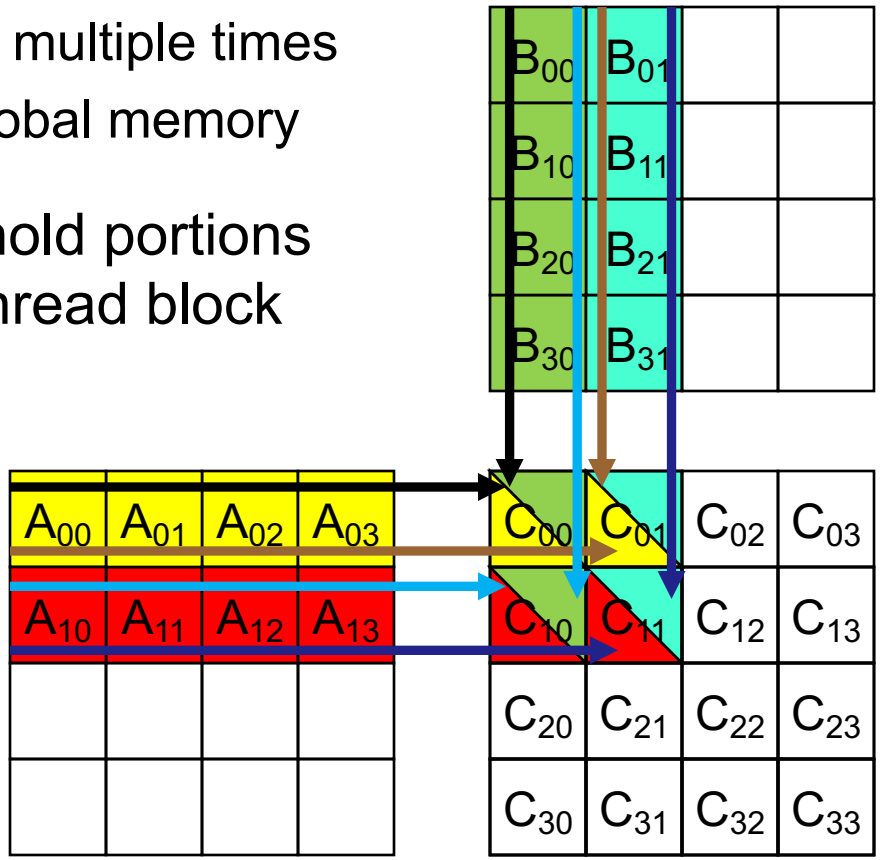
- Thread Scheduling & Divergence
- GPU Memory Hierarchy
- Improving matrix multiplication performance
  - Multi-tiled kernel
  - Loop unrolling
- Memory optimizations
  - Global memory
  - Shared memory
- More general performance optimization process

Naïve Matrix Multiplication Kernel – Code

```
__global__ void matmul(int *a, int *b, int *c, int n) {

   int tx = threadIdx.x; int ty = threadIdx.y;
   int col = tx + blockDim.x * blockIdx.x;
   int row = ty + blockDim.y * blockIdx.y;
   int index = row*n + col;
   int cvalue = 0;

   if (row<n && col<n)
      for (int k=0; k<n; k++) cvalue += a[row*n+k] * b[k*n+col];

   c[index] = cvalue;
}
```

Memory Usage in the Naïve Matrix Multiplication Kernel

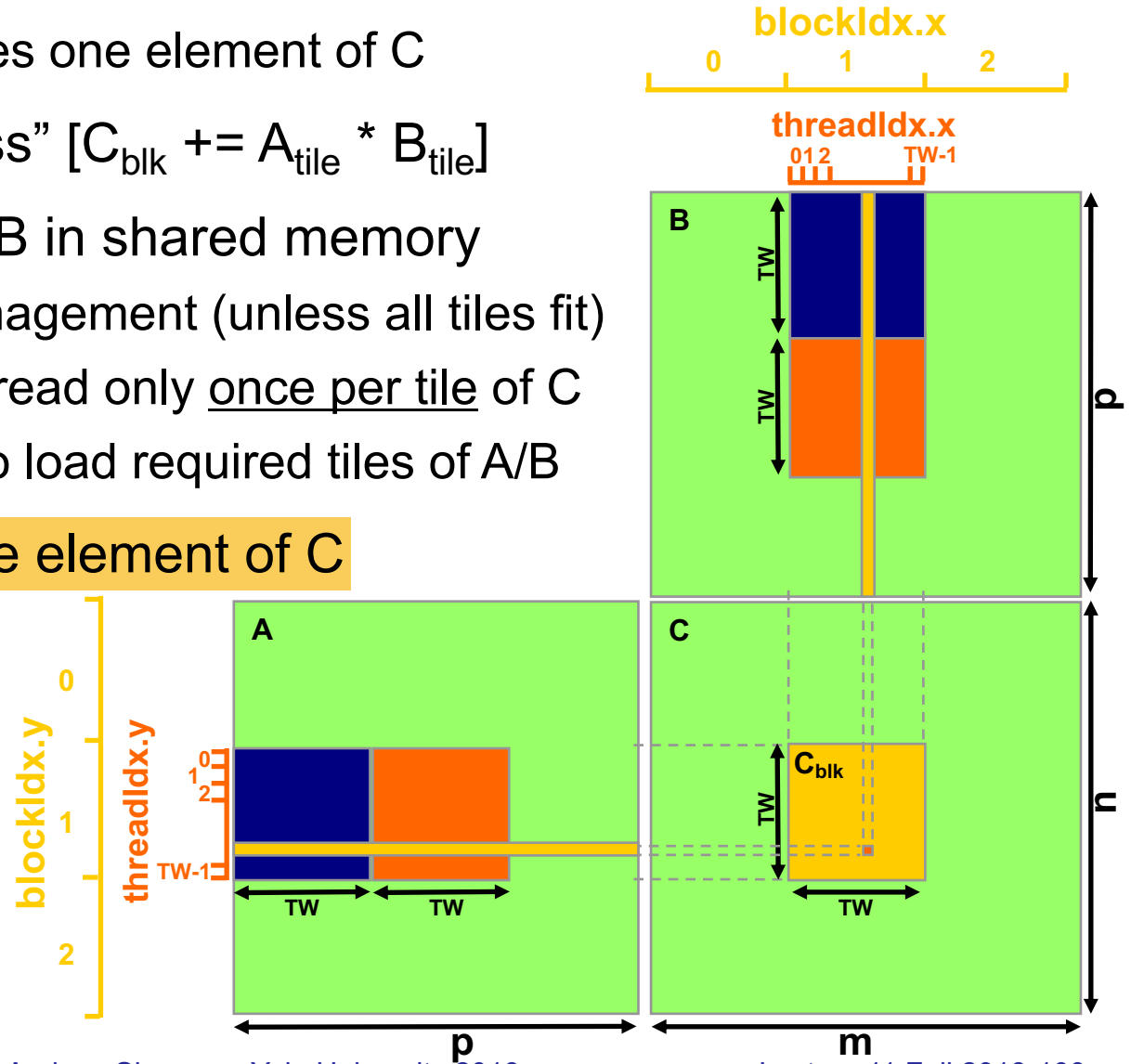
- Computing an entry of C requires reading an entire row of A and an entire column of B
  - Each entry of A and B is used multiple times
  - Reads are very costly from global memory
- Idea: Use shared memory to hold portions of A and B required for each thread block



Simple case with 4x4 thread blocks

Tiled Matrix Multiplication Kernel – Concepts

- Blocks compute TW x TW "tiles" of C
  - Each thread computes one element of C
- "Matrix thought process" [C_{blk} += A_{tile} * B_{tile}]
- Cache all entries of A/B in shared memory
  - Manual memory management (unless all tiles fit)
  - Elements of A/B are read only once per tile of C
  - Threads cooperate to load required tiles of A/B
- Each thread writes one element of C



Simplest case:

TW = blockDim.x = blockDim.y

Tiled Matrix Multiplication Kernel – Code

```
__global__ void matmul_tiled(int *a, int *b, int *c, int n) {

   __shared__ int atile[TW][TW], btile[TW][TW];
   int tx = threadIdx.x; int ty = threadIdx.y; int cvalue = 0;
   int col = tx + blockDim.x * blockIdx.x;
   int row = ty + blockDim.y * blockIdx.y;

// Loop over tiles (assumes TW divides n)
   for (int m=0; m<n/TW; m++) {
      atile[ty][tx] = a[row*n + m*TW + tx]; //Copy to shared memory
      btile[ty][tx] = b[(m*TW+ty)*n + col]; //Copy to shared memory
      __syncthreads();

      for (int k=0; k<TW; k++) cvalue += atile[ty][k] * btile[k][tx];
      __syncthreads();
   }

   c[row*n + col] = cvalue;
}
```

Tiled Matrix Multiplication Kernel – Execution Configuration

- Each block handles 1 tile:
  - `dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);`
- Grid is sized to allow each thread to handle 1 element of C:
  - `dim3 dimGrid(N/TILE_WIDTH, N/TILE_WIDTH);`
  - Note: This presumes `N` is divisible by `TILE_WIDTH`
- Can use the default setting for shared memory (usually 48KB)
- Better to set it using runtime calls (see next slide). Often, you'll want to use the following before kernel launch:

```
cudaFuncSetCacheConfig(matmul, cudaFuncCachePreferShared)
```

which gives 48 KB of shared memory & 80 KB of L1 cache (on K80)

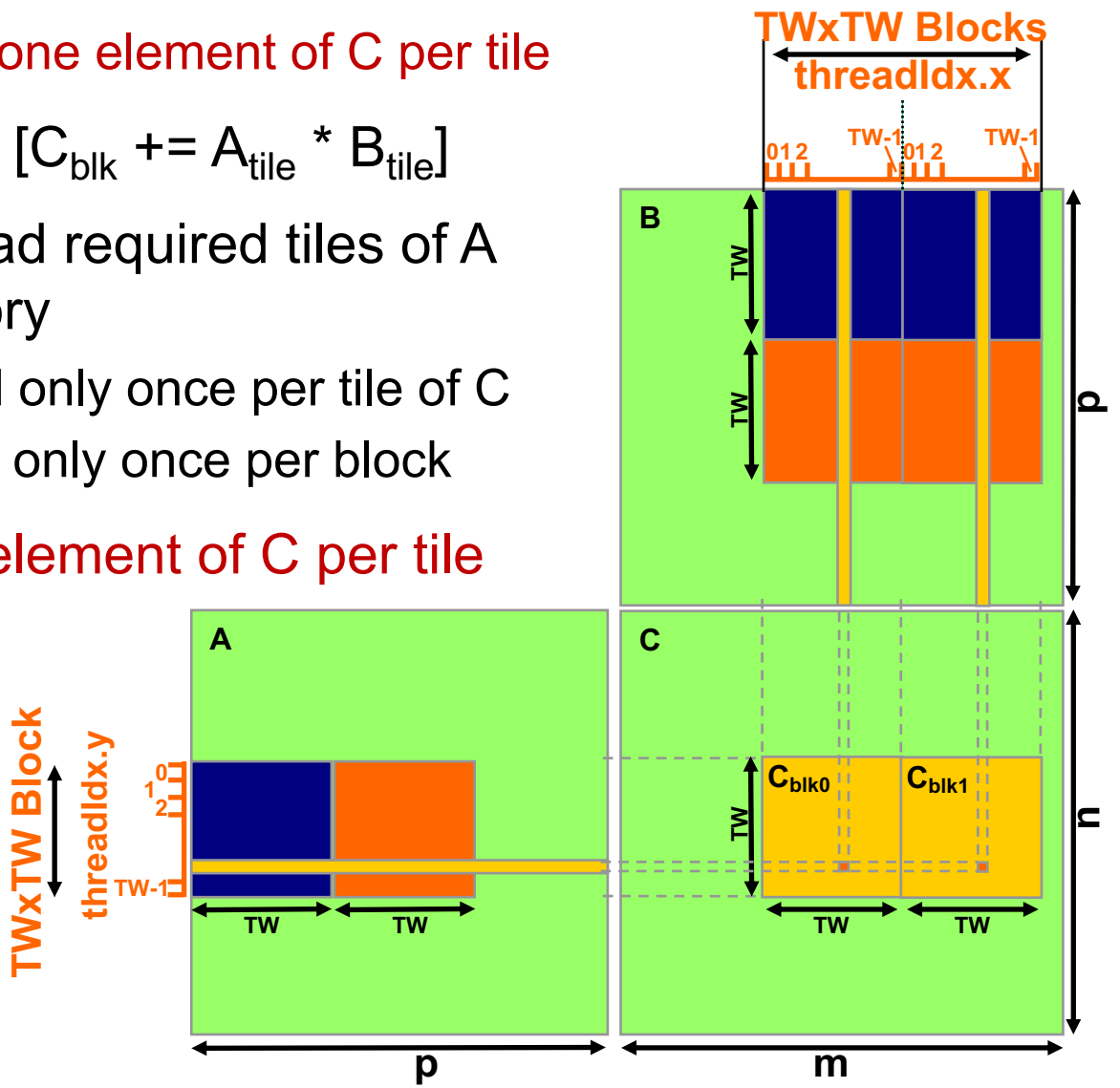
Setting the Size of the Shared Memory on the K80

- Specify size in host code before launch
  - **cudaFuncSetCacheConfig(MyKernel, configuration)**
  - **configuration constants:**
    - **cudaFuncCachePreferShared:** 48KB shared; 80KB L1
    - **cudaFuncCachePreferL1:** 112KB L1, 16KB shared
    - **cudaFuncCachePreferNone:** Default; looks at other settings
- May want to use dynamically-allocated shared memory (though this requires a bit of change to the kernel code):

```
size_t Ns = 2 * TW*TW * sizeof(int);
matmul_tiled<<<Grid,Block,Ns>>>(d_a, d_b, d_c, N);

__global__ void matmul_tiled(. . .) {
   extern __shared__ int bigarray[];
   int *atile=&bigarray[0], *btile=&bigarray[TW*TW];
   . . .
}
```

Multi-Tiled Matrix Multiplication Kernel – Concepts

- Blocks compute multiple TW x TW "tiles" of C
  - Each thread computes one element of C per tile
- "Matrix thought process" [C_{blk} += A_{tile} * B_{tile}]
- Threads cooperate to load required tiles of A and B into shared memory
  - Each element of B read only once per tile of C
  - Each element of A read only once per block
- Each thread writes one element of C per tile

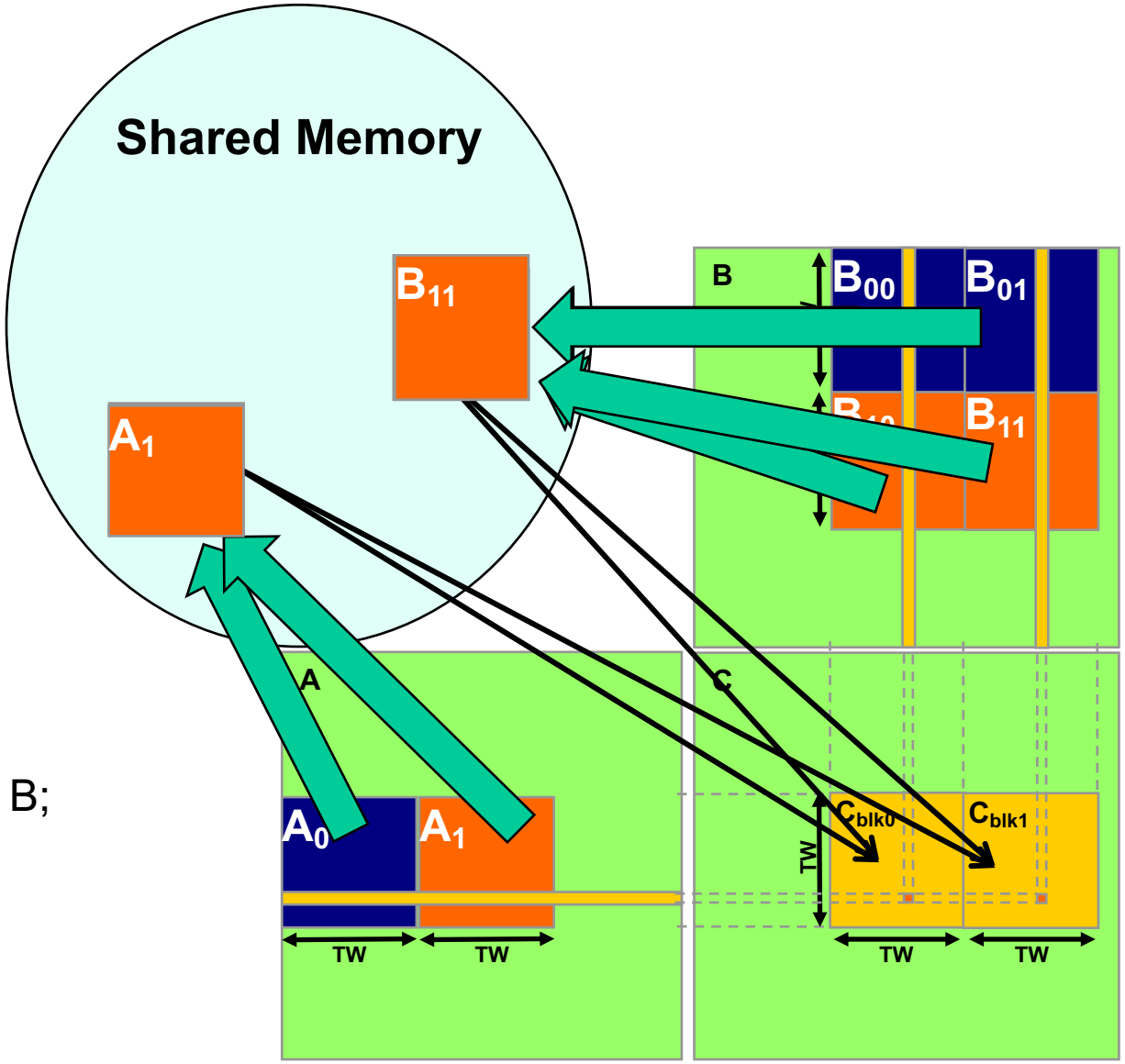


Simplest case:

TW = blockDim.x = blockDim.y

NTB = No. of tiles per block

Multi-Tiled Matrix Multiplication Kernel – Impl. I



Strategy:

Load a tile of A;

Load a tile of B;

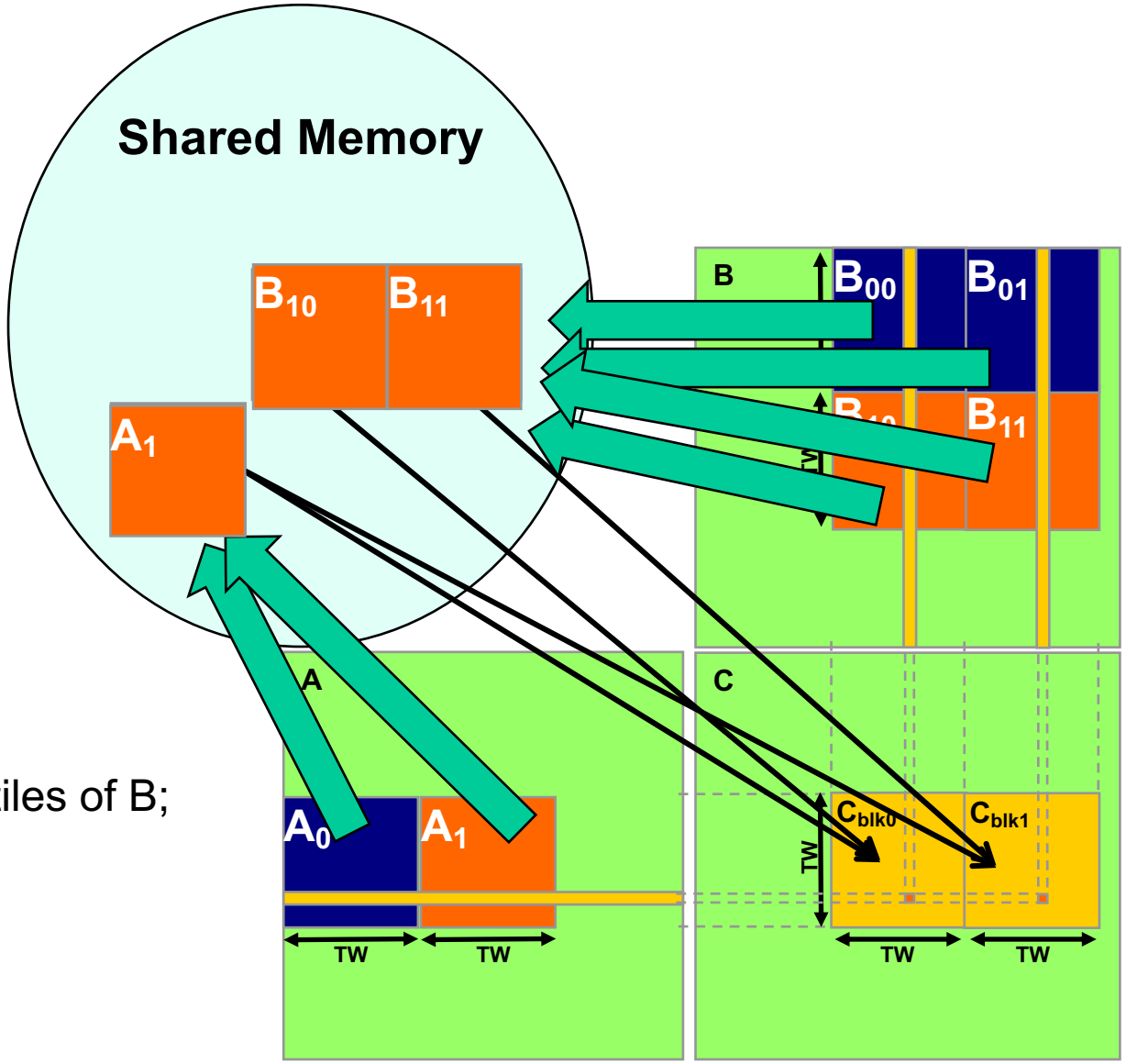
Update C;

Possible Implementation – I

- One possible code structure

```
__shared__ FP atile[TW][TW], btile[TW][TW];
FP cvalue[NTB];
for (kt=0; kt<NTB; kt++) cvalue[kt] = 0.;
for (m=0; m<n/TW; m++) {
   load atile[ty][tx];
   for (kt=0; kt<NTB; kt++) {
      load btile[ty][tx] with element [ty][tx] in kt-th tile of b;
      __syncthreads();
      for (int k=0;k<TW;k++) cvalue[kt] += atile[ty][k]*btile[k][tx];
      __syncthreads();
}
store elements of c;
```

Multi-Tiled Matrix Multiplication Kernel – Impl. II



Strategy:

Load a tile of A;

Load row of tiles of B;

Update C;

Possible Implementation – II

- Another possible code structure

```
__shared__ FP atile[TW][TW], btile[NTB][TW][TW];
FP cvalue[NTB];
for (kt=0; kt<NTB; kt++) cvalue[kt] = 0.;
for (m=0; m<n/TW; m++) {
   load atile[ty][tx];
   for (kt=0; kt<NTB; kt++) load btile[kt][ty][tx];
   __syncthreads();
   for (kt=0; kt<NTB; kt++)
      for (int k=0; k<TW; k++)
         cvalue[kt] += atile[ty][k] * btile[kt][k][tx];
   __syncthreads();
}
store elements of c;
```

Possible Implementation – III

- Another possible code structure (swap loops)

```
__shared__ FP atile[TW][TW], btile[NTB][TW][TW];
FP cvalue[NTB];
for (kt=0; kt<NTB; kt++) cvalue[kt] = 0.;
for (m=0; m<n/TW; m++) {
   load atile[ty][tx];
   for (kt=0; kt<NTB; kt++) load btile[kt][ty][tx];
   __syncthreads();
   for (int k=0; k<TW; k++) {
      for (kt=0; kt<NTB; kt++)
         cvalue[kt] += atile[ty][k] * btile[kt][k][tx];
   }
   __syncthreads();
}
store elements of c;
```

Possible Implementation – IIIa (Shared Memory for cvalue)

- Another possible code structure, using shared memory for cvalue

```
__shared__ FP atile[TW][TW], btile[NTB][TW][TW], cvalue[NTB][TW][TW];
for (kt=0; kt<NTB; kt++) cvalue[kt][ty][tx] = 0.;
for (m=0; m<n/TW; m++) {
   load atile[ty][tx];
   for (kt=0; kt<NTB; kt++) load btile[kt][ty][tx];
   __syncthreads();
   for (int k=0; k<TW; k++)
      for (kt=0; kt<NTB; kt++)
         cvalue[kt][ty][tx] += atile[ty][k]*btile[kt][k][tx];
   __syncthreads();
}
store elements of c;
```

Possible Implementation – IIIb (Unrolled, NTB=4)

- Another possible code structure (unrolled innermost loop; no shared memory needed for cvalue)

```
__shared__ FP atile[TW][TW], btile[NTB][TW][TW];
FP cvalue0=0., cvalue1=0., cvalue2=0., cvalue3=0.;
for (m=0; m<n/TW; m++) {
   load atile[ty][tx];
   for (kt=0; kt<NTB; kt++) load btile[kt][ty][tx];
   __syncthreads();
   for (int k=0; k<TW; k++) {
      cvalue0 += atile[ty][k] * btile[0][k][tx];
      cvalue1 += atile[ty][k] * btile[1][k][tx];
      cvalue2 += atile[ty][k] * btile[2][k][tx];
      cvalue3 += atile[ty][k] * btile[3][k][tx];
   }
   __syncthreads();
}
store elements of c;
```

Possible Implementation – IIIc (Unrolled, NTB=4)

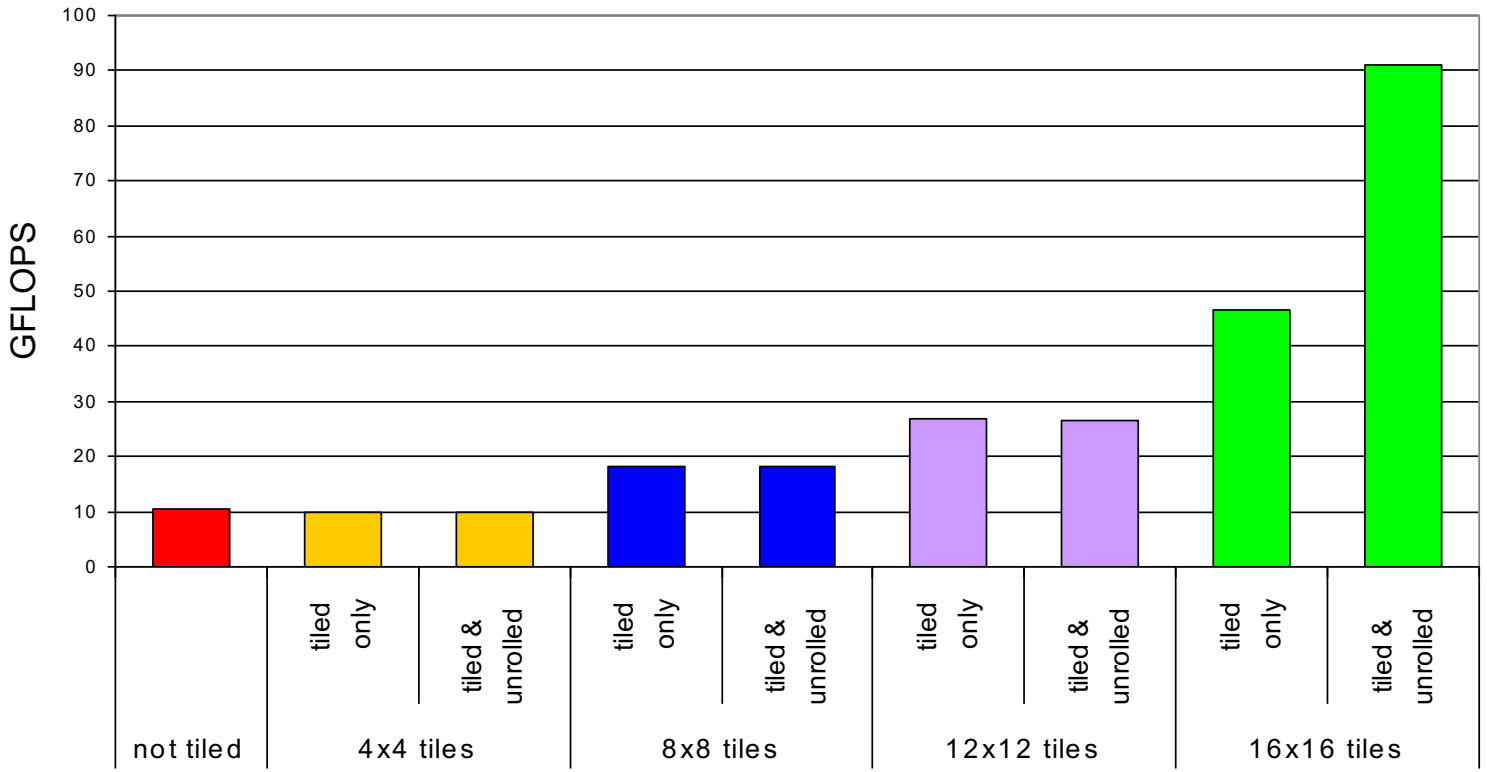
- Another possible code structure (unrolled innermost loop; no shared memory needed for cvalue)

```
__shared__ FP atile[TW][TW], btile[NTB][TW][TW];
FP cvalue0=0., cvalue1=0., cvalue2=0., cvalue3=0., atyk;
for (m=0; m<n/TW; m++) {
   load atile[ty][tx];
   for (kt=0; kt<NTB; kt++) load btile[kt][ty][tx];
   __syncthreads();
   for (int k=0; k<TW; k++) {
      atyk = a[ty][k];
      cvalue0 += atyk * btile[0][k][tx];
      cvalue1 += atyk * btile[1][k][tx];
      cvalue2 += atyk * btile[2][k][tx];
      cvalue3 += atyk * btile[3][k][tx];
   }
   __syncthreads();
}
store elements of c;
```

Performance Considerations for Matrix Multiplication

- SMXs in K40 can have up to 48KB shared memory
  - Shared memory is divided among all resident blocks
  - Each shared matrix tile uses TW*TW*4B of shared memory for single precision (2x for double precision).
  - This limits the number of shared tiles and resident blocks
- Registers & Scheduling
  - On K80, 128K (32-bit) registers per SMX are divided among active resident blocks (at most 64K registers per block)
  - With enough registers, multiple blocks can be resident on each SMX
    - Increases likelihood of always having enough ready threads for all cores
    - Makes it easier to hide latency
  - May be desirable to reduce block dimensions to enable more to fit
    - If a code uses 30 registers per thread, only two 32x32 block can be active per SMX, but 8 16x16 blocks could fit (if shared memory usage permits).
    - This may well give better performance

Tiling Size Effects (on G80, an older GPU)



Topics

- Thread Scheduling & Divergence
- GPU Memory Hierarchy
- Improving matrix multiplication performance
  - Loop unrolling
  - Multi-tiled kernel
- Memory optimizations
  - Global memory
  - Shared memory
- More general performance optimization process

Cuda Global Memory Operations

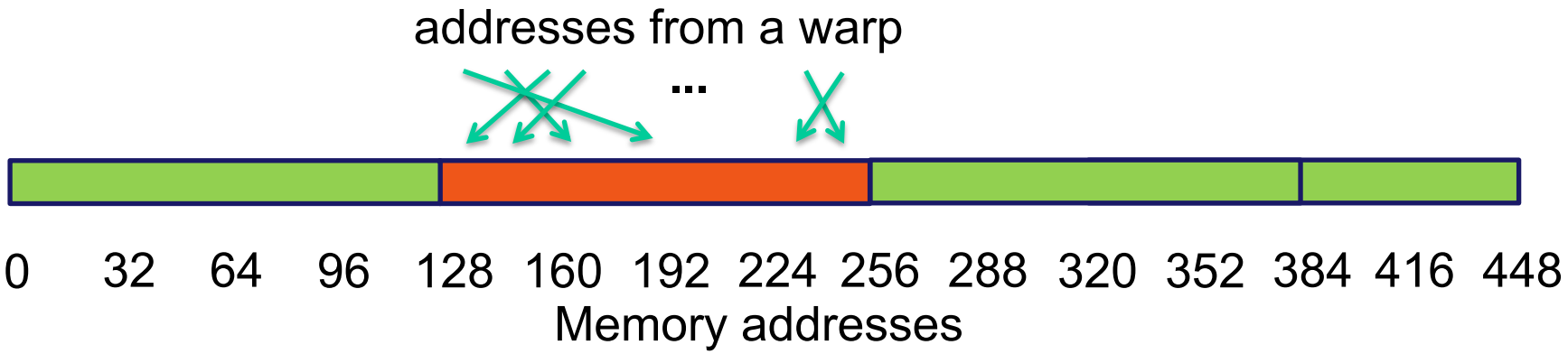
- Memory operations are executed per warp
  - 32 threads in a warp provide memory addresses
  - Hardware determines into which cache lines those addresses fall
  - By default, all operations go through L2 cache (128-byte cache line)
  - Ideally, all 128 bytes are used by 32 threads (1 float/int each)
- Two types of loads:
  - Caching (default mode)
    - Attempts to hit in L1, then L2, then GM (L1 not used in 3.x and higher)
    - Load granularity is 128-byte line
  - Non-caching (mostly ignored here)
    - Compile with "–Xptxas –dlcm=cg" option to nvcc
    - Attempts to hit in L2, then GM
    - Load granularity is 32-bytes
- Stores: Write-back for L2, 32-byte granularity (No L1 for 3.x+)

Caching Load

- Warp requests 32 aligned, consecutive 4-byte words (1 per thread)
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

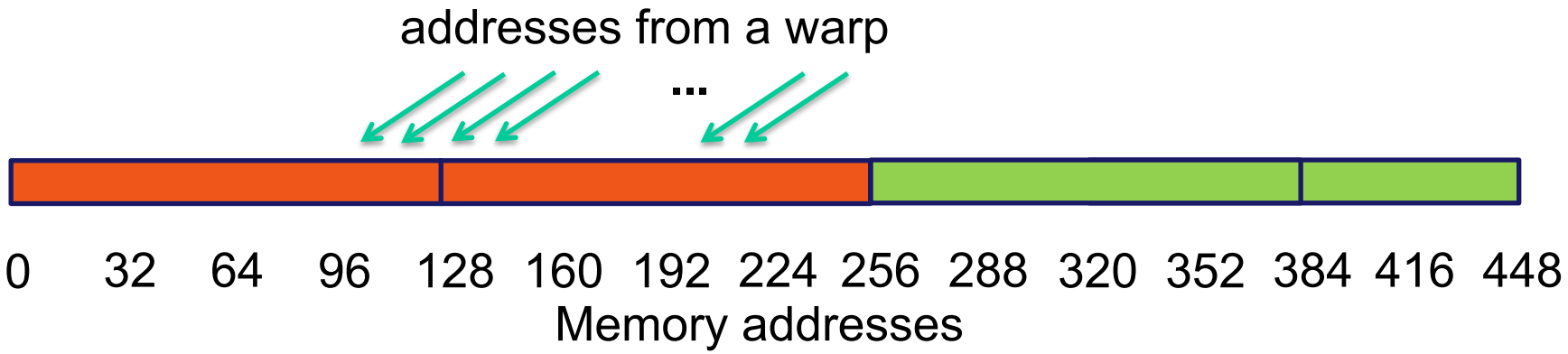
Caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%



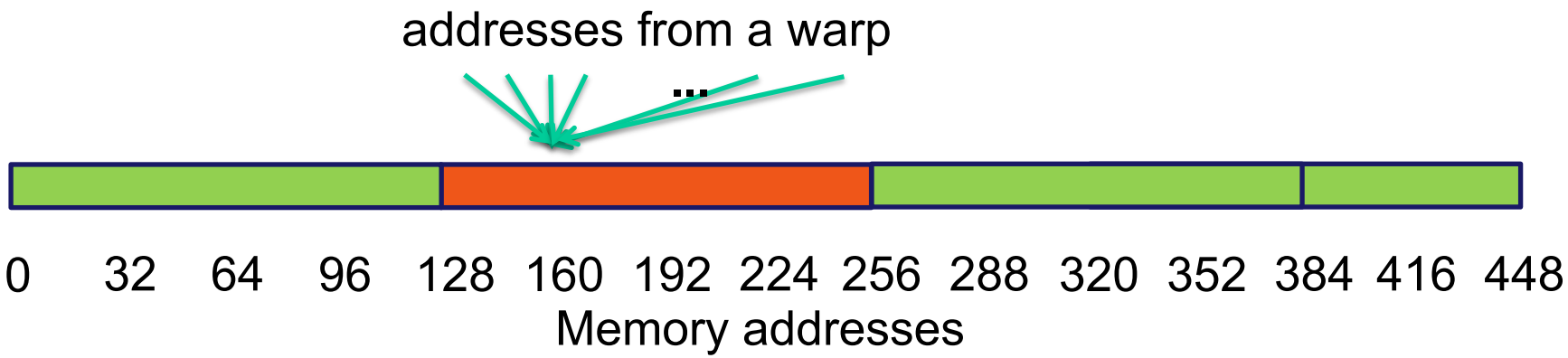
Caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within 2 cache-lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%
- This has been improved in newer devices



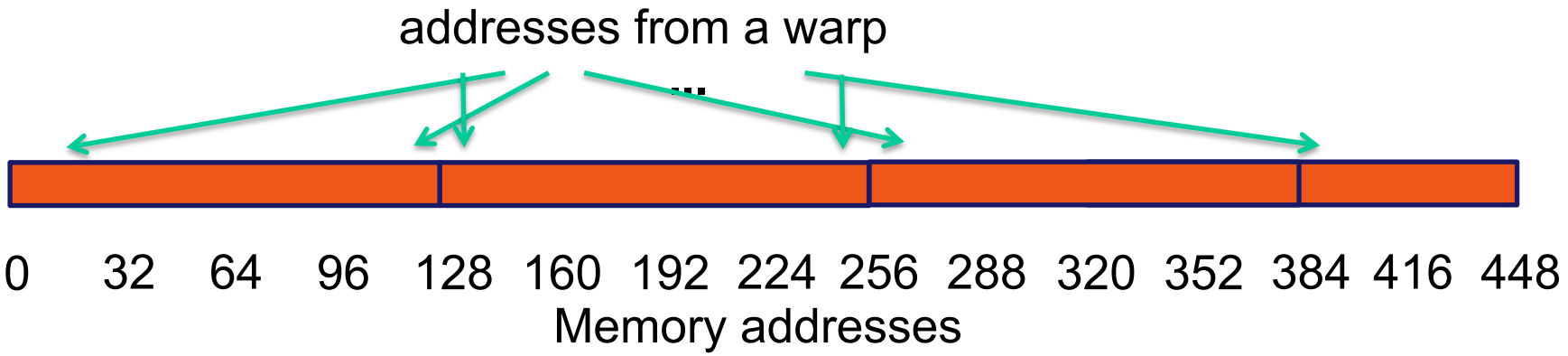
Caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%



Caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within N cache-lines
  - Warp needs 128 bytes
  - N*128 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*128)



Summary: GMEM Optimization

- Strive for perfect “coalescing” per warp
  - Align starting address (may require padding)
  - A warp should access within a contiguous region
  - Structure of Arrays is better than Array of Structures (Why?)
- Have enough concurrent accesses to saturate the bus
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching threads (warps)
  - Process several elements per thread (loop unrolling)
    - Multiple loads get pipelined
    - Indexing calculations can often be reused
- Try L1 and caching configurations to see which one works best
  - Caching vs non-caching loads (compiler option)
  - 16KB vs 48KB L1 (CUDA call)

Topics

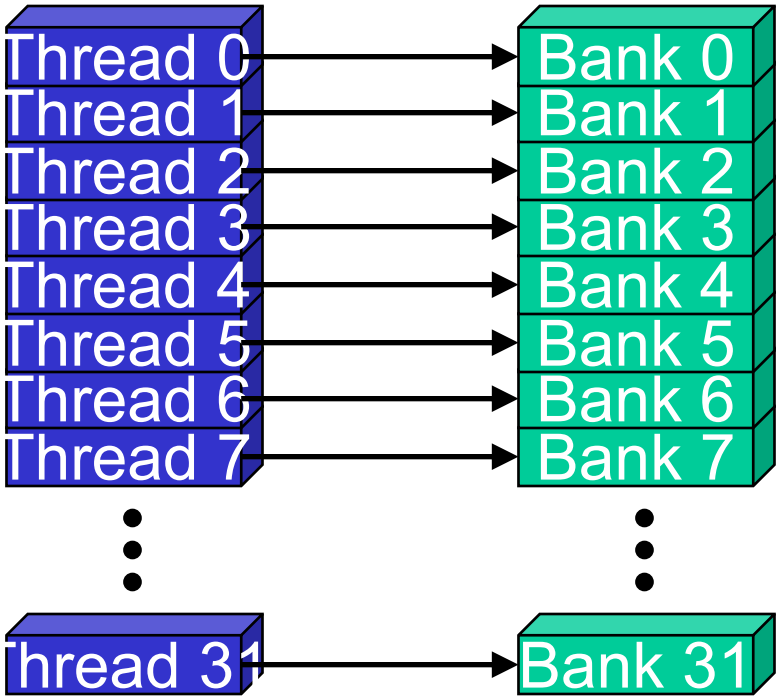
- Thread Scheduling & Divergence
- GPU Memory Hierarchy
- Improving matrix multiplication performance
  - Loop unrolling
  - Multi-tiled kernel
- Memory optimizations
  - Global memory
  - Shared memory
- More general performance optimization process

Shared Memory Optimization

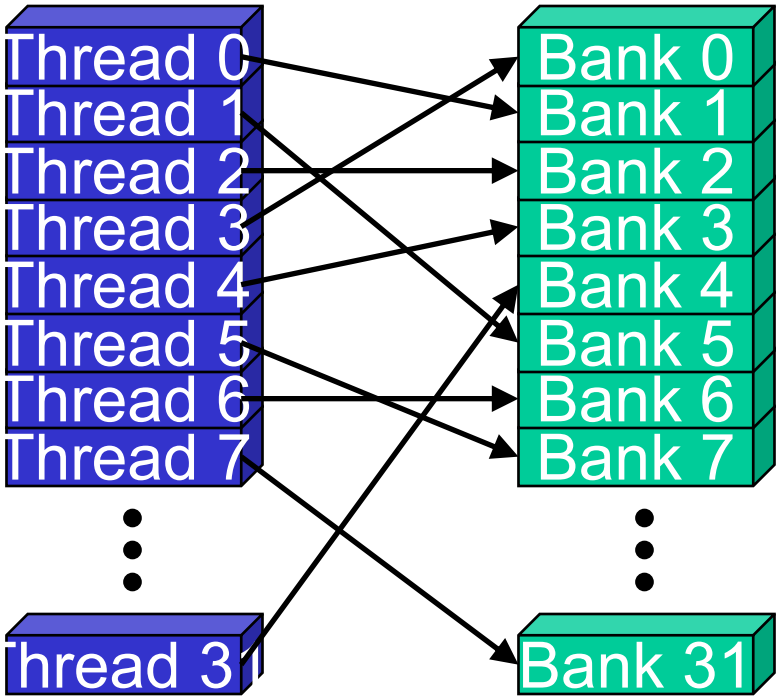
- Uses:
  - Inter-thread communication within a block
  - Manually cache data to reduce redundant global memory accesses
  - Improve global memory access patterns (may rearrange in Smem)
- Shared Memory Organization:
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks (“round robin”)
- Performance:
  - 4 bytes per bank per 2 clocks per SM; Want to avoid bank conflicts
  - Shared memory accesses are issued per 32 threads (warp)
  - Serialization: if *n* threads in a warp access different 4-byte words in the same bank, the accesses are executed serially
  - Multicast: *n* threads can access bytes in the same word in one fetch
  - 8-byte accesses handled specially to avoid conflicts

Bank Addressing Examples

- No Bank Conflicts

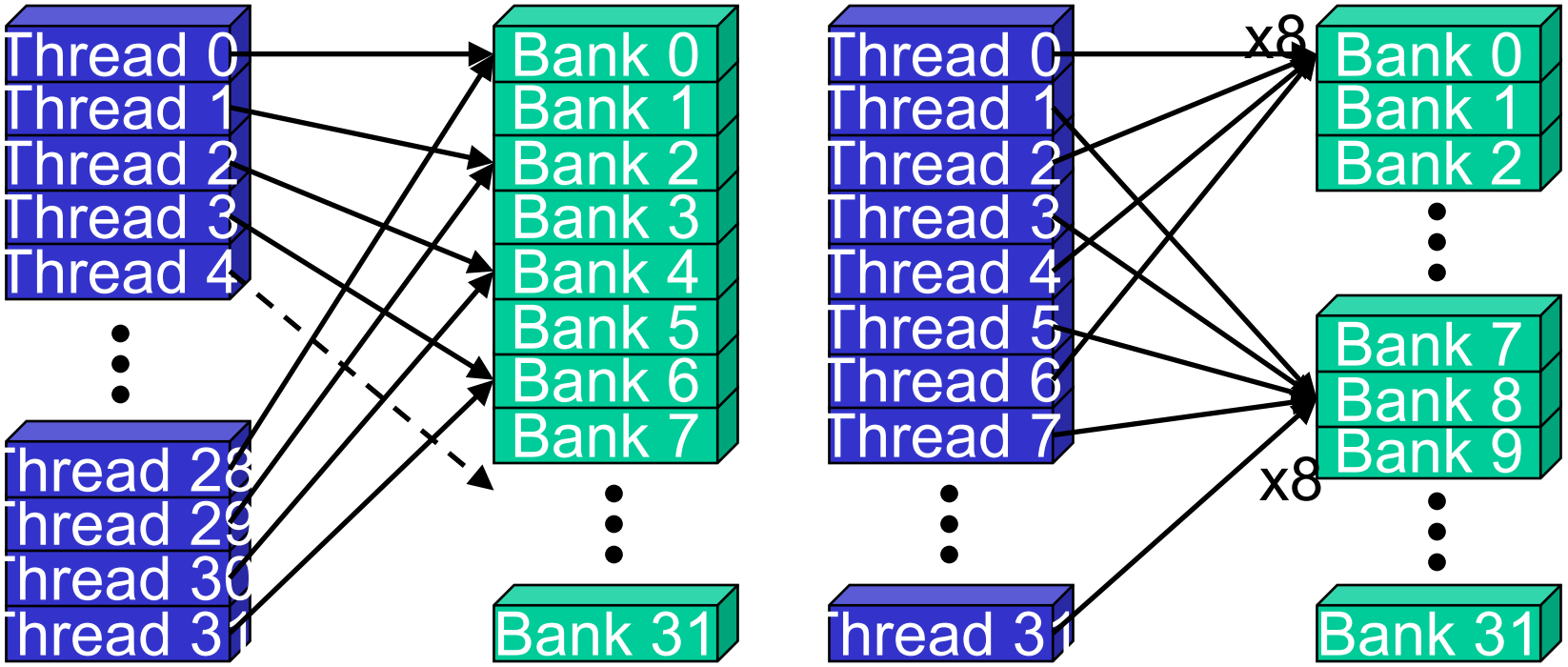


- No Bank Conflicts



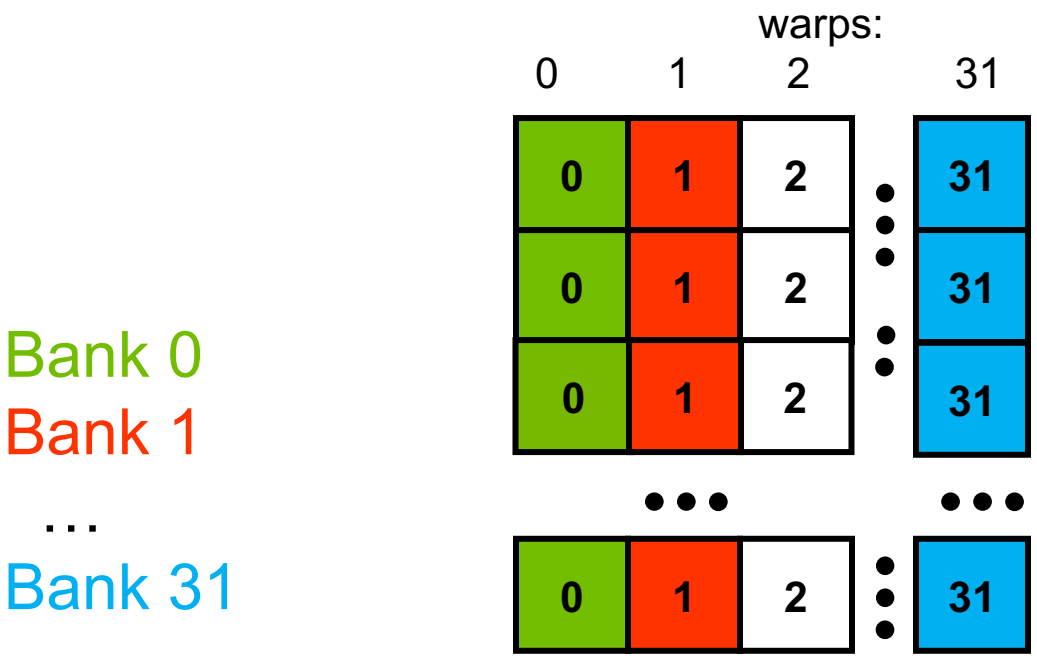
Bank Addressing Examples

- 2-way Bank Conflicts
- 8-way Bank Conflicts



Shared Memory: Avoiding Bank Conflicts

- 32x32 SMem array
- Warp accesses a column:
  - 32-way bank conflicts (all threads in a warp access the same bank)



Shared Memory: Avoiding Bank Conflicts

- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts

Bank 0
Bank 1
…
Bank 31

Topics

- Thread Scheduling & Divergence
- GPU Memory Hierarchy
- Improving matrix multiplication performance
  - Loop unrolling
  - Multi-tiled kernel
- Memory optimizations
  - Global memory
  - Shared memory
- More general performance optimization process

Performance Optimization Process

- Determine what limits kernel performance
  - Memory throughput
  - Instruction throughput
  - Memory Latency
  - Combination of the above
- Focus on appropriate performance metric(s) for each kernel
  - E.g., Gflops/s aren't the issue for a bandwidth-bound kernel
- Address the limiters in order of importance
  - Determine how close to the HW limits each resource is
  - Analyze for possible inefficiencies
  - Apply optimizations

3 Ways to Assess Performance Limiters

- Algorithmic (theory)
  - Based on algorithm's memory and arithmetic requirements
  - Least accurate
    - Tends to undercount instructions and, potentially, memory accesses
    - May not properly assess hardware behaviors, but can set expectations
- Profiler
  - Based on profiler-collected memory and instruction counters
  - More accurate
    - Reflects actual behavior on the hardware (based on sample runs)
    - May not account well for overlapped memory and arithmetic operations
- Custom code instrumentation
  - Modify source to measure memory-only and arithmetic-only times
  - Most accurate, but may be difficult to implement for some codes
  - Can use printf() to print information

Things to Know About Your GPU

- Important memory specs:
  - For K80
    - Global memory: ~12 GB; High latency; BW = 288 GB/sec
    - Shared memory: up to 48 KB/SMX; low latency; BW = 216 GB/sec/SMX
    - Register file: 128K per SMX (64K can be used by one thread block)
    - Refer to CUDA Programming Guide or to architecture documentation for details
- Theoretical instruction throughput
  - Varies by instruction type and GPU compute capability
    - refer to the CUDA Programming Guide for details

Using the Profiler

- Relevant hardware counters:
  - instructions_issued
    - Incremented by 1 per warp, counter is for one SM
  - dram_reads, dram_writes
    - Incremented by 1 per 32B access to DRAM
    - Note that the VisualProfiler converts each of the above to 2 counters
  - For codes with many L2 cache hits, look at L2 counters (accesses to L2 are expensive compared to arithmetic)
- Compute instruction:byte ratio and compare to the balanced one:
  - (#SMs) * 32 * insts_issued : 32B * (dram_reads + dram_writes)
  - Ratios less than 3.76 are memory bound