AMS 250: An Introduction to High Performance Computing

GPU Computing

Shawfeng Dong

shaw@ucsc.edu

(831) 502-7743

Applied Mathematics & Statistics

University of California, Santa Cruz

Outline

- Why GPU / Accelerator-based / Manycore Computing?
- GPU Architecture
- Introduction to CUDA C Programming
  - Thread Hierarchy
  - Memory Hierarchy
  - CUDA C Extensions
  - CUDA Runtime API
  - Shared Memory
  - CUDA Streams

Why GPU Computing?

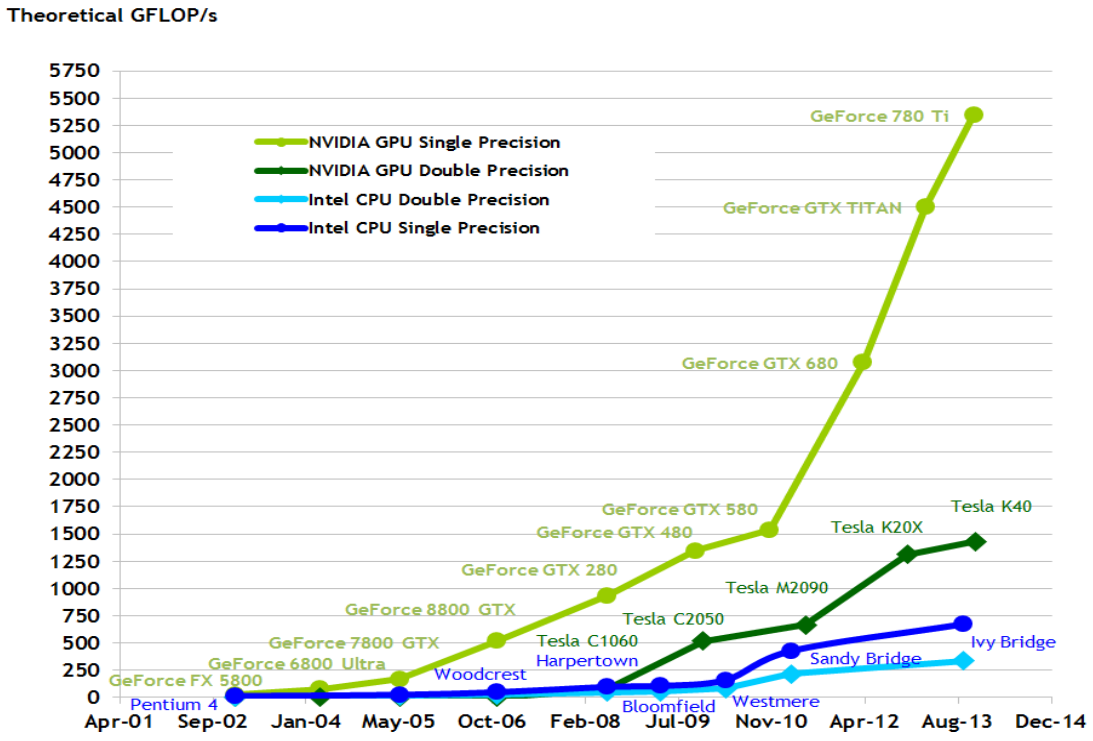
"I think they're right on the money, but the huge performance differential (currently 3 GPUs ~= 300 SGI Altix Itanium2s) will invite close scrutiny so I have to be careful what I say publically until I triple check those numbers."

-John Stone, UIUC, circa 2007

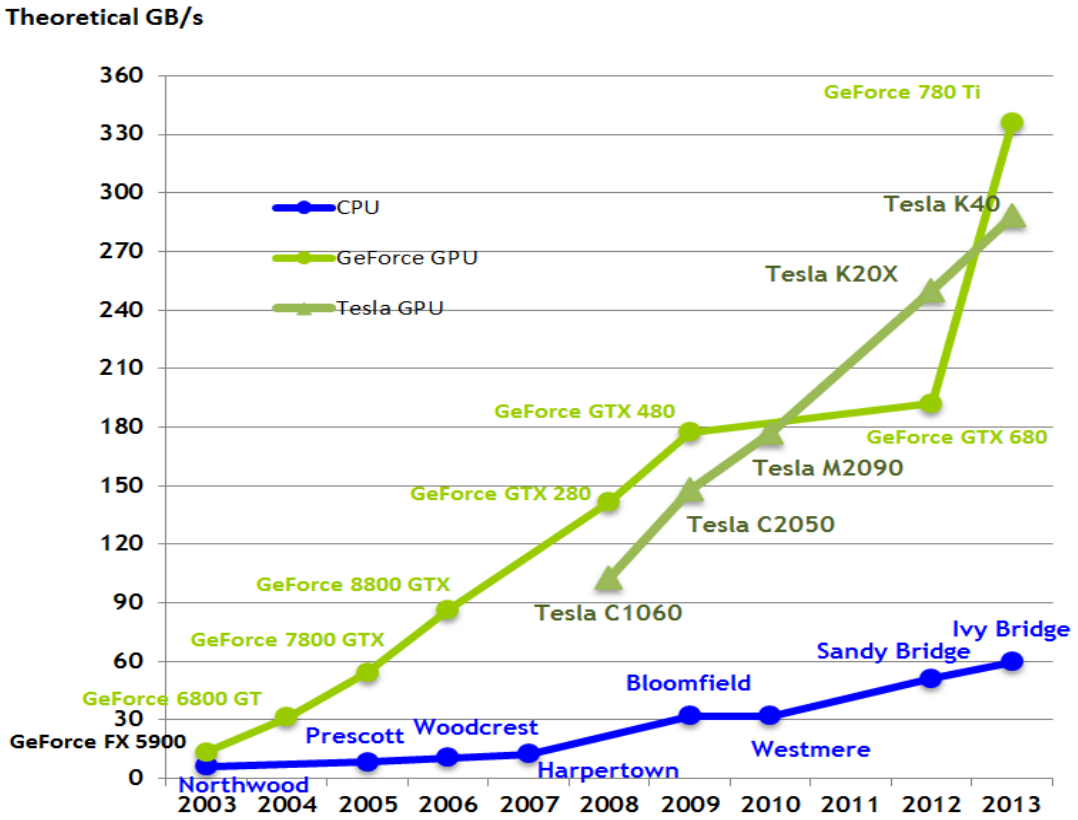
Is a top-of-the-line GPU 100 times faster than a top-of-the-line CPU?

Why Manycore Computing?

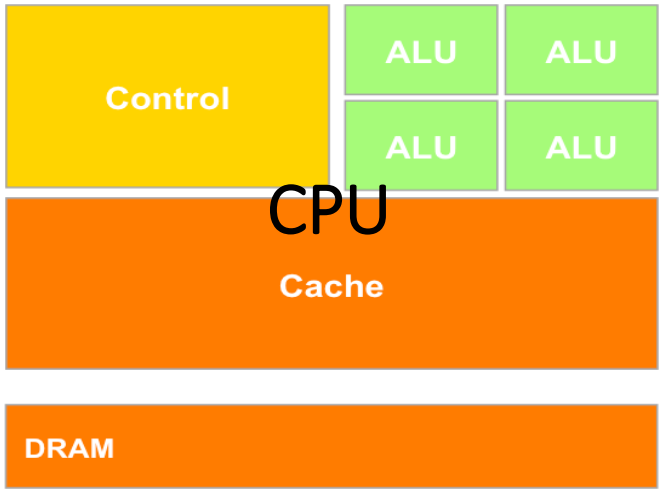
FLOPS

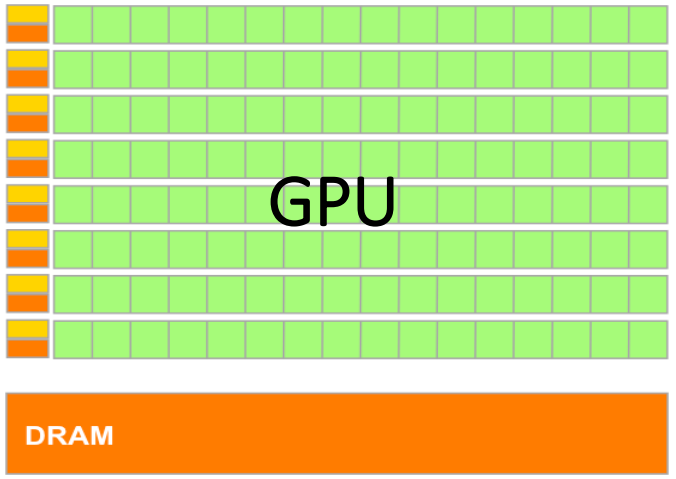


Memory Bandwidth



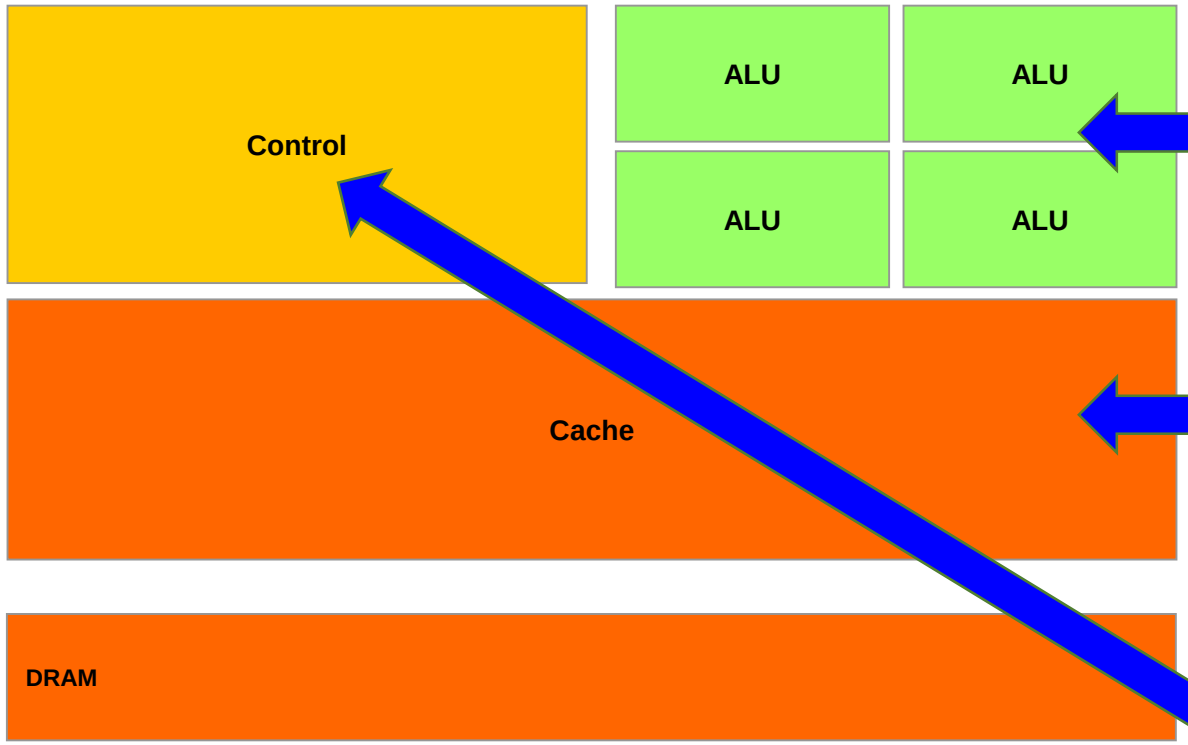
Multicore CPU vs. Manycore GPU





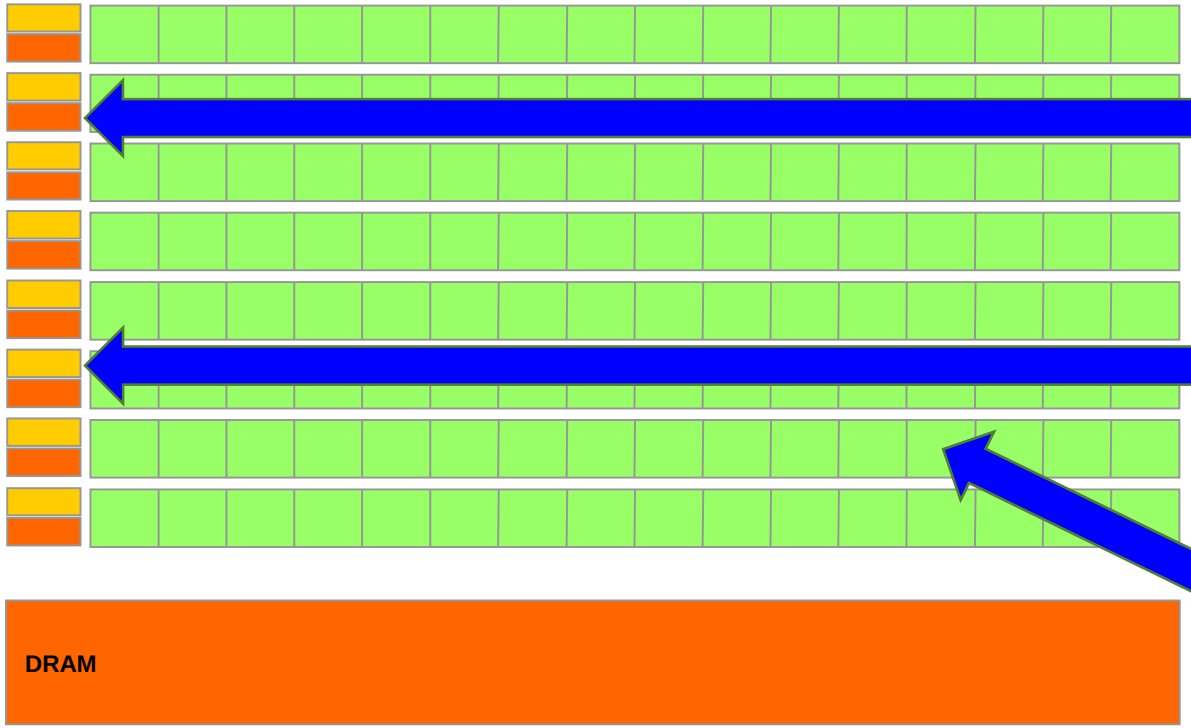
- CPU: Latency Oriented Design
  - Optimized for low latency access to cached data sets
- GPU: Throughput Oriented Design
  - Optimized for data-parallel, high throughput computing

CPU: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

GPU: Throughput Oriented Design



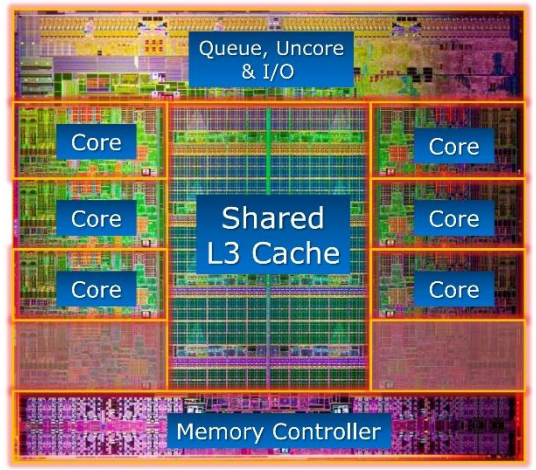
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency, but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Winning Applications Use both CPU and GPU

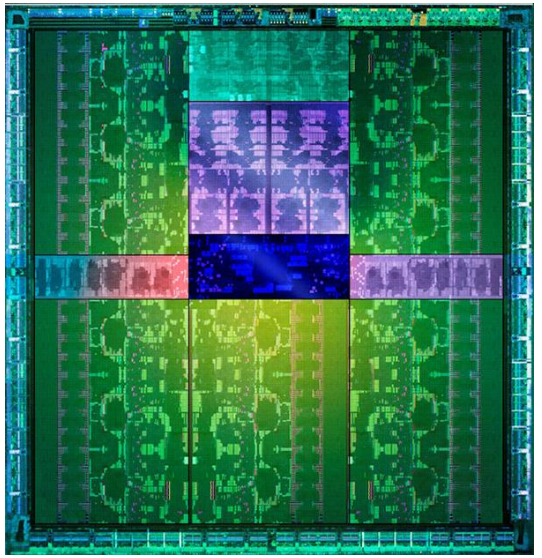
- CPUs for sequential parts where latency matters
- CPUs can be 10X+ faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
- GPUs can be 10X+ faster than CPUs for parallel code

Multicore versus Manycore – Real Chips

	Intel Xeon E5-2650	Nvidia Tesla K20	
Architecture	Sandy Bridge	Kepler	
Processing Units	8 cores (with 256-bit AVX)	13 SM / 2496 cores / 832 DP units	
Clock Speed	2.0 GHz	706 MHz	
Single Precision	256 GLOPS	3.52 TFLOPS	
Double Precision	128 GFLOPS	1.17 TFLOPS	~9x
Memory Bandwidth	51.2 GB/s	208 GB/s	~4x
Memory Size	32 GB	5 GB	
Registers	~100 per core	65536 x 32-bit per SM	
L1 Cache	64 KB per core	64 KB per SM	
L2 Cache	256 KB per core	768 KB shared	
L3 Cache	20 MB shared	N/A	
Process	32nm	28nm	
Transistor Count	2.3B	7.1B	
Thermal Design Power	95W	225W	



Intel Sandy Bridge (32nm)

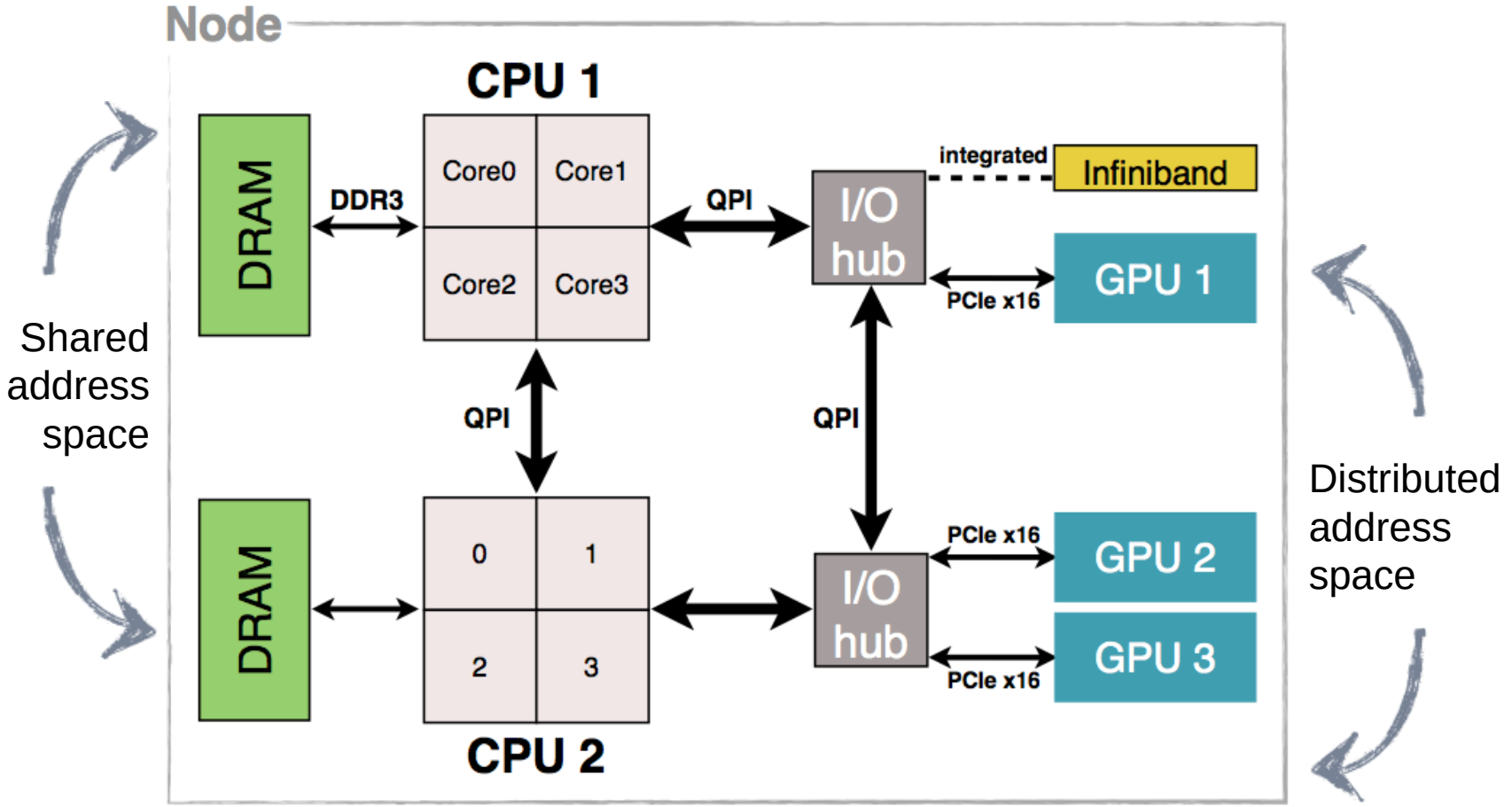


Nvidia GK110 (28nm)

Connecting Manycore GPUs to Multicore CPUs

Typically, GPU devices are accessed over PCIe

- Bandwidth of PCIe 2.0 x16 = 8 GB/s (18GB/s duplex)
- Bandwidth of PCIe 3.0 x16 = 15.75 GB/s (31.5GB/s duplex)

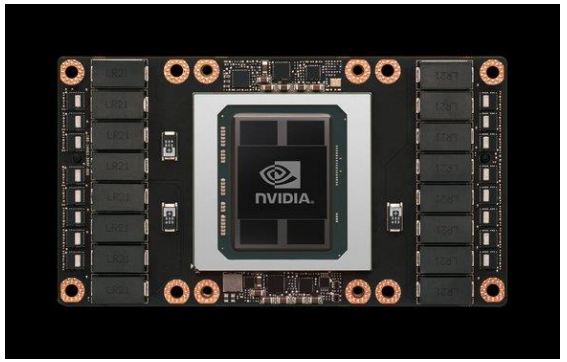


Nvidia Tesla K20 GPU

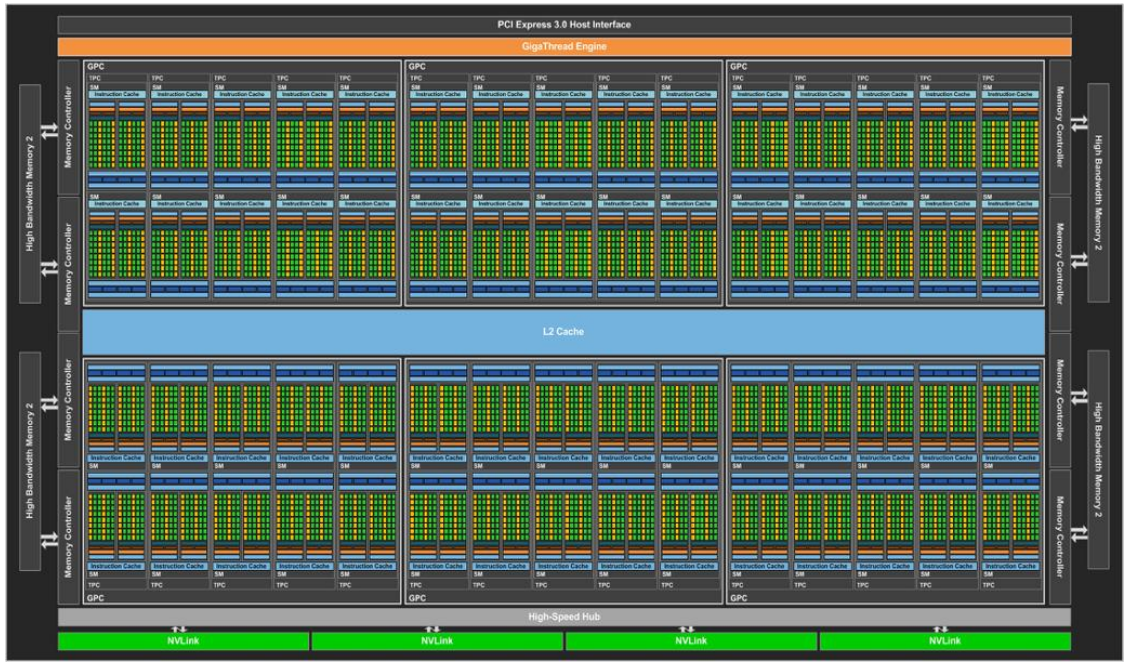
- **Kepler** microarchitecture: GK110
- Compute capacity: 3.5
- 2496 FP32 cores / 832 FP64 cores / 13 SMs
  - 192 FP32 cores and 64 FP64 cores per SM
- Core speed: 706 MHz
- Double precision performance: 1.17 TFLOPS
  = 0.706 (GHz) x 832 (DP units) x 2 (FMA)
- Single precision performance: 3.52 TFLOPS
  = 0.706 (GHz) x 2496 (CUDA cores) x 2 (FMA)
- Memory: 5.2GHz, 320-bit wide, 5GB GDDR5
  5.2 (GHz) x 320 / 8 = 208 GB/s
- PCI express *2.0* x16
  500 (MB/s) x 8/10 x 16 = 8 GB/s (16 GB/s duplex)

http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last

Nvidia Tesla P100 GPU

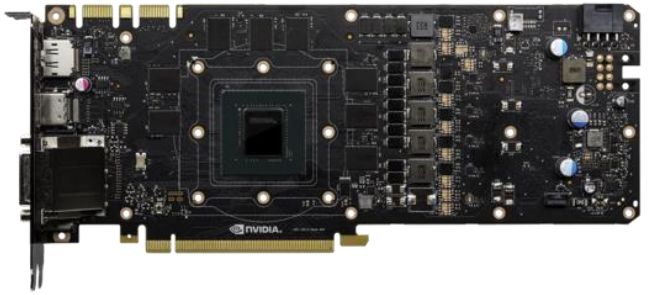


- **Pascal** microarchitecture: GP100
- Compute capacity: 6.0
- 3584 FP32 cores / 1792 FP64 cores / 56 SMs
  - 64 FP32 cores and 32 FP64 cores per SM
- Core speed: 1328 MHz / 1480 MHz (boost)
- Double precision performance: 4.76 TFLOPS
- Single precision performance: 9.3 TFLOPS
- Half precision performance: 18.7 TFLOPS
- Memory: 1.75GHz CoWoS HBM2
  16GB, 4096-bit, at 732GB/s,
  or 12GB, 3072-bit, at 549 GB/s
- PCI express 3.0 x16
  15.75 GB/s (31.5 GB/s duplex)



http://www.pcworld.com/article/3052222/components-graphics/nvidias-pascal-gpu-tech-specs-revealed-full-cuda-count-clock-speeds-and-more.html

Nvidia GeForce GTX 1080 Ti



- **Pascal** microarchitecture: GP102
- Compute capacity: 6.1
- 3584 FP32 cores / 0 FP64 cores / 28 SMs
  - 128 cores per SM
- Core speed: 1582 MHz
- Single precision performance: 11.4 TFLOPS
- Half precision performance: 178.1 GFLOPS (!)
- Double precision performance: 356.2 GFLOPS
- Memory: 11GHz, 352-bit wide, 11GB GDDR5X
  11 (GHz) x 352 / 8 = 484 GB/s
- PCI express 3.0 x16
  15.75 GB/s (31.5 GB/s duplex)

https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/

Nvidia Tesla V100 GPU

- **Volta** microarchitecture: GV100
- Compute capacity: 7.0
- 5120 CUDA cores / 640 Tensor Cores / 80 SMs
  - 64 CUDA cores and 8 Tensor cores per SM
- Double precision performance: 7 (PCIe) / 7.8 (SXM2) TFLOPS
- Single precision performance: 14 / 15.6 TFLOPS
- Half precision performance: 28 / 31.2 TFLOPS
- Tensor performance: 112 / 125 TFLOPS
- Memory: 1.75GHz CoWoS Stacked HBM2
  32/16GB, 4096-bit, at 900GB/s
- Interconnect Bandwidth
  - PCIe 3.0 x16: 15.75 GB/s (31.5 GB/s duplex)
  - Nvidia NVLink: 300 GB/s

https://www.anandtech.com/show/11367/nvidia-volta-unveiled-gv100-gpu-and-tesla-v100-accelerator-announced

Hummingbird GPU Node

- There is a GPU node in the Hummingbird cluster
  https://www.hb.ucsc.edu/
- Master node of Hummingbird: hb.ucsc.edu
  – Use your UCSC account name and *blue* password to log in
    `ssh -l username hb.ucsc.edu`
- Specifications of the GPU node
  - 2x Intel *"Skylake"* Xeon Silver 12-core 4116 processors @ 2.1 GHz
  - 96 GB memory
  - 4x Tesla P100 PCIe GPUs, each with 16 GB HBM2 memory

Interactive Shell on the GPU Node

1. Log in to Hummingbird master node: `ssh -l username hb.ucsc.edu`
2. Use the **salloc** command to request an interactive job. For example, to request 1 Tesla P100 GPU and 24GB memory:

   ```
   salloc -p 96x24gpu4 --gres=gpu:p100:1 --mem=24000
   ```

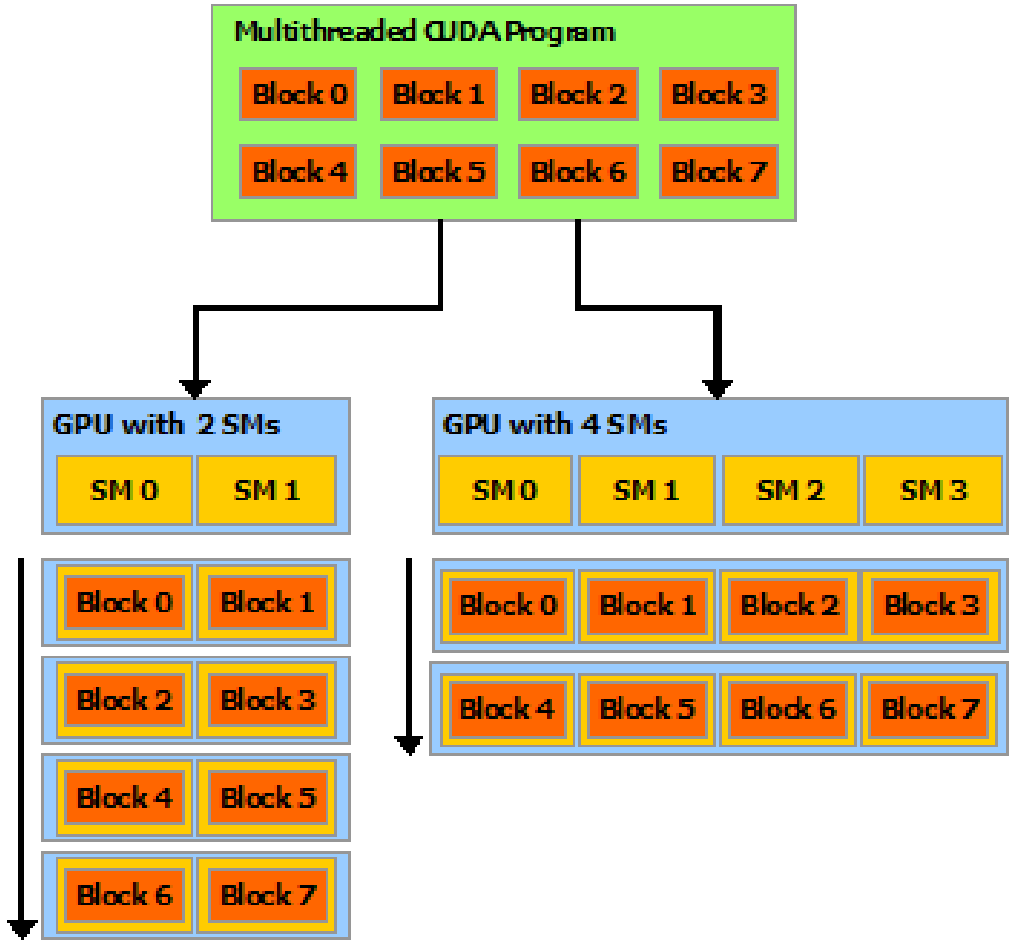
3. Once the requested resources are allocated, **salloc** will return; you will land on a new shell on the master node, and will be in your *work directory* where the salloc command was executed.
4. SSH to the GPU node: `ssh hbcgpu-021`
5. You can now run interactive commands, such as nvidia-smi, on the GPU node
6. Once you are done, log out of the GPU node: `exit`
7. Don't forget to relinquish the job allocation: `exit`

Streaming Multiprocessors (SM)

- GPU
  - A number of SMs (Steaming Multiprocessors)
  - Memory
- Each SM has its own
  - Control unit
  - Pipeline for execution
    - 192 FP32 cores, 64 FP64 cores per SM on K20
    - 64 FP32 cores, 32 FP64 cores per SM on P100
    - 64 FP32 cores, 0 FP64 cores per SM on 1080
  - Registers
  - Shared memory/L1 cache
  - Read-only data cache
  - Texture unit
- SM is the unit of transparent scalability

Transparent Scalability

- Threads are assigned to SMs in block granularity
  - Coarse-grained data parallelism
  - Task parallelism
- Threads of a thread **block** execute concurrently on *one* SM
- Multiple thread blocks can execute concurrently on the same SM
  - Up to 2048 threads per SM
  - Up to 16 blocks per SM (Kepler GK110)
- Each block can execute in any order relative to other blocks



https://www.microway.com/knowledge-center-articles/in-depth-comparison-of-nvidia-tesla-kepler-gpu-accelerators/

Single Instruction Multiple Threads

- SIMT (**S**ingle **I**nstruction **M**ultiple **T**hreads) is the execution model on GPUs
- SIMT is a special case of SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata)
- Fine-grained data parallelism and thread parallelism within a thread block
- GPU threads are lightweight and fast switching
- GPU needs 1000s of threads for full efficiency
  - Max threads per SM = 2048 (on Kepler GK110 & Pascal GP104)
  - Could be 8 blocks if 256 threads per block (typical)
  - Could be 16 blocks if 128 threads per block
- Each block is executed as 32-thread ***warps*** (32-way SIMD)
  - An implementation decision, not part of CUDA programming model
  - Warps are scheduling units in SM
  - If threads of a warp diverge via conditional branch, the warp *serially* executes each branch path taken

3 ways to Accelerate Applications with GPU

Programming Languages	Compiler Directives	Libraries
• CUDA C, CUDA Fortran, OpenCL, etc. • Maximum flexibility • Most performance	• OpenACC, OpenMP 4.5, etc. • Easy to use • Portable code	• cuBLAS, cuRAND, cuFFT, etc. • Easy to use, “drop-in” acceleration • Most performance

GPU Programming Languages

Programming Languages: Most Performance and Flexible Acceleration

☺**Performance:** Programmer has best control of parallelism and data movement

☺**Flexible:** The computation does not need to fit into a limited set of library patterns or directive types

☹**Verbose:** The programmer often needs to express more detail

CUDA Programming Model

- CUDA = **Compute Unified Device Architecture**
- General purpose programming model for GPU
  - GPU as a dedicated super-threaded, massively data parallel co-processor
  - Optimized for computation (graphics-free API)
  - Data sharing with OpenGL buffer objects
  - Explicit GPU memory management
- CUDA C consists of a minimal set of extensions to the C language and a runtime library
  - Declspecs
  - Built-in types
  - Built-in variables
  - Runtime functions
  - Kernel launches

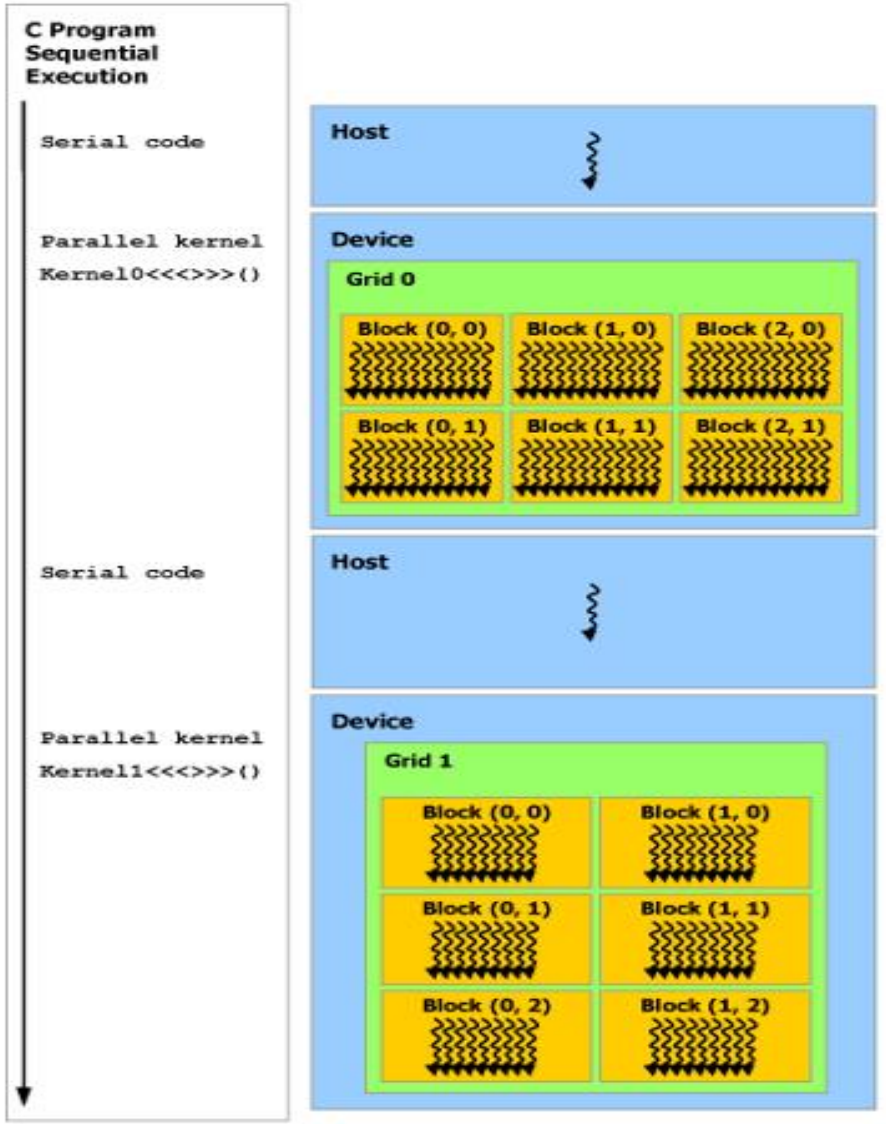
What CUDA Supports

- Thread parallelism
  - each thread is an independent thread of execution
- Data parallelism
  - across thread in a block
  - across blocks in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels executing in separate *streams*

Heterogeneous Programming

Integrated host+device C/C++ program

- Serial or modestly parallel parts in host C code
- Highly parallel parts in device kernel code



“Hello, world!” in CUDA C

```
#include <stdio.h>

__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello, world!\n");
    return 0;
}
```

hello.cu

- CUDA source files use the file name suffix ".cu"
- Build and run the code:

```
$ nvcc hello.cu -o hello.x
$ ./hello.x
Hello, world!
```

- mykernel does nothing, somewhat anticlimactic!

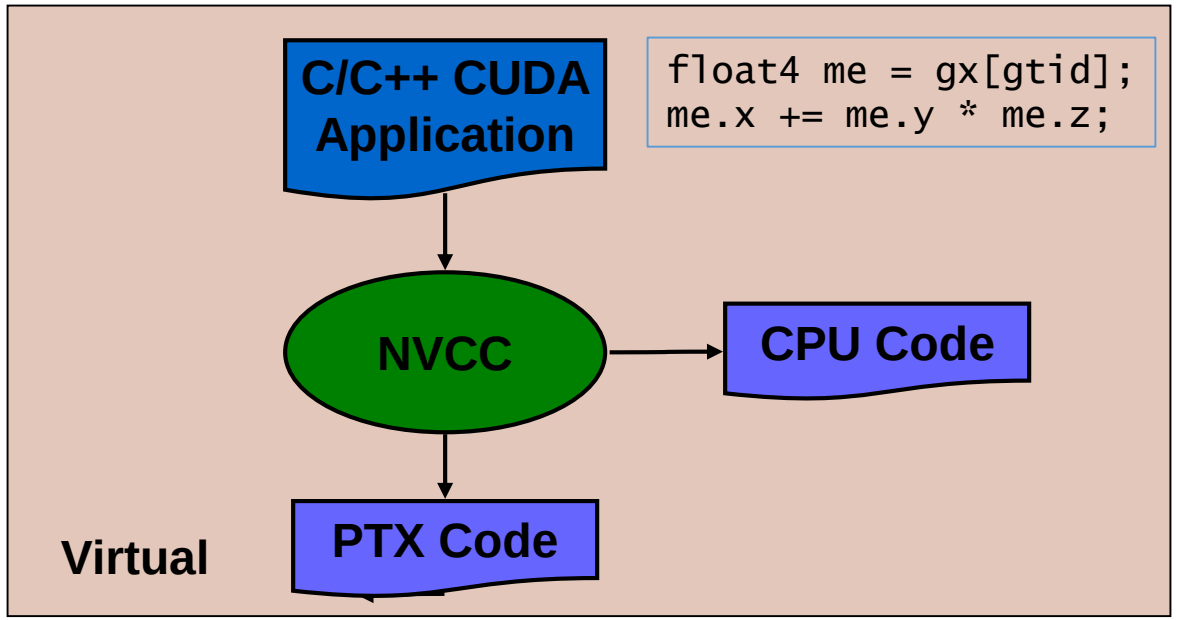
Compilation

- Any source file (with .cu suffix) containing CUDA language extensions must be compiled with nvcc
- nvcc is a compiler driver
- nvcc outputs:
  - **C** code for host, which is then compiled with *gcc/g++* (by default) on Linux
  - **PTX** intermediate assembly code for device, which is
    - Either compiled to object code directly
    - Or interpreted at runtime

For example:

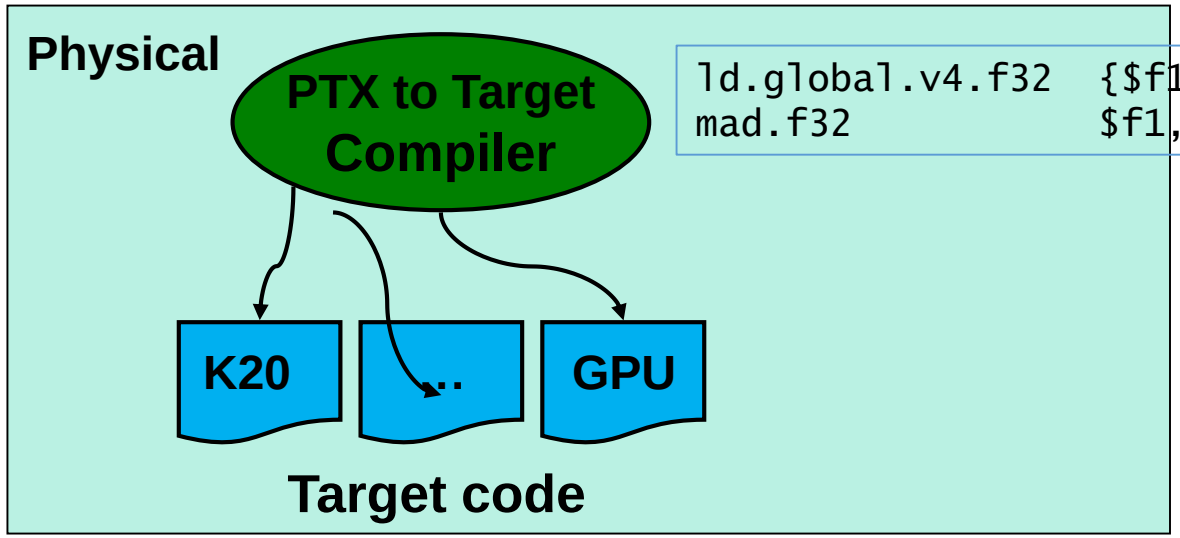
```
$ nvcc hello.cu -o hello.x
```

Compiling a CUDA Program



Parallel **T**hread e**X**ecution (**PTX**)

- Virtual Machine and ISA
- Programming model
- Execution resources and state



http://docs.nvidia.com/cuda/parallel-thread-execution/index.html

nvcc

- To learn more about the CUDA Compiler Driver **nvcc**,
  - consult *nvcc* documentation: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html
  - or check the online help with `nvcc -h`
- By default, **nvcc** invokes *gcc/g++* for host code compilation on Linux.
- You can use the `-ccbin` option to specify a different compiler for host code compilation. For example, if you prefer Intel C/C++ compiler, use the option `-ccbin icpc`.
- Use the `-Xcompiler` option to specify options directly to the compiler/preprocessor.

nvcc (cont'd)

- By default, **nvcc** compiles codes for *Fermi* GPUs (the default is `-arch=compute_20 -code=sm_20,compute_20`)
  - The features of a GPU device depend on its **compute capability**, represented by a version number (*X.Y*)
  - The compute capability of Fermi GPUs is 2.0
  - The compute capability of Tesla K20 is 3.5
  - The compute capability of Tesla P100 is 6.0
  - The compute capability of GTX 1080 Ti is 6.1
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities
  - https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-steering-gpu-code-generation

nvcc (cont'd)

- To generate code for Nvidia Tesla K20 (a *Kepler* GPU), use option `-arch=sm_35`, e.g.:

```
$ nvcc -Xcompiler "-O3" -arch=sm_35 hello.cu -o hello.x
```

- To generate code for Nvidia GTX 1080 Ti (a *Pascal* GPU), use option `-arch=sm_61`, e.g.:

```
$ nvcc -Xcompiler "-O3" -arch=sm_61 hello.cu -o hello.x
```

- To generate code for Nvidia Tesla P100 (a *Pascal* GPU), use option `-arch=sm_60`, e.g.:

```
$ nvcc -Xcompiler "-O3" -arch=sm_60 hello.cu -o hello.x
```

- To generate codes for both K20 & P100 (*fat binary*), e.g.:

```
$ nvcc -Xcompiler "-O3" -gencode arch=compute_35,code=sm_35 \
       -gencode arch=compute_60,code=sm_60 \
       hello.cu -o hello.x
```

Compiling a CUDA Program on Hummingbird

1. On Hummingbird master nodes, load the module for CUDA Toolkit 9.1:

```
$ module load cuda
```

2. Compile for Nvidia Tesla P100 (a *Pascal* GPU):

```
$ nvcc -Xcompiler "-O3" -arch=sm_60 hello.cu -o gpu_hello.x
```

Sample Batch Script for GPU Jobs on Hummingbird

Batch script *gpu.slurm:*

```
#!/bin/bash -l

#SBATCH -J gpu
#SBATCH -p 96x24gpu4
#SBATCH --gres=gpu:p100:1
#SBATCH --mem=24000
#SBATCH -t 1:00:00
#SBATCH --mail-user=shaw@ucsc.edu
#SBATCH --mail-type=ALL

module load cuda
./gpu_hello.x
```

Comments:

```
### your favorite shell

### job name
### job queue
### request 1 P100 GPU
### and 24GB host memory
### request 1 hour walltime
### ask SLURM to send emails
### when jobs aborts, starts and ends

### load the module for CUDA Toolkit
### run your CUDA executable
```

To submit the job:

```
sbatch gpu.slurm
```

A Sample CUDA program implementing
BLAS level-1 function *SAXPY*

A * X + Y

where
X & Y are vectors
A is a scalar

```
#define N 20480
// declare the kernel
__global__ void saxpy(float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) { y[i] += a*x[i]; }
}
int main(void) {
    float *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(float);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory
    cudaMemcpy(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    saxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpy(y, dy, size, cudaMemcpyDeviceToHost);
    // deallocate device memory
    cudaFree(dx); cudaFree(dy);
}
```

CUDA C Extensions

CUDA Runtime Functions

```
#define N 20480
// declare the kernel
__global__ void saxpy(float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) { y[i] += a*x[i]; }
}
int main(void) {
    double *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(float);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory
    cudaMemcpy(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    saxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpy(y, dy, size, cudaMemcpyDeviceToHost);
    // deallocate device memory
    cudaFree(dx); cudaFree(dy);
}
```

Kernels

- A kernel run N times in parallel by N different *CUDA threads*
- Defined using the __global__ (2 underscores before *and* after **global**) declaration specifier (must return *void*)
- Launched using the <<<…>>> *execution configuration syntax*
- Kernel Launches are ***asynchronous*** with respect to the host
  - They return immediately!

Example:

```
__global__ void saxpy(double a, double *x, double *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        y[i] += a*x[i];
    }
}

    saxpy<<<N/256,256>>>(a, dx, dy);
```

Kernel Execution Configuration

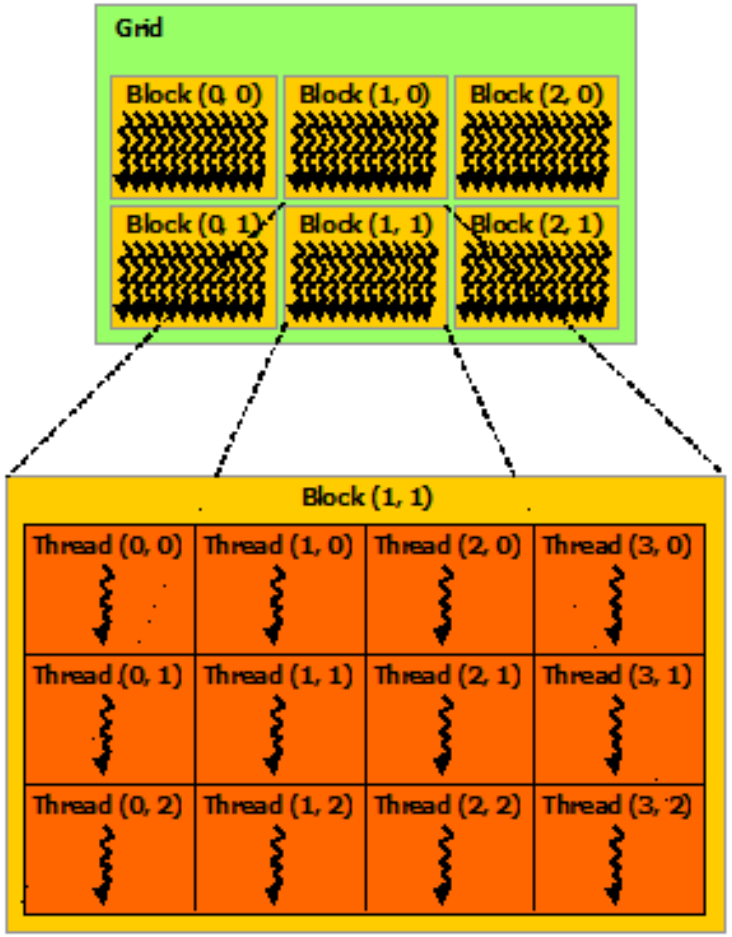
- A kernel function must be called with an execution configuration
- The execution configuration is an expression of the form <<< Dg, Db, Ns, S >>>, where
  - Dg specifies the dimension and size of the **grid** (of type dim3)
  - Db specifies the dimension and size of each **block** (of type dim3)
  - Ns specifies the size of shared memory allocated per block (optional)
  - S specifies the the associated stream (optional)

```
__global__ void KernelFunc(...);

KernelFunc<<< Dg, Db, Ns, S >>> (...);
```

Thread Hierarchy

A **Grid** is a collection of **Threads**. Threads in a Grid execute a *Kernel Function* and are divided into Thread **Blocks**.

- Threads within a block (running on the same SM) cooperate via shared memory, atomic operations and barrier synchronization
- Threads in different blocks cannot cooperate (maybe running on different SMs)
- Each grid is executed on one device



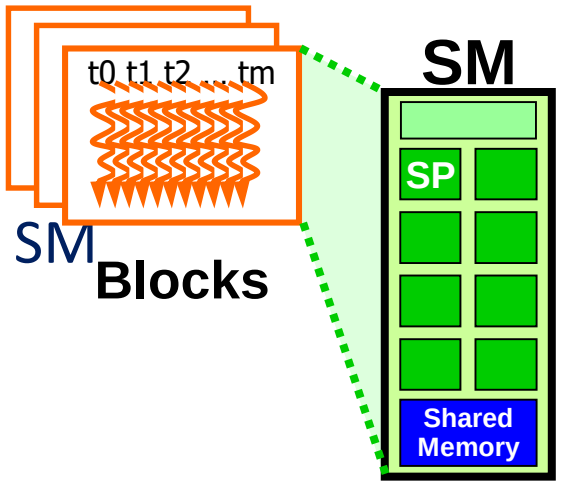
Note: In multidimensional blocks, the **x** thread index runs first, followed by the **y** thread index, and finally followed by the **z** thread index

Thread Block Size

- Between 256 and 512 threads per block is usually a good choice
  - Overly small blocks will limit the # of concurrent threads due to limitation on maximum # of concurrent blocks/SM
  - Overly large blocks can hinder performance, e.g., by increasing cost of any synchronization/barrier among all the threads in a block
- All CUDA-capable GPUs to date prefer # of threads per block to be a multiple of 32 if possible
  - 32 threads is the ***warp*** size. Non-multiples of 32 waste some resources and cycles
  - A multiple of 32 threads wide (x-dimension) facilitates coalesced memory access to adjacent memory addresses

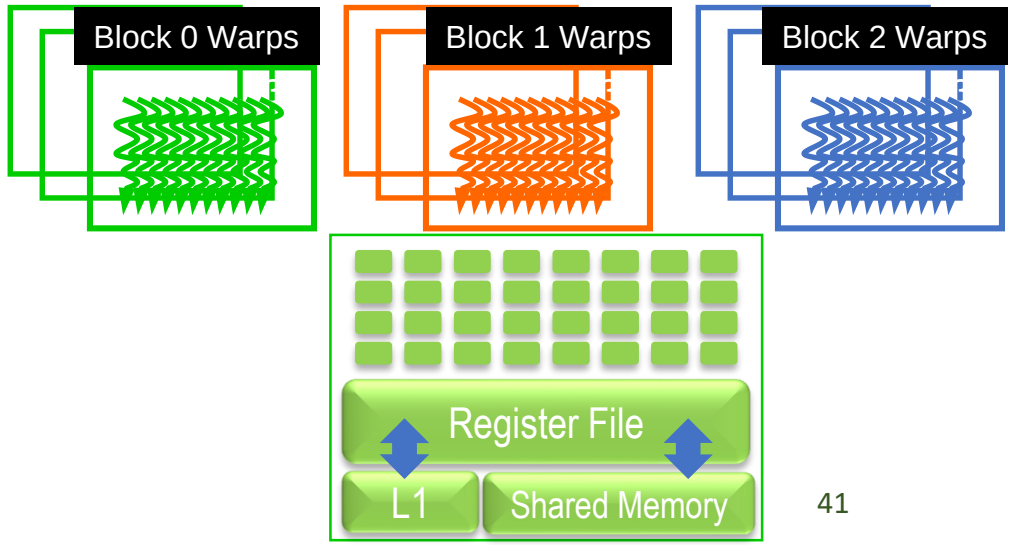
Executing Thread Blocks

- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Up to **16** blocks to each SM as resource allows
  - Both P100 and GTX 1080 Ti can take up to **2048** threads per SM
    - Could be 256 (threads/block) * 8 blocks
    - Or 512 (threads/block) * 4 blocks, etc.
- SM maintains thread/block indices
- SM manages/schedules thread execution
- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution based on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected



Warps as Execution Units

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD (32-way SIMD)
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps



Control Flow Divergence

```
__global__ void odd_even(int n, int* x) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( (i & 0x01) == 0 ) {
        x[i] = x[i] + 1;
    }
    else {
        x[i] = x[i] + 2;
    }
}
```

- Half the threads in the warp execute the if clause, the other half the else clause
- The system automatically handles **control flow divergence**, conditions in which threads within a warp execute different paths through a kernel
- Performance decreases with degree of divergence in warps

Built-in Variables

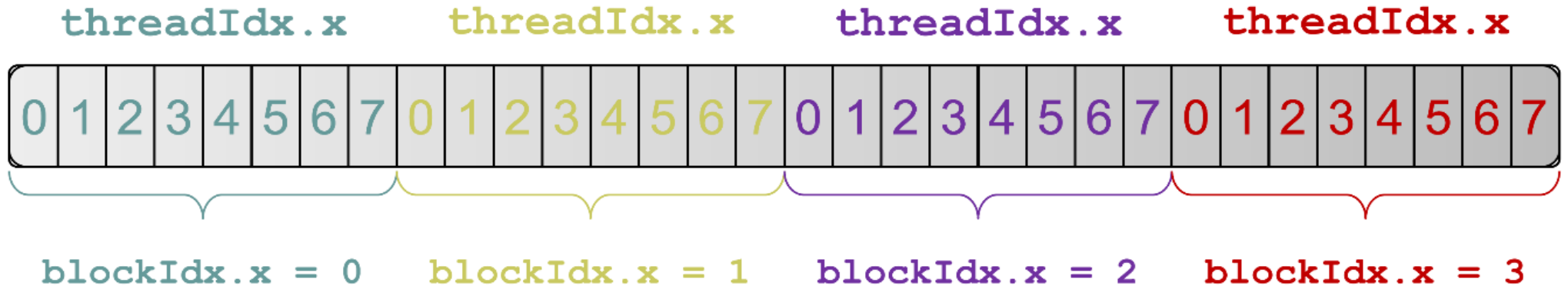
The following built-in variables specify the grid and block dimensions and block and threads indices:

- gridDim is of type dim3 and contains the *dimensions* of the **grid**
- blockIdx is of type uint3 and contains the **block** *index* within the **grid**
- blockDim is of type dim3 and contains the *dimensions* of the **block**
- threadIdx is of type uint3 and contains the **thread** *index* within the **block**
- warpSize is of type int and contains the warp size in threads

where
uint3: 3-component vector of uint. It is a structure, and the 1st, 2nd, and 3rd components are accessible through the fields x, y, and z, respectively.
dim3: integer vector type based on uint3. When defining a variable of type dim3, any component left unspecified is initialized to 1.

Example: Array Indexing

- Consider indexing into an array, one thread accessing one element
- Assume you launch with M=8 threads per block and the array is 32 entries long

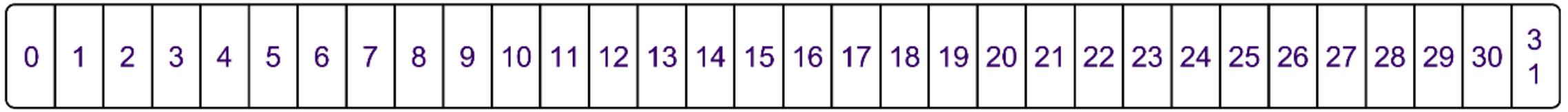


- With M threads per block, a unique index for each thread is given by

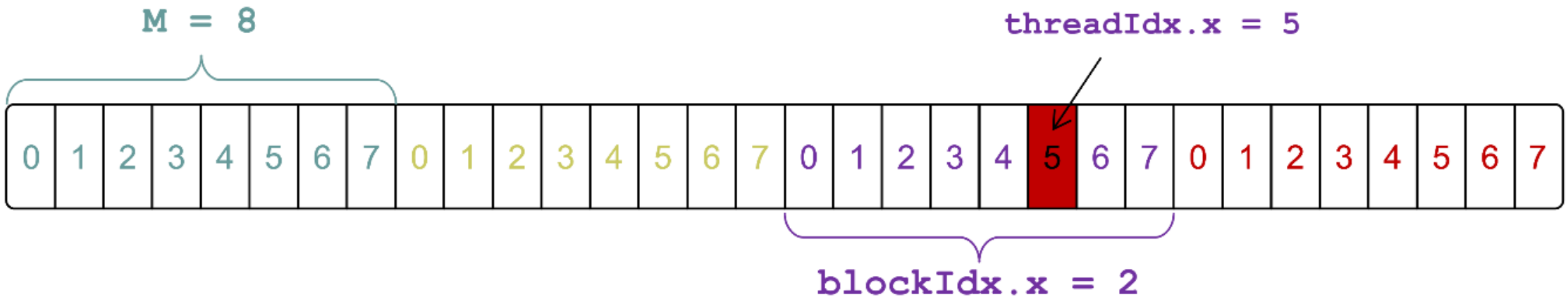
```
int index = threadIdx.x + blockIdx.x * M;
```

Where M = size of the block of threads; i.e., **blockDim.x**

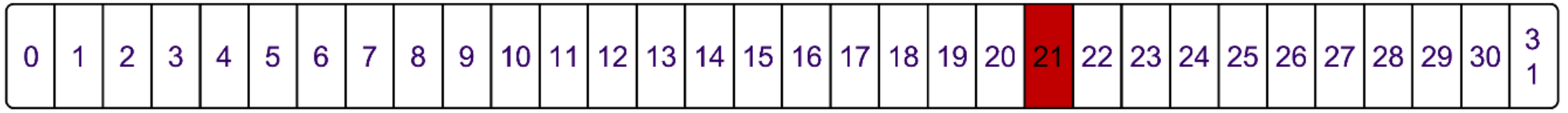
Example: Array Indexing



What is the array entry that thread of index 5 in block of index 2 will work on?

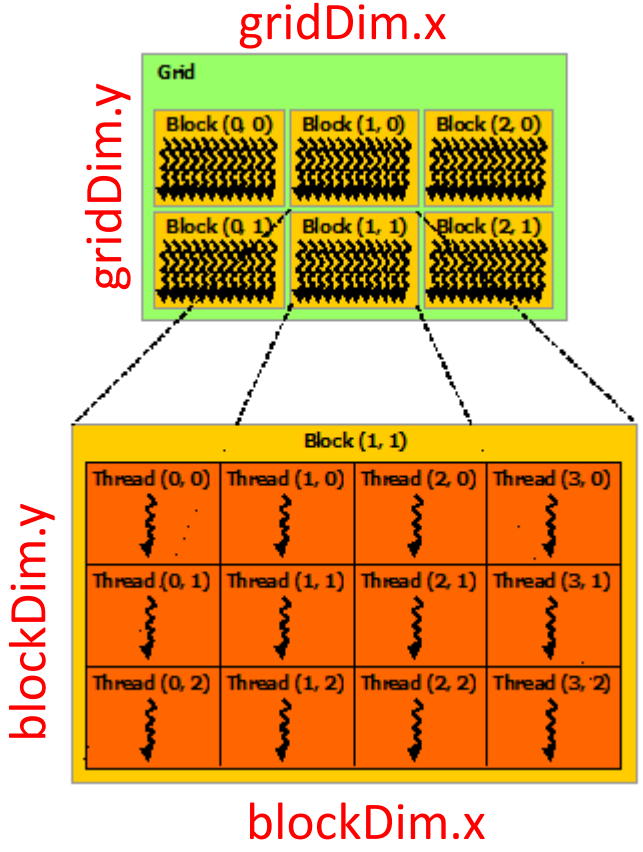



```
int index = threadIdx.x + blockIdx.x * blockDim.x;
          =      5      +     2      *      8
          = 21
```



2D Thread IDs

Each thread uses IDs to decide what data to work on

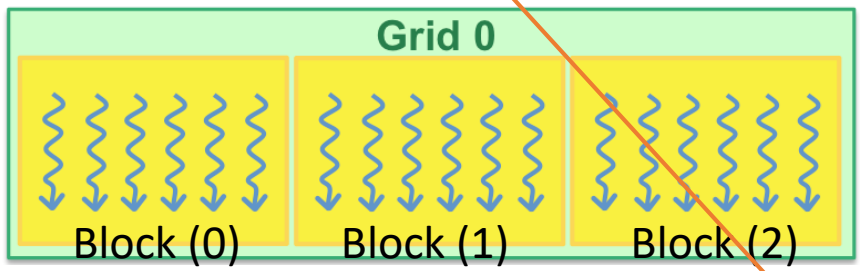



```
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main() {
    . . .
    dim3 threadsPerBlock(16, 16);
    dim3 blocksPerGrid(N/16, N/16);
    MatAdd<<<blocksPerGrid,threadsPerBlock>>>(A, B, C);
    . . .
}
```

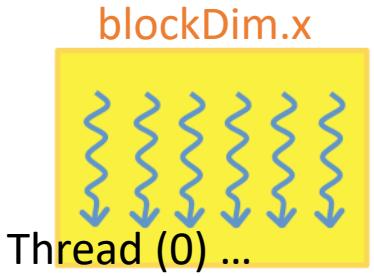
blockDim

gridDim

1D Grid: N/256 Blocks



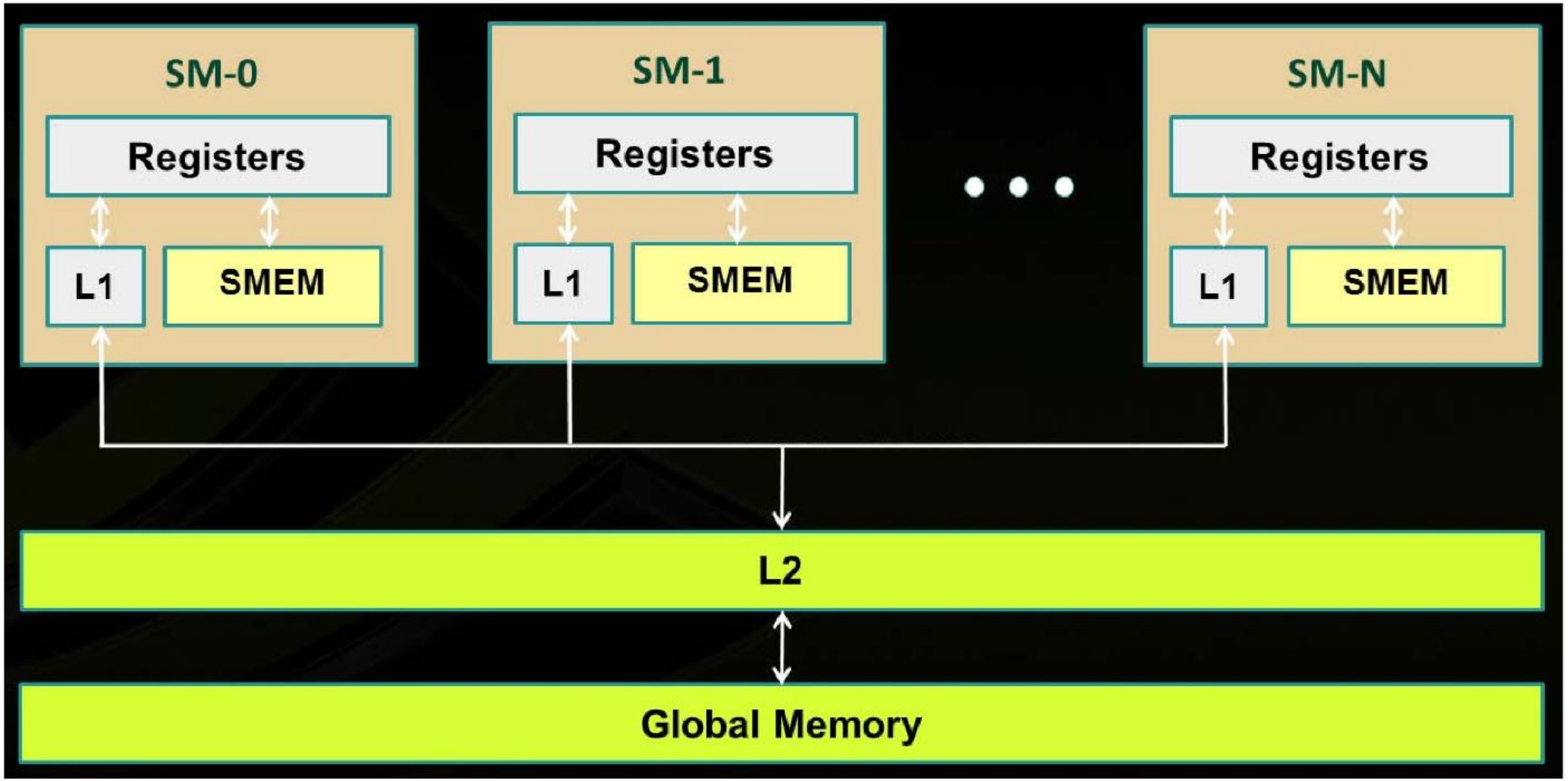
20480 Threads
80 Blocks
GTX 1080 has 20 SMs
up to 16 blocks per SM



1D Block: 256 Threads

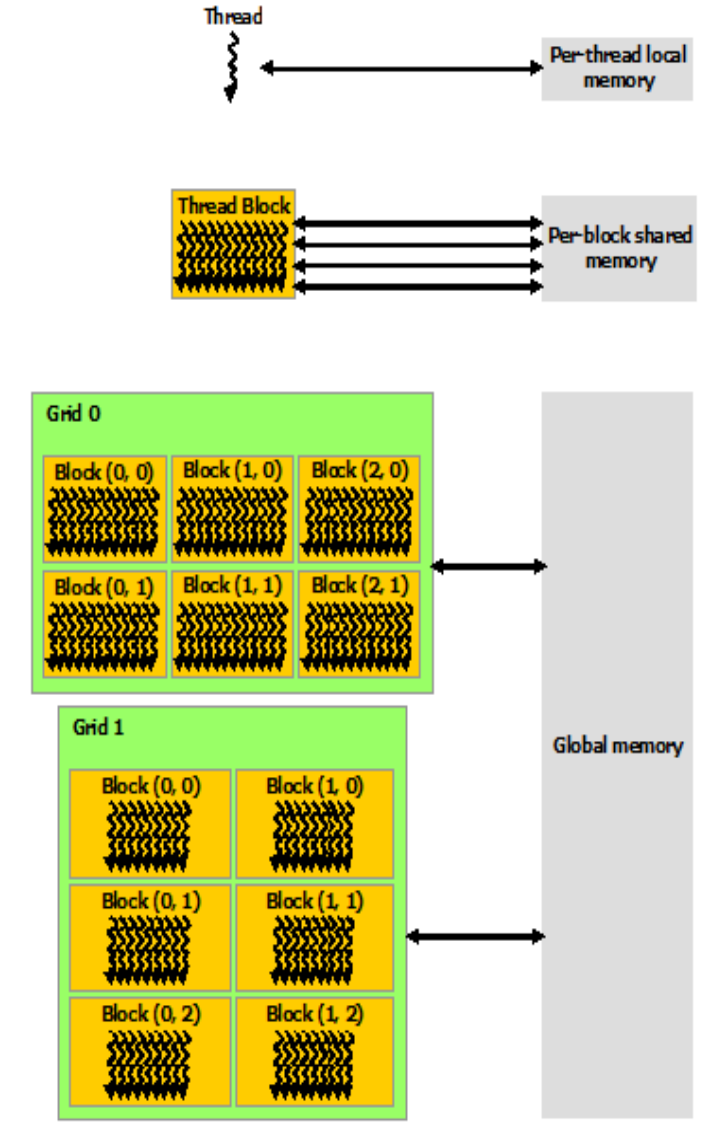
```
#define N 20480
// declare the kernel
__global__ void saxpy(float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) { y[i] += a*x[i]; }
}
int main(void) {
    double *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(float);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory
    cudaMemcpy(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    saxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpy(y, dy, size, cudaMemcpyDeviceToHost);
    // deallocate device memory
    cudaFree(dx); cudaFree(dy);
}
```

GPU Memory Layout



CUDA Memory Hierarchy

- Each thread has private local memory
- Each thread block has shared memory visible to all threads of the block
- All threads have access to the global memory
- All threads have access to the read-only constant and texture memory (not covered in this lecture)



i: per-thread local memory, resides in registers

dx & dy: global memory, accessible by all threads

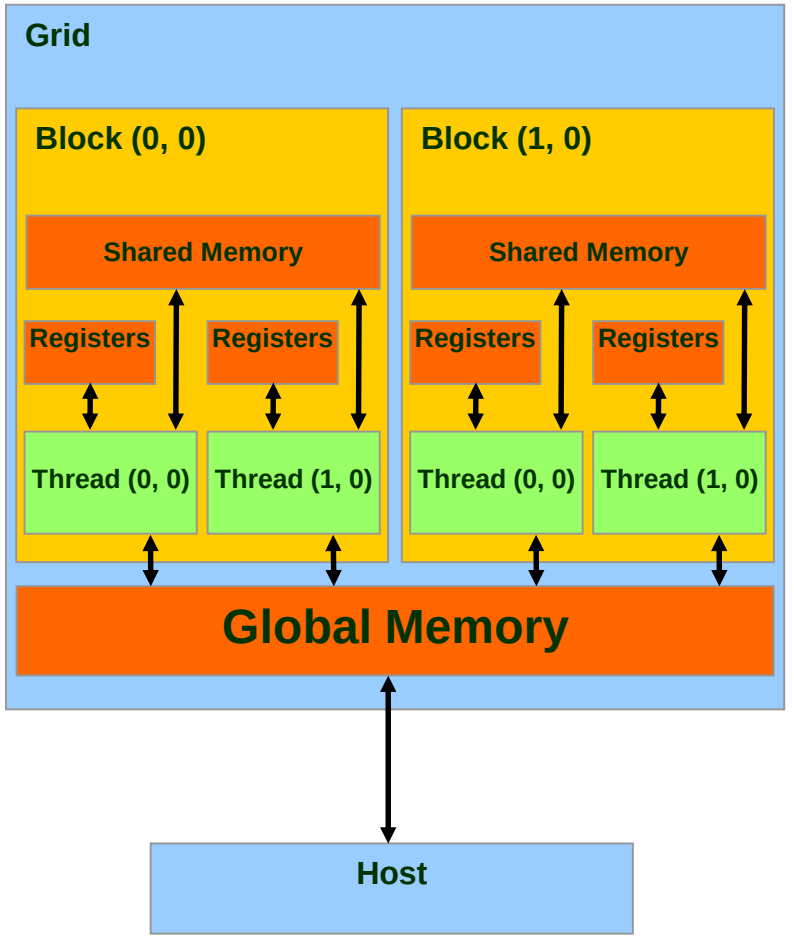
```
#define N 20480
// declare the kernel
__global__ void saxpy(float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) { y[i] += a*x[i]; }
}
int main(void) {
    double *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(float);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory
    cudaMemcpy(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    saxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpy(y, dy, size, cudaMemcpyDeviceToHost);
    // deallocate device memory
    cudaFree(dx); cudaFree(dy);
}
```

Global Memory

- Main means of data transfer between host and device
- Contents visible to all threads
- *Long* latency access
- All global memory accesses are cached (Fermi and later). A cache line is 128 bytes.

128 bytes = 32 (warp size) x 4 bytes

- To use bandwidth effectively, when threads loads, they should:
  - present a set of unit strided loads (**contiguous** dense accesses)
  - Keep sets of loads **aligned** to vector boundaries



CUDA Device Memory Allocation

cudaMalloc : Allocates object in the device **Global Memory**

```
cudaError_t cudaMalloc(void ** devPtr,
                       size_t size)
```

cudaFree : Frees object from device **Global Memory**

```
cudaError_t cudaFree(void * devPtr)
```

Example:

```
float *x, *y, a, *dx, *dy;
size_t size = N*sizeof(float)
// allocate device memory
cudaMalloc((void **) &dx, size);
cudaMalloc((void **) &dy, size);
// deallocate device memory
cudaFree(dx);
cudaFree(dy);
```

Error Handling

In practice, you should check for API errors in host code

http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__ERROR.html

```
cudaError_t err = cudaMalloc((void **) &dx, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n", cudaGetErrorString(err),
          __FILE__, __LINE__);
   exit(EXIT_FAILURE);
}
```

CUDA Error Handling Macro

```
#include <stdio.h>
#include <assert.h>

inline cudaError_t checkCuda(cudaError_t result)
{
  if (result != cudaSuccess) {
    fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
    assert(result == cudaSuccess);
  }
  return result;
}

int main()
{

/*
 * The macro can be wrapped around any function returning
 * a value of type `cudaError_t`.
 */

  checkCuda( cudaDeviceSynchronize() )
}
```

CUDA Device Memory Allocation (cont'd)

cudaMallocPitch : Allocates 2D arrays on the device

```
cudaError_t cudaMallocPitch(void **  devPtr,
                            size_t * pitch,
                            size_t   width,
                            size_t   height)
```

cudaMalloc3D : Allocates 3D arrays on the device

```
cudaError_t cudaMalloc3D(struct cudaPitchedPtr * pitchedDevPtr,
                         struct cudaExtent       extent)
```

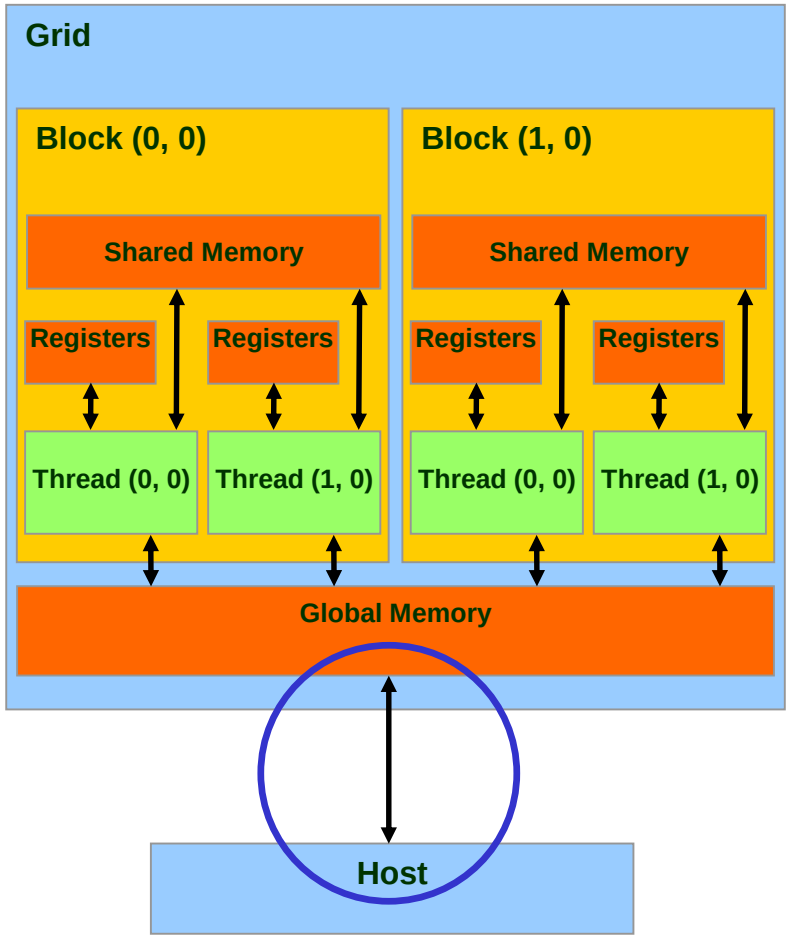
Recommended for allocating 2D and 3D objects. The allocation is appropriately padded to meet the alignment requirement.

CUDA Host-Device Data Transfer

```
cudaError_t cudaMemcpy(void *          dst,
                       const void *    src,
                       size_t          count,
                       cudaMemcpyKind  kind)
```

cudaMemcpy *synchronously* copies count bytes from src to dst

- kind is one of
  - **cudaMemcpyHostToHost**
  - **cudaMemcpyHostToDevice**
  - **cudaMemcpyDeviceToHost**
  - **cudaMemcpyDeviceToDevice**
- *Asynchronous* transfer: **cudaMemcpyAsync**
- PCIe 2 x16 peak bandwidth = 8 GB/s
- PCIe 3 x16 peak bandwidth = 15.75 GB/s

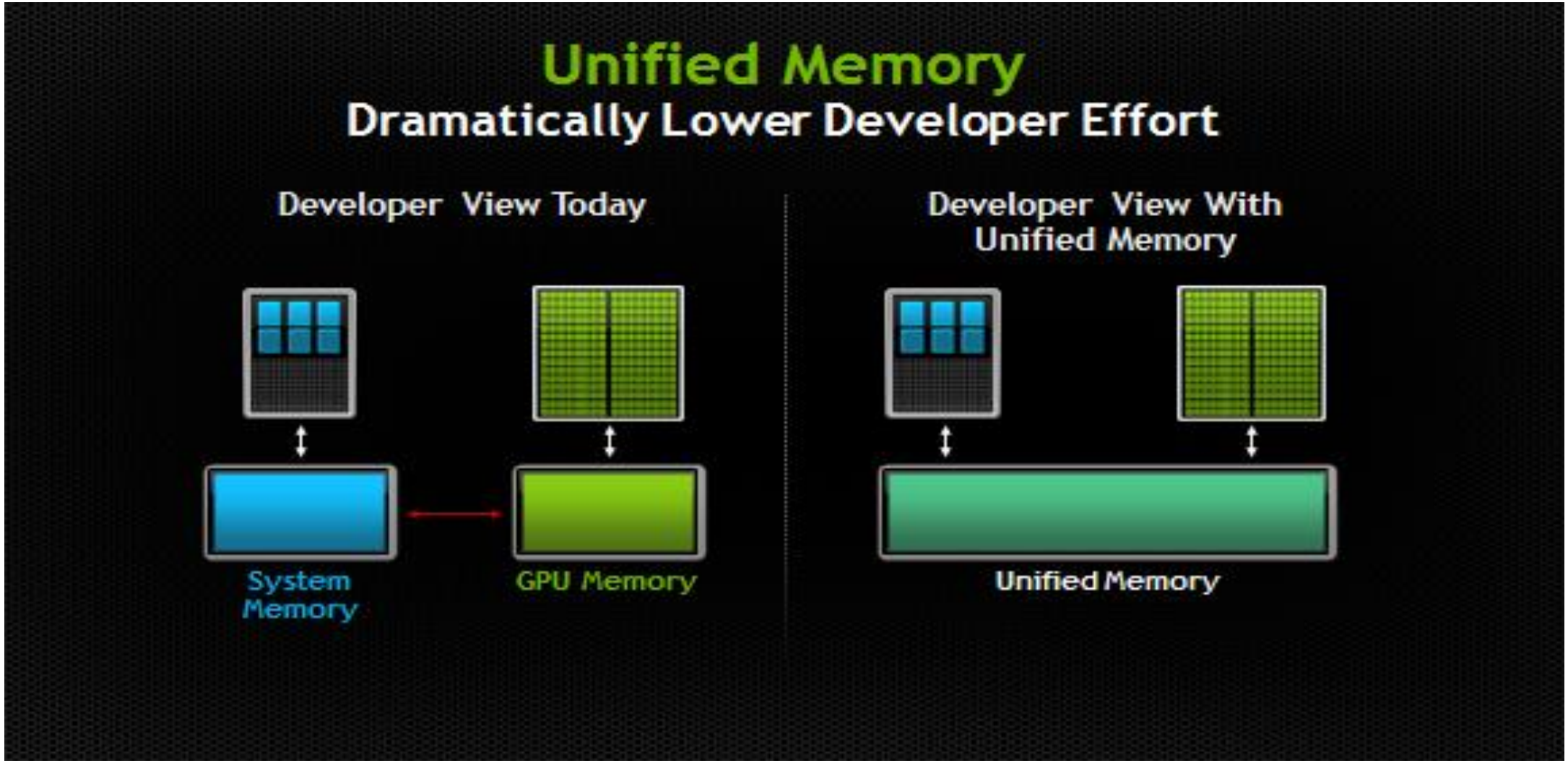


http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

CUDA Host-Device Data Transfer example

```
#define N 20480
__global__ void saxpy(float a, float *x, float *y) {
...  }
int main(void) {
    double *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(float);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory
    cudaMemcpy(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    daxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpy(y, dy, size, cudaMemcpyDeviceToHost);
    // deallocate device memory
    cudaFree(dx); cudaFree(dy);
}
```

Unified Memory in CUDA 6



Unified Memory

- Both CPUs and GPUs see a single coherent memory image with a common address space
- Unified Memory eliminates the need for explicit data movement via cudaMemcpy*
- cudaMallocManaged allocates memory that will be automatically managed by the Unified Memory system

```
cudaError_t cudaMallocManaged(void ** devPtr,
                              size_t  size,
                            unsigned int flags)
```

Without Unified Memory

```
int main(void) {
    double *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(double);
    // allocate host memory
    x = (double *)malloc(size);
    y = (double *)malloc(size);
    // assign some values to x and y
    // allocate device memory
    cudaMalloc(&dx, size);
    cudaMalloc(&dy, size);
    // copy from host to device
    cudaMemcpy(dx, x, size,
               cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size,
               cudaMemcpyHostToDevice);
    // launch the kernel function
    daxpy<<<N/256,256>>>(a, dx, dy);
    // copy from device to host
    cudaMemcpy(y, dy, size,
               cudaMemcpyDeviceToHost);
    // deallocate memory
    cudaFree(dx); cudaFree(dy);
    free(x); free(y);
}
```

With Unified Memory

```
#define N 20480

int main(void) {
    double *x, *y, a;
    size_t size = N*sizeof(double)

    // allocate unified memory
    cudaMallocManaged(&x, size);
    cudaMallocManaged(&y, size);
    // assign some values to x and y

    // launch the kernel function
    daxpy<<<N/256,256>>>(a, x, y);

    cudaDeviceSynchronize();

    for(int i=0; i<N; i++)
      printf("y[%d] = %e\n", i, y[i]);

    // deallocate memory
    cudaFree(x); cudaFree(y);
}
```

Kernel launches are asynchronous

More on Unified Memory

- Unified Memory in CUDA 6, *by* Mark Harris
- Unified Memory for CUDA Beginners, *by* Mark Harris
- Maximizing Unified Memory Performance in CUDA, by Nikolay Sakharnykh

CUDA Variable Type Qualifier

	resides in	lifetime	accessible from
__device__	global memory	application	all threads and hosts
__constant__	constant memory	application	all threads and hosts
__shared__	shared memory	thread block	threads within the block

- An automatic variable declared in device code generally resides in a register
  - Except that per-thread arrays reside in global memory!
- The **__constant__** or **__shared__** qualifier can be optionally used together with __device__
- A **__device__** variable can be additionally qualified with **__manage__**. Such a variable can be directly referenced from host code (*Unified Memory* access).
  - Another way to allocate *Unified Memory* is to use **cudaMallocManaged()**.

CUDA Function Type Qualifiers

Function Type Qualifier	Executed on	Callable from
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host & device
__host__ float HostFunc()	host	host

- Note: each "__" consists of two underscore characters!
- **__device__** and **__host__** can be used together
- **__host__** is optional if used alone
- **__global__** defines a *kernel* function (a kernel run *N* times in parallel by *N* different CUDA threads)
  - A kernel function must return **void**
  - A call to a **__global__** function is *asynchronous*, which returns before the device has completed its execution

Host Runtime API

- Provides functions to deal with:
  - Device management (including multi-device systems)
    - e.g., on multi-device system, a host thread can set the device it operates on at any time by calling cudaSetDevice()
  - Memory management
  - Error handling
  - etc.
- There is no explicit initialization function for the runtime; it initializes the first time a runtime function is called

http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

CUDA Device Management Functions

- **cudaGetDeviceCount :** returns the number of compute-capable devices
- **cudaSetDevice :** sets device to be used for GPU execution
- **cudaGetDevice :** returns which device is currently being used
- **cudaGetDeviceProperties :** returns information about the compute-device
- **cudaDeviceGetAttribute :** returns device attribute value
- **cudaSetDeviceFlags :** sets flags to be used for device executions
- **cudaGetDeviceFlags :** gets the flags for the current device
- **cudaDeviceReset :** destroys and cleans up all resources associated with the current device in the current process
- **cudaDeviceSynchronize :** blocks until the device has completed all **preceding** requested tasks
- etc.

  http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html

Example: Querying Properties of CUDA Devices

```
#include <cuda_runtime.h>
#include <cuda.h>
int main(void) {
    int deviceCount = 0;
    cudaError_t error_id = cudaGetDeviceCount(&deviceCount);
    if (error_id != cudaSuccess) {...}
    ...
    for (dev = 0; dev < deviceCount; ++dev)
    {
        cudaSetDevice(dev);
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, dev);
        printf("\nDevice %d: \"%s\"\n", dev, deviceProp.name);
        printf("  Warp size:                                     %d\n",
               deviceProp.warpSize);
        printf("  Maximum number of threads per multiprocessor:  %d\n",
               deviceProp.maxThreadsPerMultiProcessor);
        ...
    }
    exit(EXIT_SUCCESS);
}
```

Example: Multi-GPU using OpenMP for threading

```
#include <omp.h>
#include <stdio.h>  // C++ streams aren't necessarily thread safe

// a simple kernel that simply increments each array element by b
__global__ void kernelAddConstant(int *a, const int b)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] += b;
}

int main(void) {
    int int num_gpus = 0;   // number of CUDA GPUs
    // determine the number of CUDA capable GPUs
    cudaGetDeviceCount(&num_gpus);
    // initialize data
    unsigned int n = num_gpus * 8192;
    unsigned int nbytes = n * sizeof(int);
    int *a = 0;     // pointer to data on the CPU
    int b = 3;      // value by which the array is incremented
    a = (int *)malloc(nbytes);
    for (unsigned int i = 0; i < n; i++) a[i] = i;
```

Example: Multi-GPU using OpenMP for threading (cont'd)

```
// run as many CPU threads as there are CUDA devices
omp_set_num_threads(num_gpus);
#pragma omp parallel
{
    unsigned int cpu_thread_id = omp_get_thread_num();
    unsigned int num_cpu_threads = omp_get_num_threads();
    // set and check the CUDA device for this CPU thread
    int gpu_id = -1;
    cudaSetDevice(cpu_thread_id % num_gpus);
    cudaGetDevice(&gpu_id);
    // pointer to this CPU thread's portion of data
    int *sub_a = a + cpu_thread_id * n / num_cpu_threads;

    // pointer to memory on the device associated with this CPU thread
    int *d_a = 0;
    unsigned int nbytes_per_kernel = nbytes / num_cpu_threads;
    cudaMalloc((void **)&d_a, nbytes_per_kernel);
    cudaMemset(d_a, 0, nbytes_per_kernel);
    cudaMemcpy(d_a, sub_a, nbytes_per_kernel, cudaMemcpyHostToDevice));
```

Example: Multi-GPU using OpenMP for threading (cont'd)

```
        dim3 gpu_threads(128);  // 128 threads per block
        dim3 gpu_blocks(n / (gpu_threads.x * num_cpu_threads));
        kernelAddConstant<<<gpu_blocks, gpu_threads>>>(d_a, b);

        cudaMemcpy(sub_a, d_a, nbytes_per_kernel, cudaMemcpyDeviceToHost);
        cudaFree(d_a);
    }

    free(a); // free CPU memory

    exit(EXIT_SUCCESS);
}
```

Common Runtime Component: Mathematical Functions

- **pow, sqrt, cbrt, hypot**
- **exp, exp2, expm1**
- **log, log2, log10, log1p**
- **sin, cos, tan, asin, acos, atan, atan2**
- **sinh, cosh, tanh, asinh, acosh, atanh**
- **ceil, floor, trunc, round**
- etc. - http://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH.html
  - Available in both *device* and *host* code
  - Floating-point functions are overloaded for both single and double precisions (IEEE compliant) in *device* code
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

Device Runtime Component: Mathematical Functions

Some mathematical functions (e.g., **sin(x)**) have a less accurate, but faster device-only (direct mapping to hardware ISA) version (e.g., **__sin(x)**)

- **__pow**
- **__log, __log2, __log10**
- **__exp**
- **__sin, __cos, __tan**

The `-use_fast_math` compiler option forces every `func()` to compile to `__func()`

http://docs.nvidia.com/cuda/cuda-math-api/index.html

Make your program float-safe!

- Newer GPUs have double precision support
  - Double precision will have additional performance cost
  - Careless use of double or undeclared types may run more slowly
    - K20: FP64 performance = 1/3 FP32 performance
    - P100: FP64 performance = 1/2 FP32 performance
    - GTX 1080: FP64 performance = 1/32 FP32 performance
- Important to be float-safe (be explicit whenever you want single precision) to avoid using double precision where it is not needed
  - Add 'f' specifier on float literals:
    - `foo = bar * 0.123;  // double assumed`
    - `foo = bar * 0.123f;  // float explicit`
  - Use float version of standard library functions
    - `foo = sin(bar);  // double assumed`
    - `foo = sinf(bar);  // single precision explicit`

CUDA Deviations from IEEE-754

- Addition and multiplication are IEEE 754 compliant
  - Maximum 0.5 ulp (units in the least place) error
- Division accuracy is not fully compliant (2 ulp)
- Not all rounding modes are supported
- No mechanism to detect floating-point exceptions (yet)

Device Runtime Component: Synchronization Functions

- void **__syncthreads**()
- Synchronizes **all** threads in *a block*
- Variants
  - int **__syncthreads_count**(int predicate)
  - int **__syncthreads_and**(int predicate)
  - int **__syncthreads_or**(int predicate)
- Used to coordinate communication between the threads of the same block
- Used to avoid *read-after-write*, *write-after-read*, *write-after-write* hazards when accessing the same addresses in shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

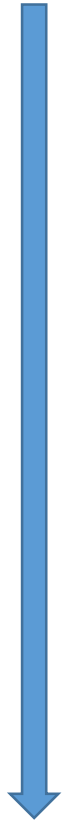
Device Runtime Component: Atomic Functions

- An atomic function performs a ***read-modify-write*** *atomic* operation on one 32-bit or 64-bit word residing in global or shared memory
- Atomic functions can only be used in *device* function
  - Atomic **Add**, **Sub**, **Exch** (exchange), **Min**, **Max**, **Inc**, **Dec**, **CAS** (compare and swap), **And**, **Or**, **Xor**
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomic-functions
- For example, **atomicAdd**:
  - int atomicAdd(int* address, int val);

    reads the 32-bit word **old** from the location pointed to by **address** in global or shared memory, computes (**old + val**), and stores the result back to memory at the same address. The function returns **old**.
  - unsigned int atomicAdd(unsigned int* address, unsigned int val);
  - float atomicAdd(float* address, float val);
  - double atomicAdd(double* address, double val);
  - etc.

Data Race without Atomic Operations

time



*address initialized to 0

Thread1:

old ← *address

new ← old + 1

*address ← new

Thread2:

old ← *address

new ← old + 1

*address ← new

- Both threads receive 0 in old
- *address becomes 1!

Key Concepts of Atomic Operations

- An ***atomic*** operation is a ***read-modify-write*** operation performed by a single hardware instruction on a memory location *address*
  - Read the old value, calculate a new value, and write the new value to the location
- The hardware ensures that no other threads can perform another ***read-modify-write*** operation on the same location until the current atomic operation is complete
  - Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue
  - All threads perform their atomic operations **serially** on the same location

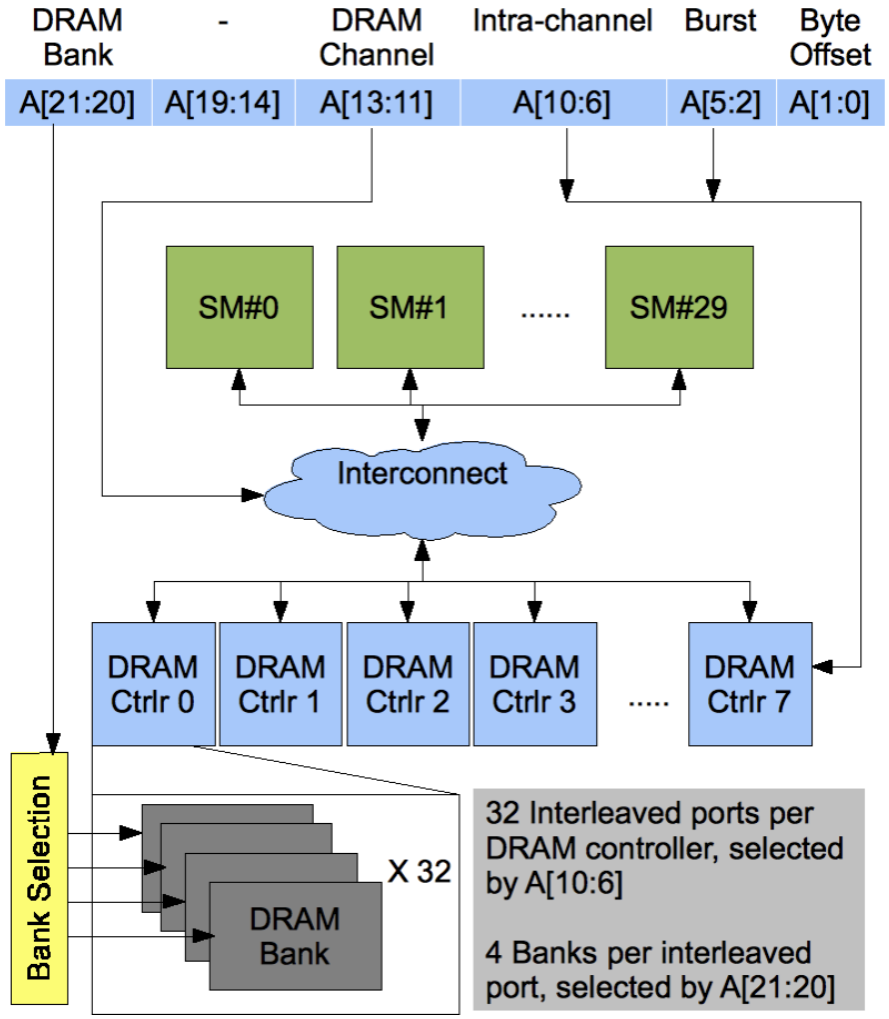
Example: A Basic Text Histogram Kernel

- The kernel receives a pointer to the input buffer of byte values
- Each thread process the input in a strided pattern

```
__global__ void histo_kernel(unsigned char *buffer,
            long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    // All threads handle blockDim.x * gridDim.x consecutive elements
    while (i < size) {
        int alphabet_position = buffer[i] - "a";
        if (alphabet_position >= 0 && alphabet_position < 26)
            atomicAdd(&(histo[alphabet_position/4]), 1);
        i += stride;
   }
}
```

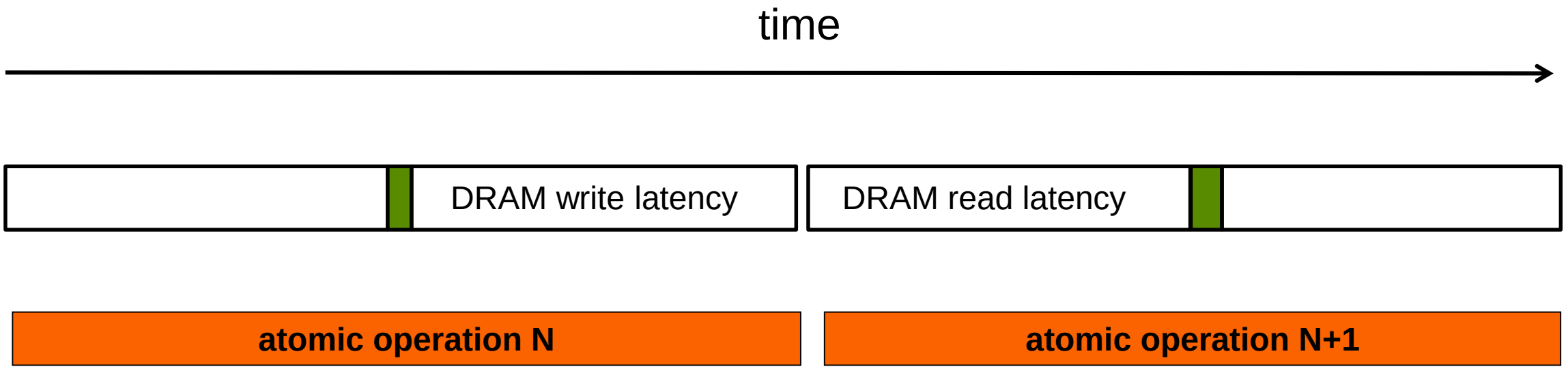
Atomic Operations on Global Memory (DRAM)

- An atomic operation on a DRAM location starts with a read, which has a latency of a few hundred cycles
- The atomic operation ends with a write to the same location, with a latency of a few hundred cycles
- During this whole time, no one else can access the location



Atomic Operations on DRAM

- Each Read-Modify-Write has two full memory access delays
  - —All atomic operations on the same variable (DRAM location) are *serialized*



Latency determines throughput

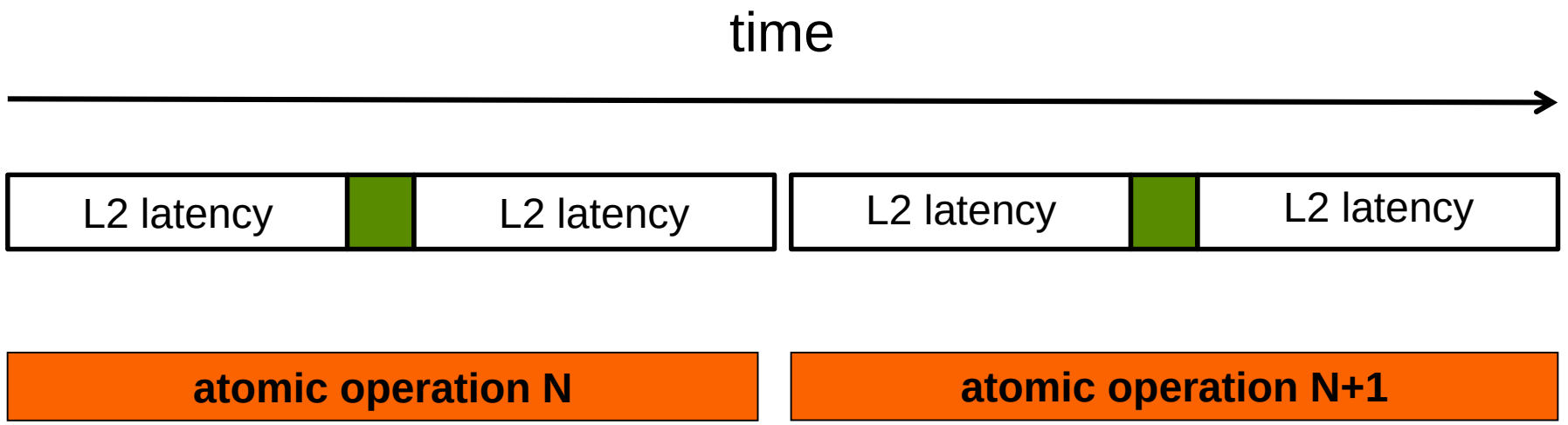
- Throughput of atomic operations on the same DRAM location is the rate at which the application can execute an atomic operation.

- The rate for atomic operation on a particular location is limited by the total latency of the read-modify-write sequence, typically more than 1000 cycles for global memory (DRAM) locations.

- This means that if many threads attempt to do atomic operation on the same location (contention), the memory throughput is reduced to < 1/1000 of the peak bandwidth of one memory channel!

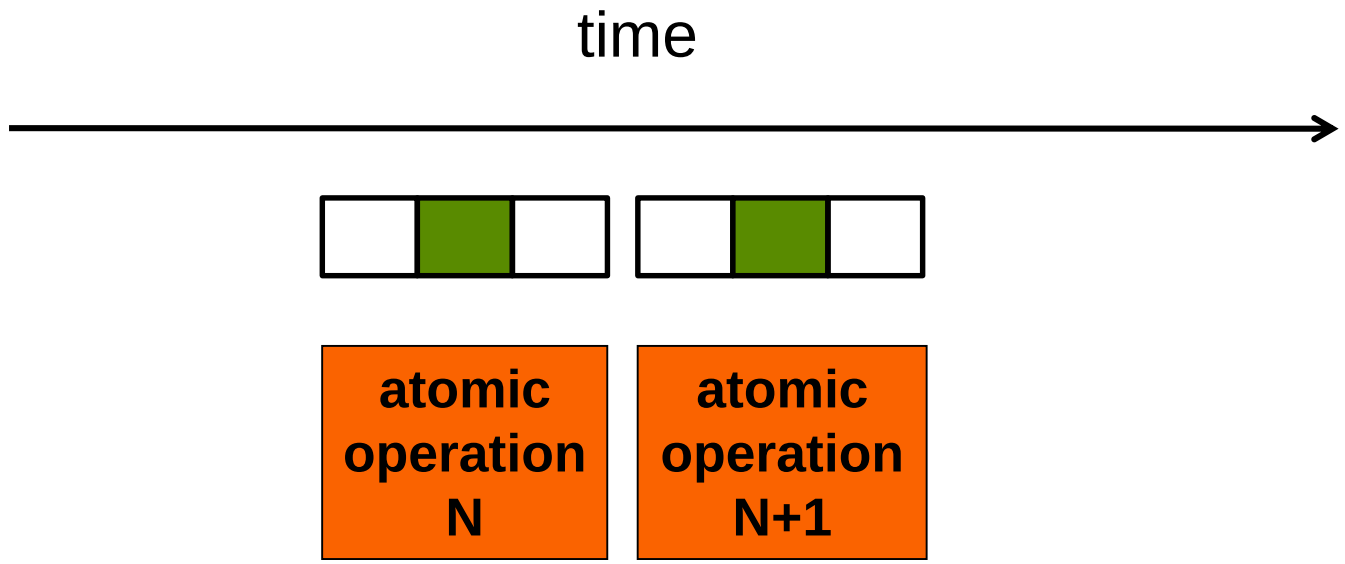
Hardware Improvements

- Atomic operations on Fermi (or later) L2 cache
  - Medium latency, about 1/10 of the DRAM latency
  - Shared among all blocks
  - “Free improvement” on Global Memory atomics



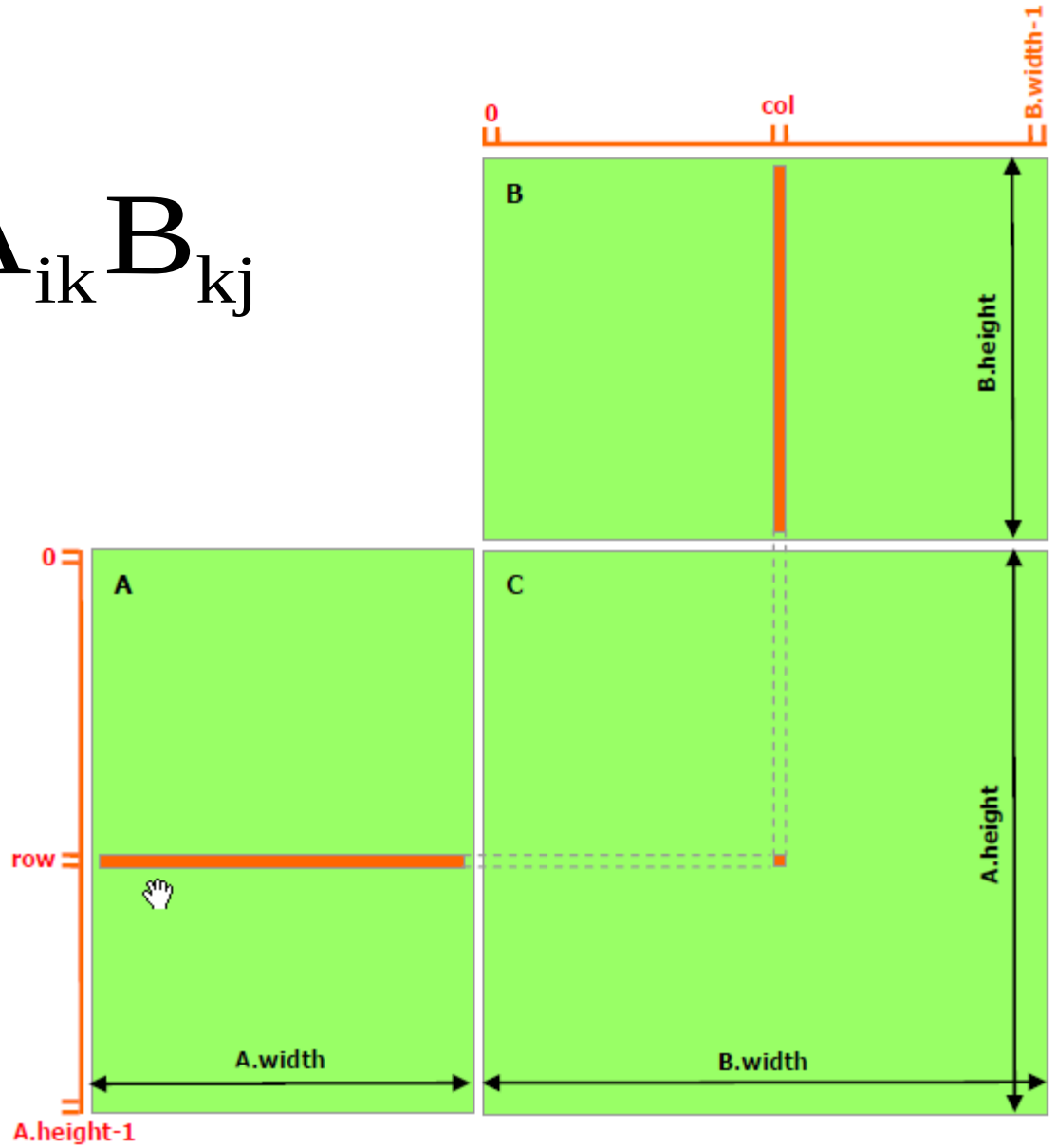
Atomic operations on Shared Memory

- Very short latency
- Private to each thread block
- Need algorithm work by programmers

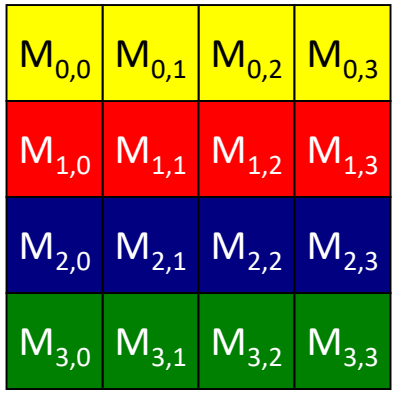


Case Study: Matrix Multiplication

$$C_{ij} = \sum_{k} A_{ik} B_{kj}$$



Memory Layout of a Matrix in C



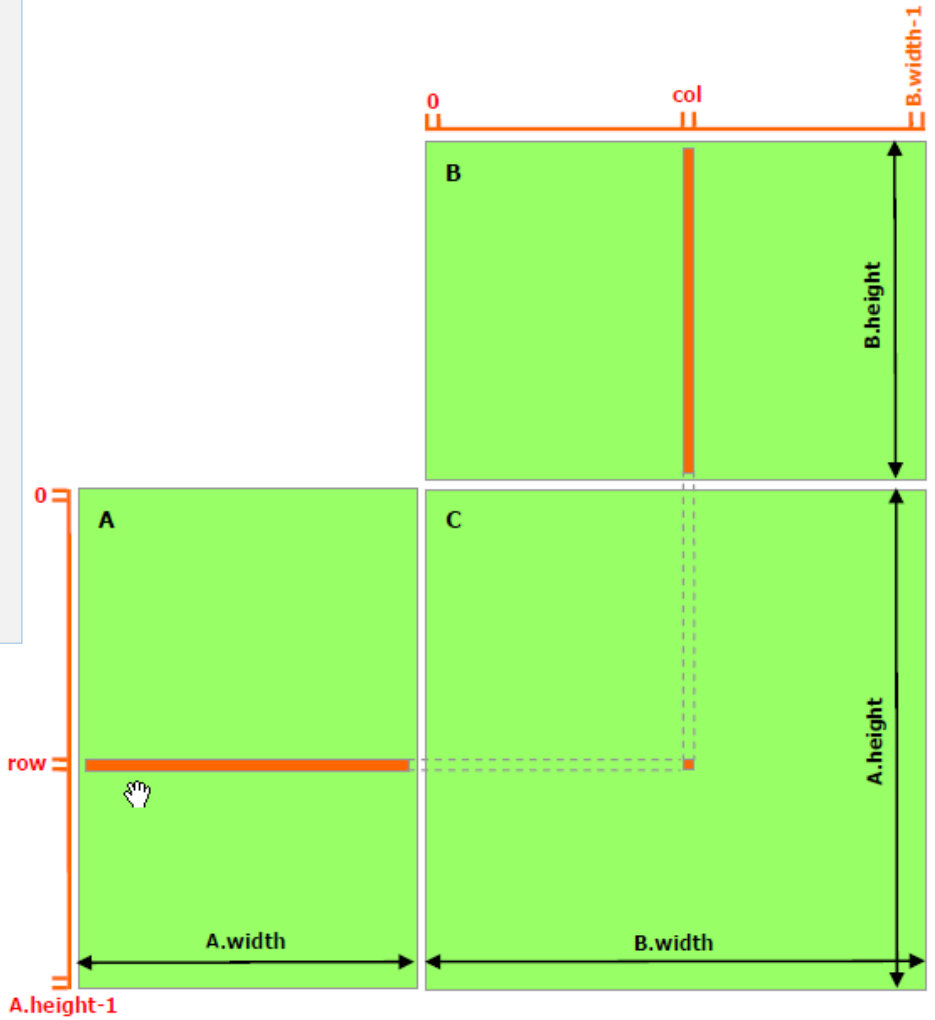

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

M

Matrix Multiplication: A Simple Host Version in C

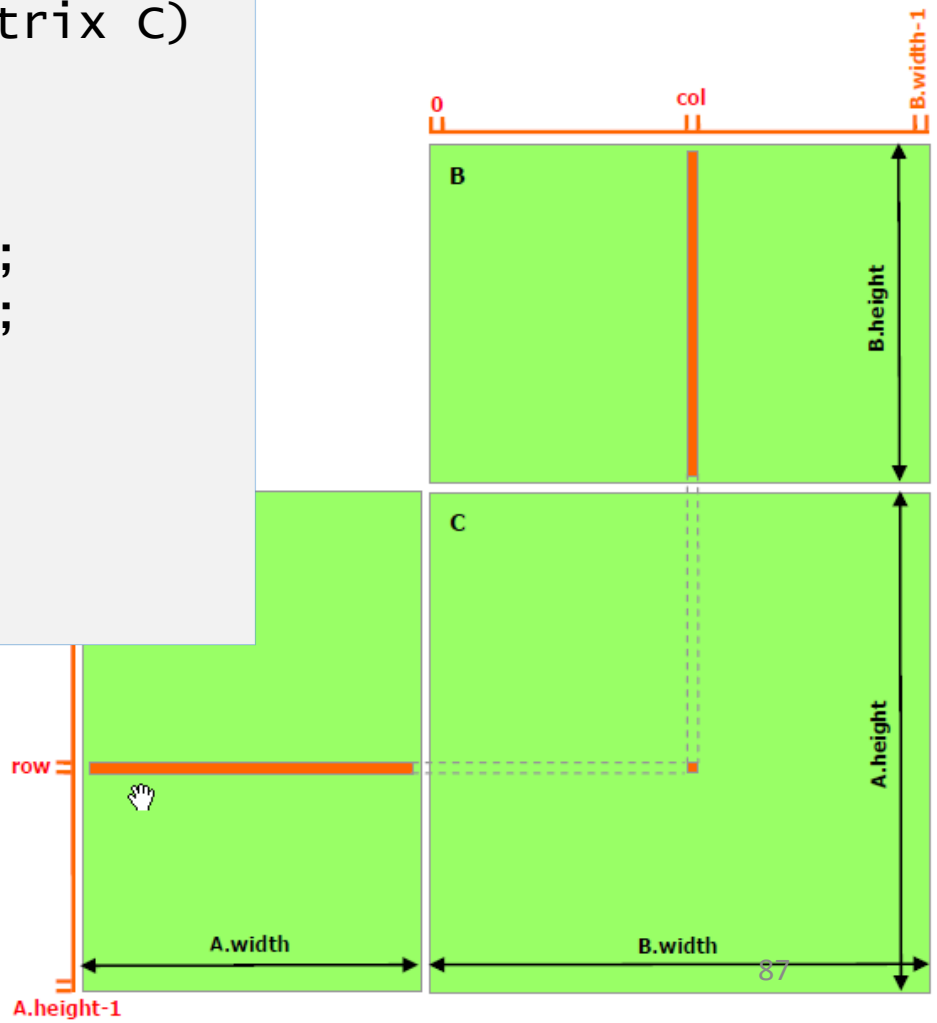
```
// Matrix multiplication on the host
void MatMulOnHost(Matrix A, Matrix B, Matrix C)
{
  for (int row = 0; row < A.height; ++row)
    for (int col = 0; col < B.width; ++col) {
      float Cvalue = 0;
      for (int k = 0; k < A.width; ++k)
        Cvalue += A.elements[row * A.width + k]
                  * B.elements[k * B.width + col];
      C.elements[row * C.width + col] = Cvalue;
    }
}
```

Simple, but unoptimized!
Why?



Matrix Multiplication: A Simple CUDA Kernel

```
// Matrix multiplication kernel
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int k = 0; k < A.width; ++k)
        Cvalue += A.elements[i * A.width + k]
                  * B.elements[k * B.width + j];
    C.elements[row * C.width + col] = Cvalue;
}
```



Matrix Multiplication: Host Code

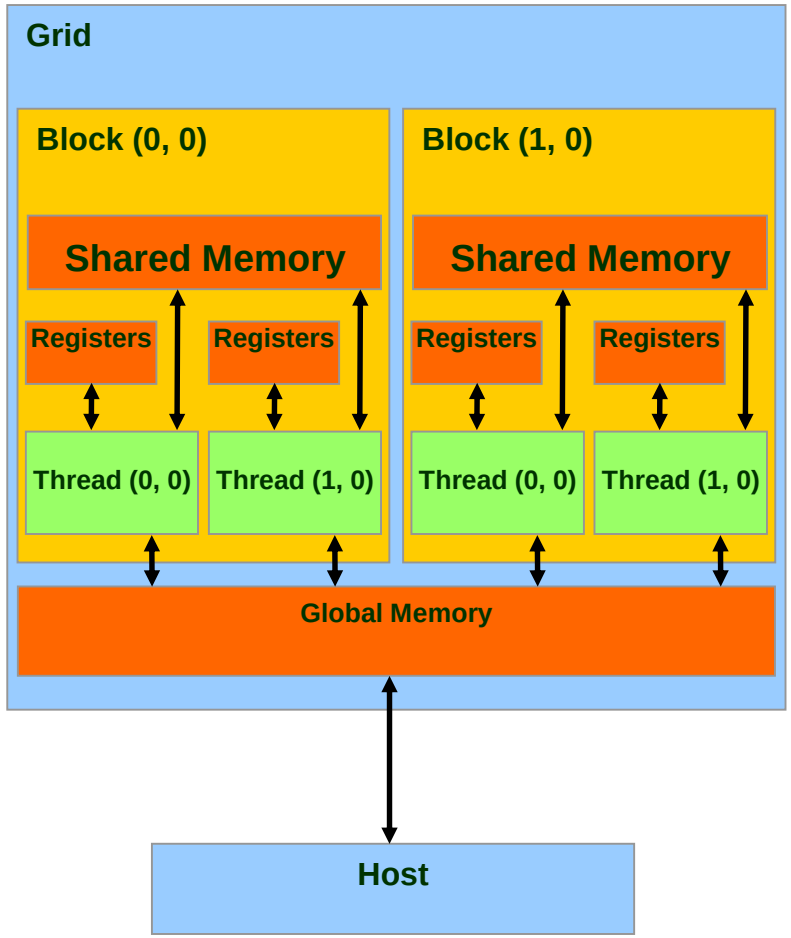
```
#define BLOCK_SIZE 16
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
    // Matrix d_B (skipped)
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A.elements); cudaFree(d_B.elements); cudaFree(d_C.elements);
}
```

Problems with the simple Implementation

- C = A * B
- Each thread computes one element of C
  - Each thread loads a row of A from global memory (repetitive and very expensive!)
  - Each thread loads a column of B from global memory (discontiguous access, even more expensive!)
  - Each thread performs one multiply and addition for each pair of A and B elements
  - Compute to off-chip memory access ratio close to 1:1 (memory bandwidth limited)
- In total, A is read *B.width* times and B is read *A.height* times from global memory

Recap: CUDA Memory Hierarchy

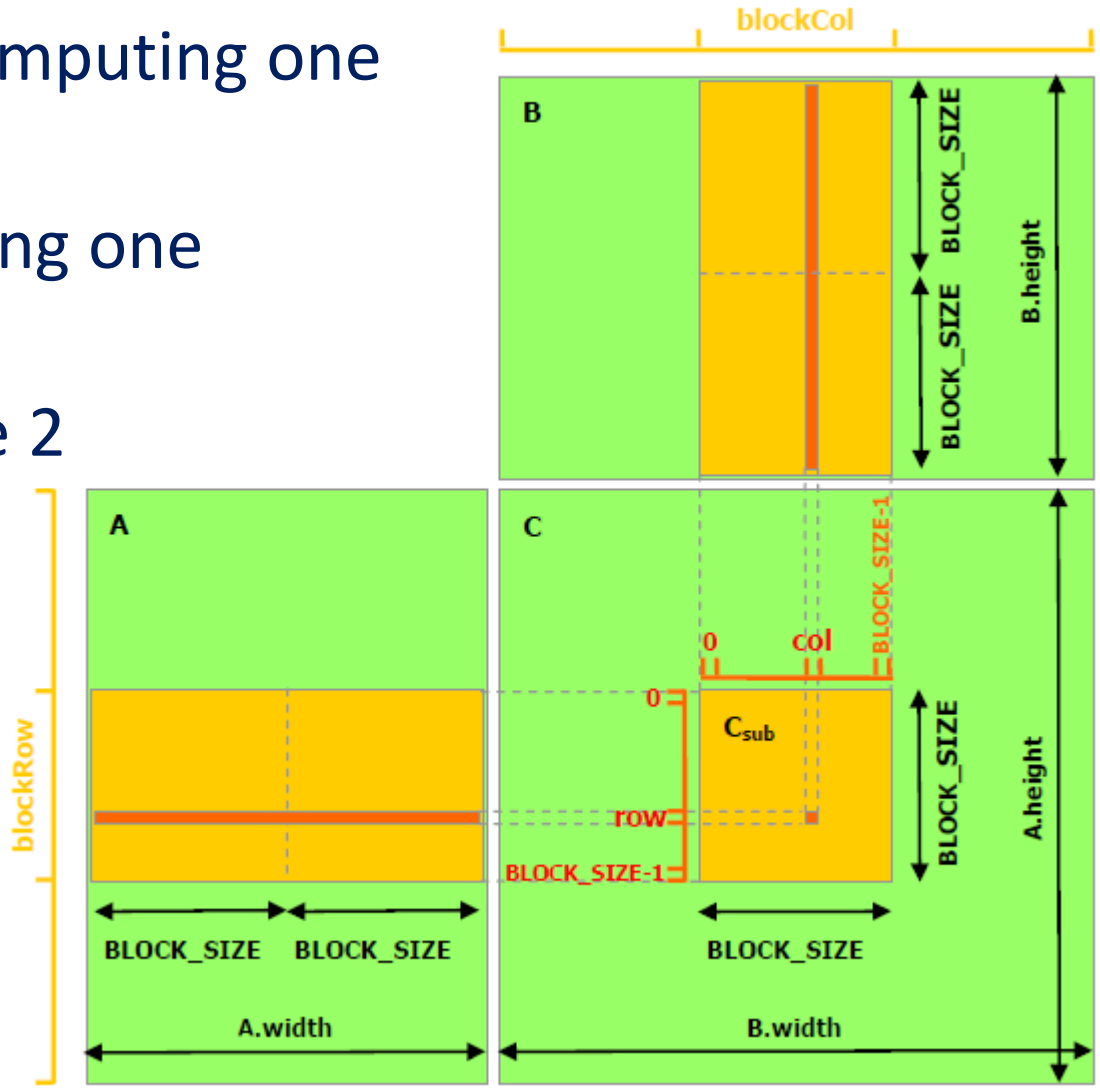
- Each thread has private local memory (fastest)
- Each thread block has shared memory visible to all threads of the block
  - Latency is an order of magnitude lower than global memory
  - Bandwidth is 4x-8x higher than global memory
- All threads have access to the global memory (slowest)
- All threads have access to the read-only constant and texture memory



Matrix Multiplication with Shared Memory

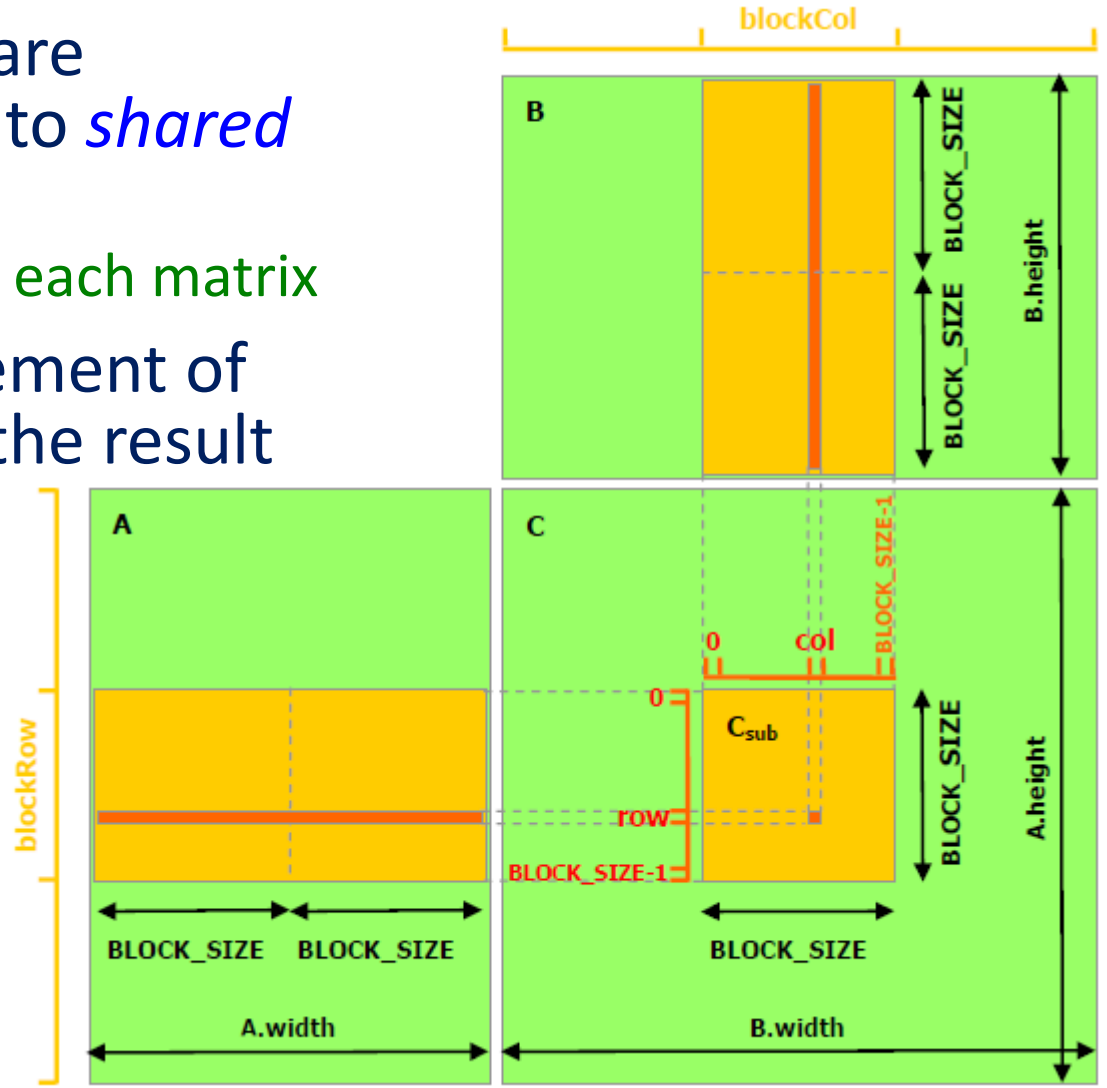
- Each thread block is responsible for computing one square sub-matrix C_{sub}
- Each thread is responsible for computing one element of C_{sub}
- To fit into the device's resources, these 2 rectangular sub-matrices of A & B are divided into square matrices of dimension *(block_size, block_size)*

$$C_{sub} = A_{(A.width,\ block\_size)} * B_{(block\_size,\ A.width)} = \Sigma\ A_{(block\_size,\ block\_size)} * B_{(block\_size,\ block\_size)}$$

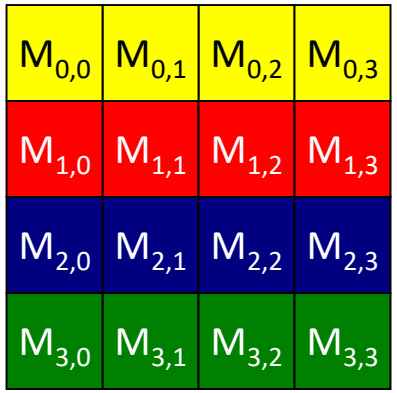


Matrix Multiplication with Shared Memory: 4 Steps

1. Load the 2 corresponding square matrices from *global memory* to *shared memory*
   – Each thread loads one element of each matrix
2. Each thread computes one element of the product; and accumulate the result into a *register*
3. Loop
4. Write final result to *global memory*



Revised Matrix Type




```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;
```

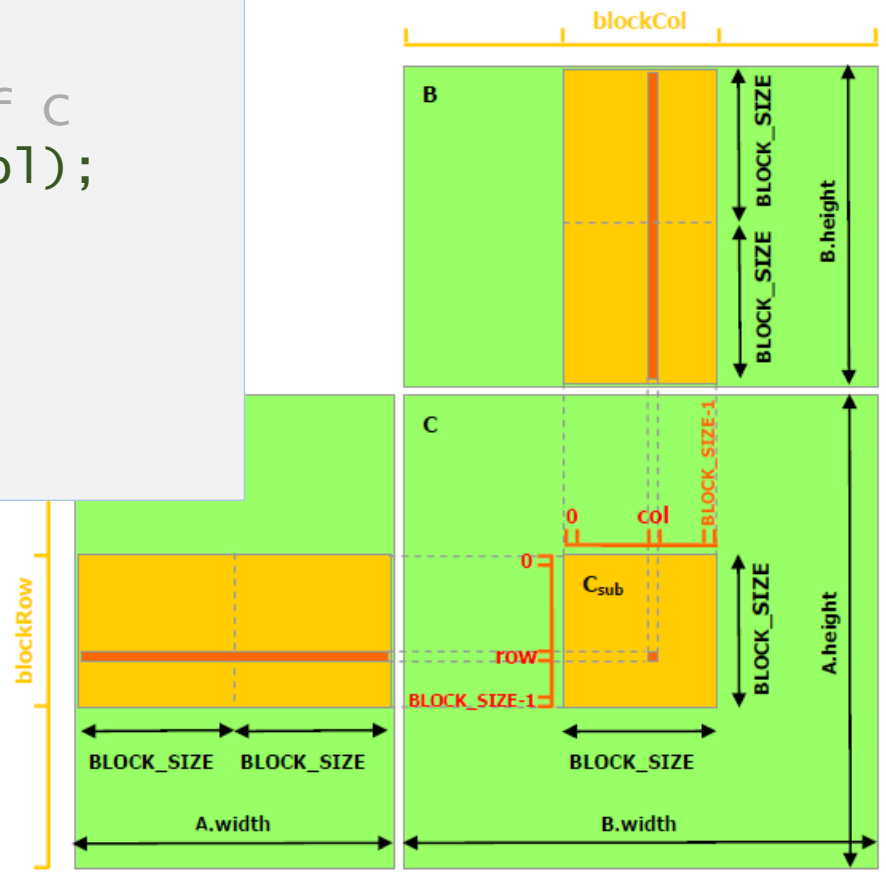
M

__device__ functions

```
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

__device__ void SetElement(Matrix A, int row, int col, float value)
{
    A.elements[row * A.stride + col] = value;
}

__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width  = BLOCK_SIZE;
    Asub.height = BLOCK_SIZE;
    Asub.stride = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                         + BLOCK_SIZE * col];
    return Asub;
}
```

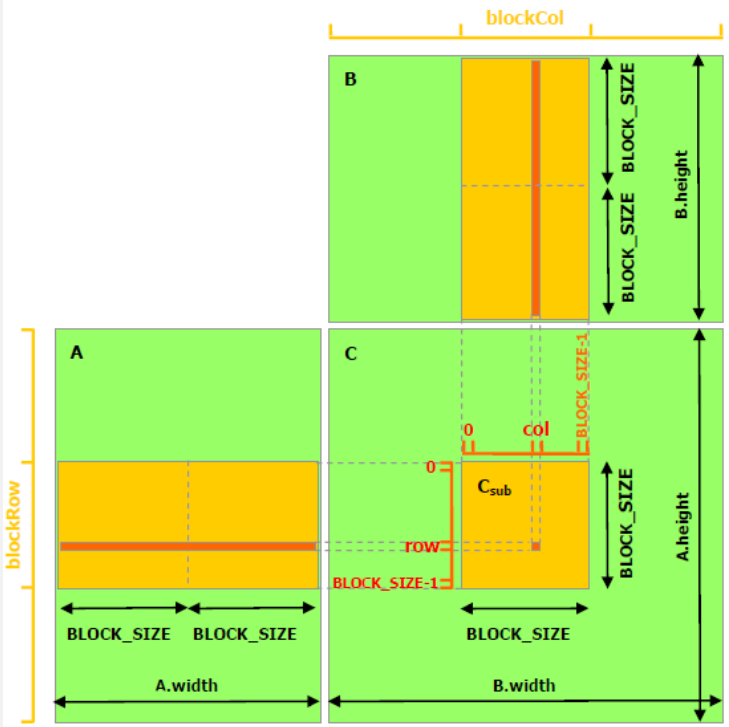
Matrix Multiplication: Revised CUDA Kernel

```
// Matrix multiplication kernel
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;
    // Each thread block computes one sub-matrix of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
    // Each thread computes one elements of Csub
    float Cvalue = 0;
    // Thread row and column with Csub
    int row = threadIdx.y;
    int col = threadIdx.y;
```



Matrix Multiplication: Revised CUDA Kernel (cont'd)

```
    // Loop over sub-matrices of A and B
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // Get sub-matrices of A and B
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
        // Shared memory used to store Asub and Bsub
        __shared__ float As[BLOCK_SIZE] [BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE] [BLOCK_SIZE];
        // Each threads loads one element of each matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        __syncthreads();
        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
        __syncthreads();
    }
    // Write Csub to device memory
    SetElement(Csub, row, col, Cvalue);
}
```



Matrix Multiplication: Revised Host Code

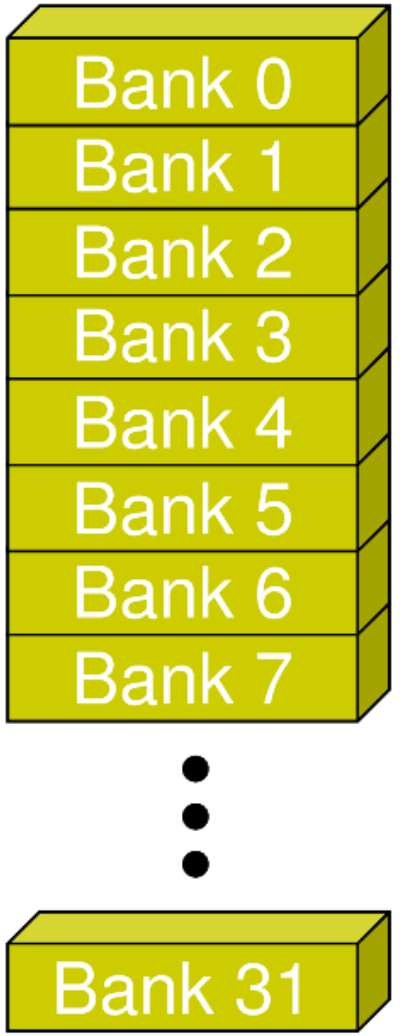
```
#define BLOCK_SIZE 16
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
    // Matrix d_B (skipped)
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A.elements); cudaFree(d_B.elements); cudaFree(d_C.elements);
}
```

Matrix Multiplication: without vs. with Shared Memory

- Without Shared Memory
  - Each thread loads a row of A and a column of B from global memory. In total, A is read *B.width* times and B is read *A.height* times from global memory
  - Each threads performs one multiply and addition for each pair of A and B elements. Compute to off-chip memory access ratio close to 1:1
- With Shared Memory
  - Each thread loads an element per sub-matrix of A and B and from global memory. In total, A is read (*B.width/block_size*) times and B is read (*A.height/block_size*) times from global memory
  - Each threads performs $block\_size^2$ multiply and addition for each pair of A and B elements. Compute to off-chip memory access ratio close to $block\_size^2$: 1

Shared Memory Architecture

- In a parallel machine, many threads access memory at the same time
  - To serve more than one thread, memory is divided into independent **banks**
  - This layout is essential to achieve high bandwidth
- Each SM has Shared Memory organized into 32 Memory Banks
  - Each bank has a bandwidth of 32 bits per two clock cycles
- Shared Memory and L1 cache draw on the same physical memory inside an SM
  - This physical memory (64KB) can be partitioned as
    - 48KB of Shared Memory and 16KB of L1 cache
    - Or 16KB of Shared Memory and 48KB of L1 cache
  - Note: Shared Memory can store less data than the registers (48KB vs. 128 KB)



Shared Memory: Transaction Rules & Bank Conflicts

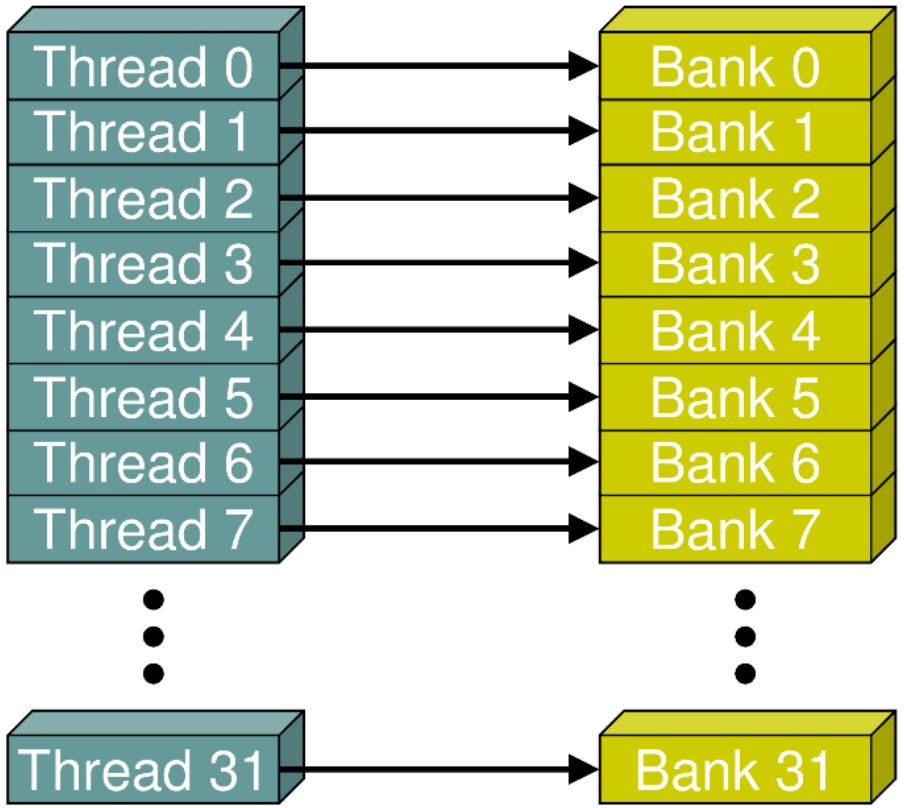
- When reading in four-byte words, 32 threads in a warp attempt to access shared memory simultaneously

- **Bank conflict**: the scenario where two different threads access different words in the same bank

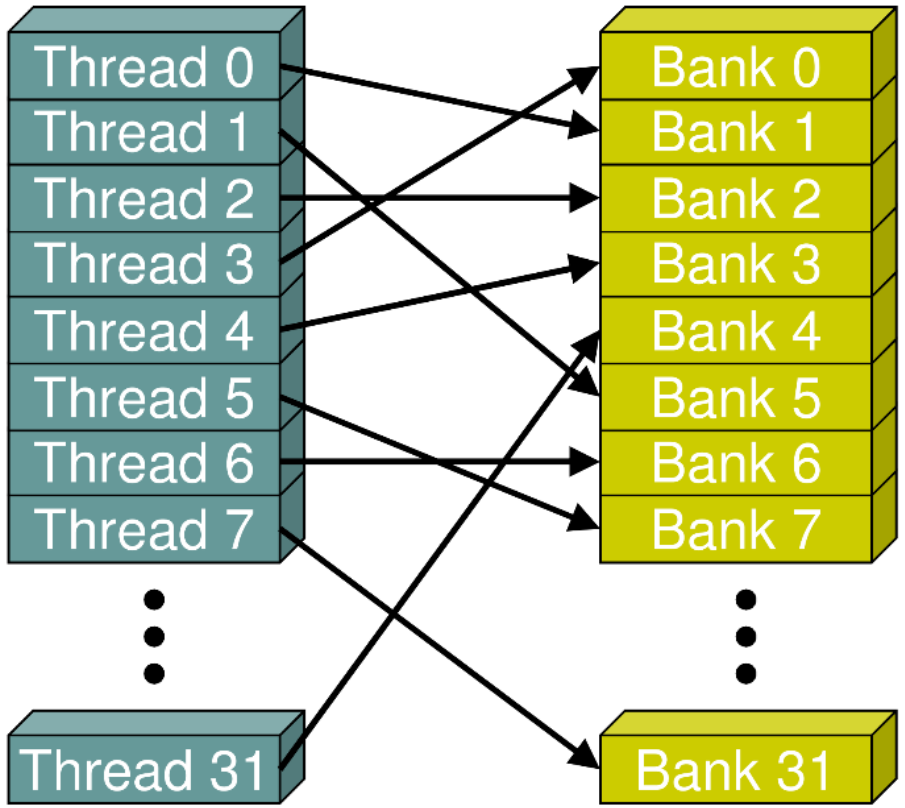
- Bank conflicts force the hardware to serialize your Shared Memory access, which adversely impacts bandwidth!

Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

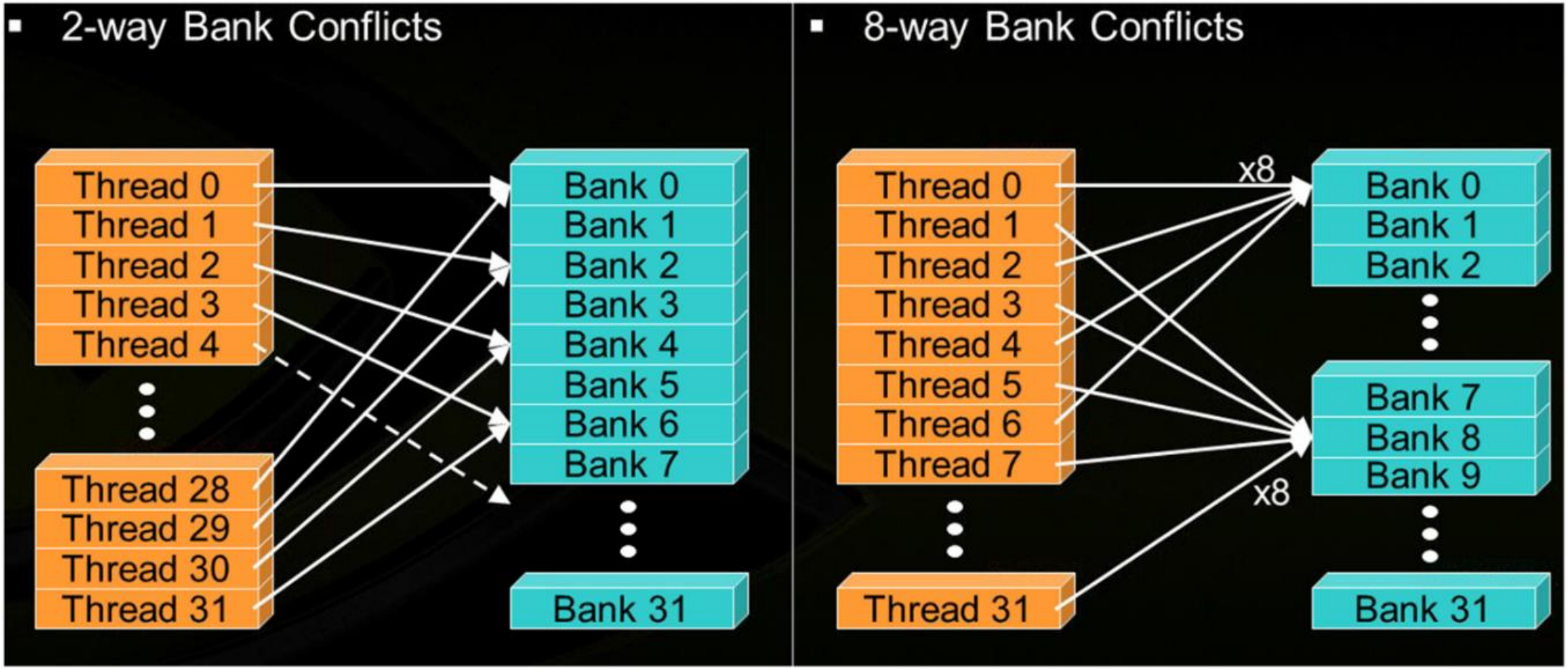


- No Bank Conflicts
  - Random 1:1 Permutation



Transactions involving 4 Byte word

Bank Addressing Examples

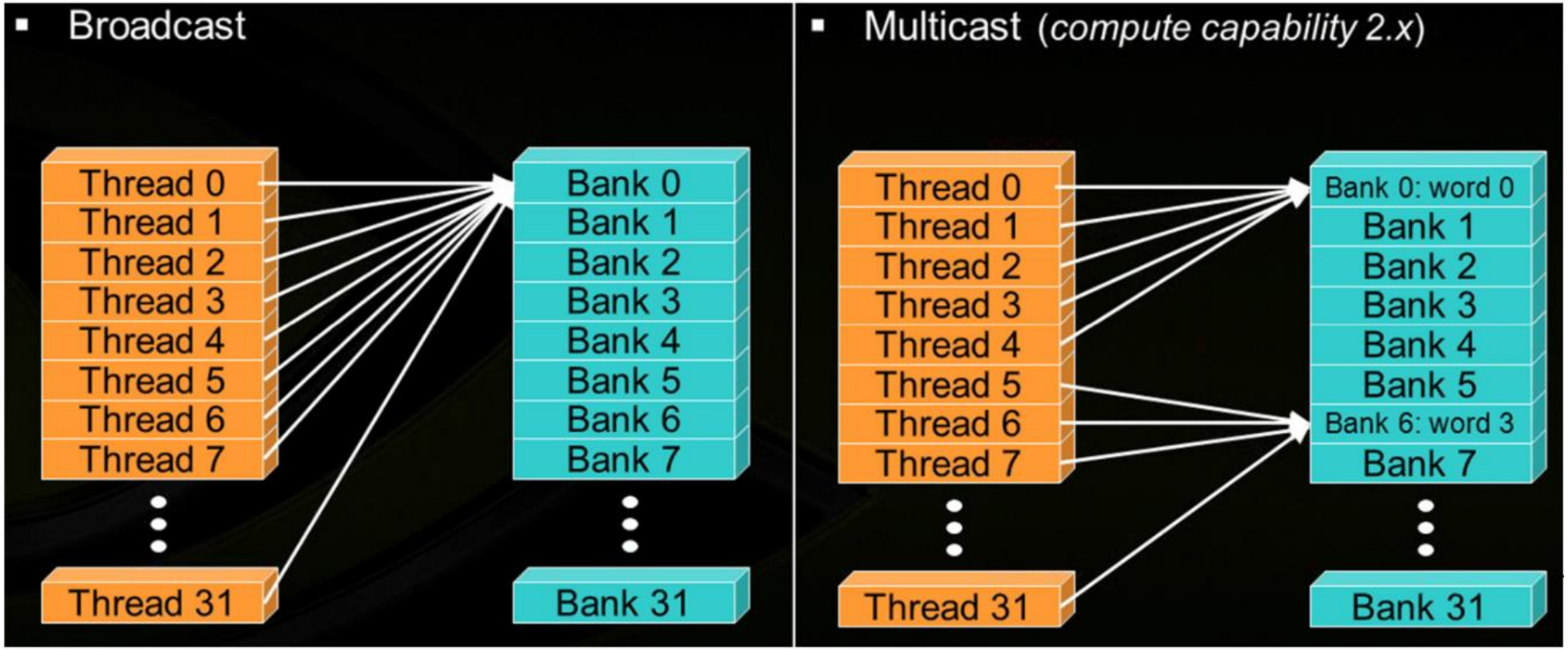


Transactions involving 4 Byte word

Banking Address Example

Two “no conflict” scenarios:

- Broadcast: all threads in a warp access the same word in a bank
- Multicast: several threads in a warp access the same word in the same bank



Why CUDA Streams?

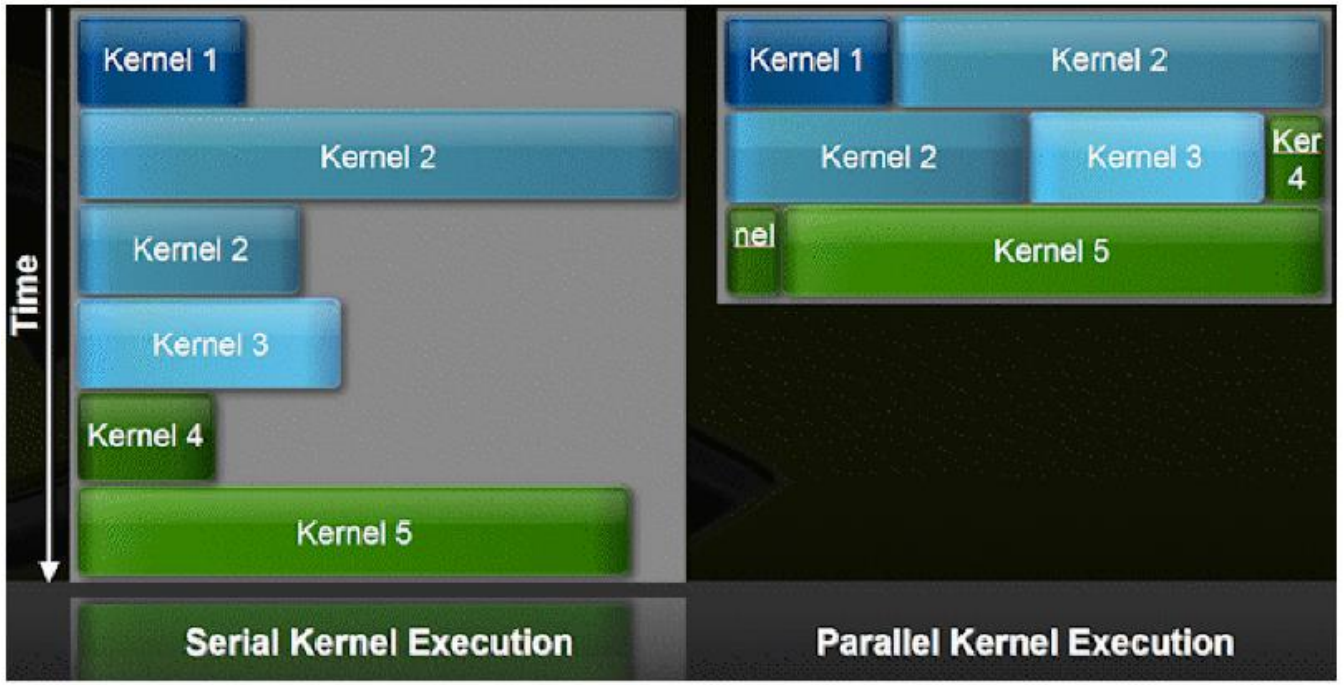
- A CUDA enabled GPU has 2 engines:
  - An execution engine
  - A copy engine, which actually has 2 subengines that can work simultaneously
    - A HostToDevice (H2D) copy subengine
    - A DeviceToHost (D2H) copy subengine
- With **Streams**, CUDA supports using both engines simultaneously
- A basic mechanism enabling **Task Parallelism**
- CUDA Streams allow programmer to
  - Overlap data transfer with device execution
  - Overlap execution on the CPU and execution on the GPU (not strictly a "streams" issue)

CUDA Streams Overview

- A programmer can manage *concurrency* through **streams**. Here concurrency refers to
  - The copy and execution engines of the GPU working at the same time
  - Or Multiple different kernels executed at the same time on the GPU
- A **stream** is a sequence of CUDA commands that execute in issue-order
  - A stream is a queue of GPU operations
  - The execution order in a stream is identical to the order in which the GPU operations are added to the stream (FIFO)
  - An operation in a stream does not commence prior to the previous operation being fully completed
- Note: All GPU calls (memcpy, kernel execution, etc.) are placed into *default* stream (stream 0) unless otherwise specified.

Concurrent Kernel Execution

- Up to 16 kernels can be run on the device at the same time
- When is this useful?
  - Devices of compute capabilities 2.x and above are wide (large number of SMs)
  - A kernel might be launched with an execution configuration that doesn't fully utilize the entire GPU
  - Main idea: multiple independent kernels can be squeezed on GPU at the same time
- GPU looks like a MIMD architecture
  - Require use of **multiple streams**



Recap: *SAXPY*

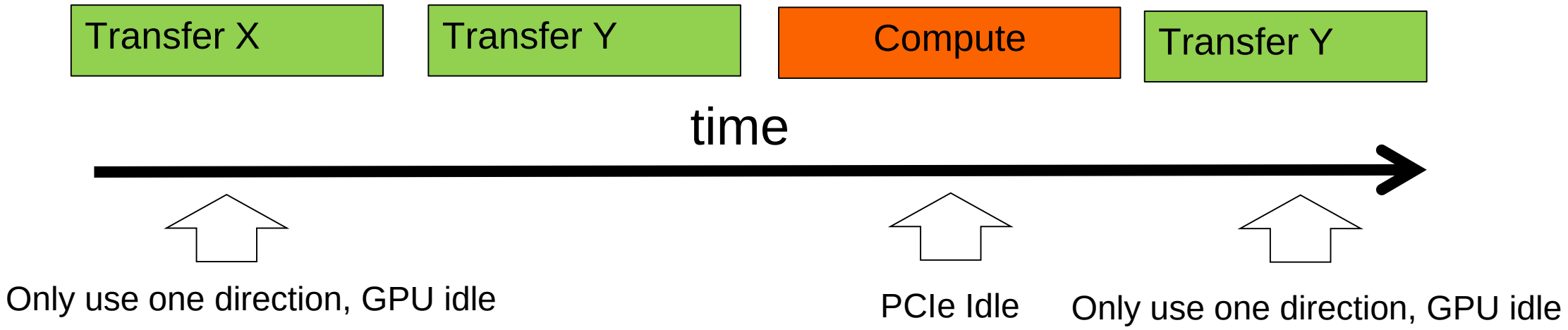
Can be paged out!

Synchronous copies!

```
#define N 20480
__global__ void saxpy(float a, float *x, float *y) {
    ...
}
int main(void) {
    float *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(float);
    x = (float *)malloc(size);
    y = (float *)malloc(size);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory
    cudaMemcpy(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    saxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpy(y, dy, size, cudaMemcpyDeviceToHost);
    // deallocate device memory
    cudaFree(dx); cudaFree(dy);
    free(x); free(y)
}
```

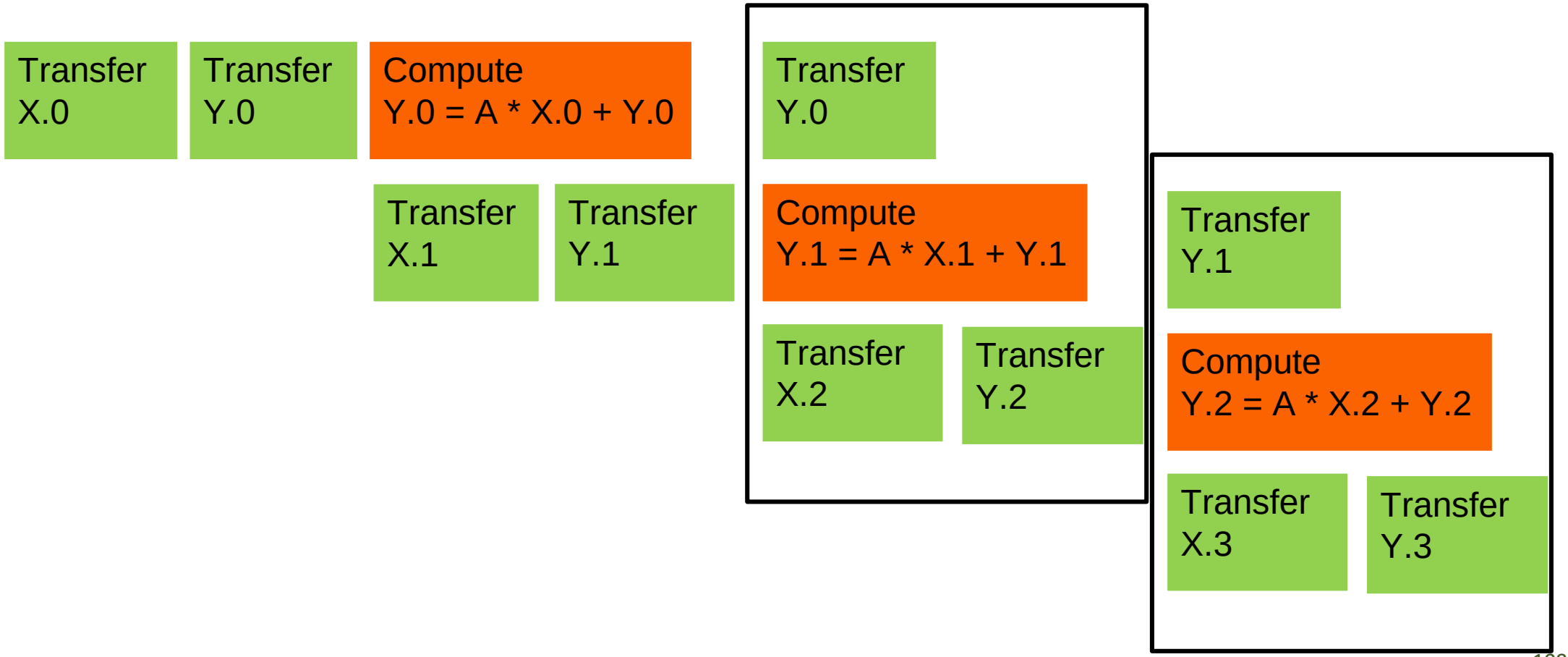
Serialized Data Transfer and Computation

So far, the way we use cudaMemcpy serializes data transfer and GPU computation for the saxpy kernel



Ideal, Pipelined Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments



Allocate & Free Pinned Memory on Host

cudaHostAlloc : Allocates page-locked (pinned) memory on the **host**

```
cudaError_t cudaHostAlloc(void ** pHost,size_t size,
                          unsigned int flags)
```

cudaFreeHost : Frees page-locked (pinned) memory

```
cudaError_t cudaFreeHost(void * ptr)
```

flags:

- cudaHostAllocDefault (we'll use for now)
- cudaHostAllocPortable
- cudaHostAllocMapped
- cudaHostAllocWriteCombined

CUDA Host-Device Data Transfer

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
                       cudaMemcpyKind kind)
```

```
cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count,
                            cudaMemcpyKind kind, cudaStream_t stream = 0)
```

cudaMemcpy *synchronously* copies count bytes from src to dst

cudaMemcpyAsync *asynchronously* copies count bytes from src to dst

- kind is one of
  - **cudaMemcpyHostToHost**
  - **cudaMemcpyHostToDevice**
  - **cudaMemcpyDeviceToHost**
  - **cudaMemcpyDeviceToDevice**

http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

Async *SAXPY*

similar to
Unified Memory
SAXPY

default Stream 0

```
#define N 20480
__global__ void saxpy(float a, float *x, float *y) {...}
int main(void) {
    float *x, *y, a, *dx, *dy;
    size_t size = N*sizeof(float);
    cudaHostAlloc((void **) &x, size, cudaHostAllocDefault);
    cudaHostAlloc((void **) &y, size, cudaHostAllocDefault);
    // initialize x and y on host (skipped)
    // allocate device memory for x and y
    cudaMalloc((void **) &dx, size);
    cudaMalloc((void **) &dy, size);
    // copy host memory to device memory asynchronously
    cudaMemcpyAsync(dx, x, size, cudaMemcpyHostToDevice);
    cudaMemcpyAsync(dy, y, size, cudaMemcpyHostToDevice);
    // launch the kernel function
    saxpy<<<N/256,256>>>(a, dx, dy);
    // copy device memory to host memory
    cudaMemcpyAsync(y, dy, size, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    // deallocate memory
    cudaFree(dx); cudaFree(dy);
    cudaFreeHost(x);
    cudaFreeHost(y)
}
```

CUDA Streams: Synchronization Aspects

- **cudaDeviceSynchronize()** halts execution *on the host* until all preceding commands in all CUDA streams have completed
- **cudaStreamSynchronize()** takes a stream as a parameter and halts execution *on the host* until all preceding commands in the given CUDA stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device
- **cudaStreamWaitEvent()** takes a CUDA *stream* and an *event* as parameters and makes all the commands added to the given stream after the call to **cudaStreamWaitEvent()** delay their execution until the given event has completed. Note: this halts the execution of tasks *in a stream*!
- **cudaStreamQuery()** provides applications with a way to know if all preceding commands *in a stream* have completed

NOTE: To avoid unnecessary slowdowns, use these synchronization functions sparingly

http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html

cudaStreamWaitEvent Example

- Assume stream1 and stream2 have already been defined/initialized
- The point of this example:
  - Use the two copy subengines at the same time
  - Wait on the stream2 launching of the myKernel until the copy operation in stream1 is completed

```
cudaEvent_t event;
cudaEventCreate (&event);                                   // create event
cudaMemcpyAsync ( d_in, in, size, H2D, stream1 );           // 1) H2D copy of new input
cudaEventRecord (event, stream1);                           // record event
cudaMemcpyAsync ( out, d_out, size, D2H, stream2 );         // 2) D2H copy of previous result
cudaStreamWaitEvent ( stream2, event );                     // wait for event in stream1
myKernel<<< 1000, 512, 0, stream2 >>> ( d_in, d_out );      // 3) GPU must wait for 1 and 2
someCPUfunction ( blah, blahblah )                          // this gets executed right away
```

Create & Destroy CUDA Streams

cudaStreamCreate : Creates a new asynchronous stream

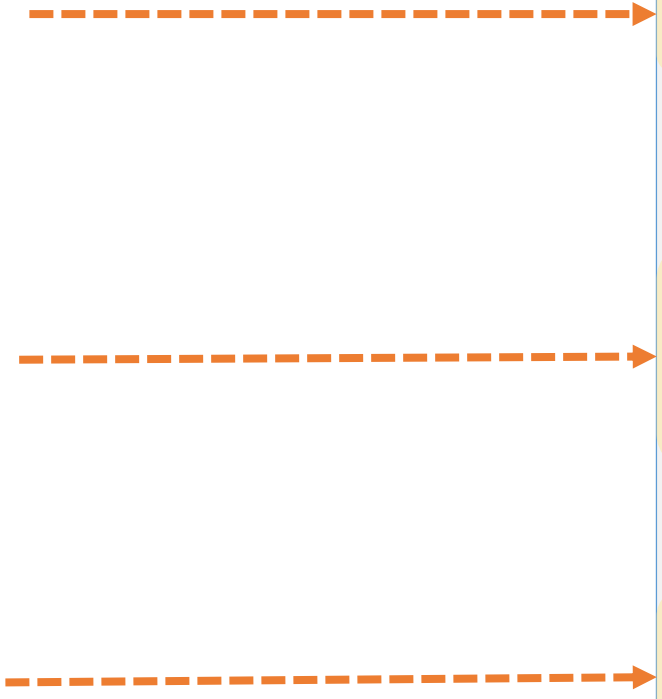
```
cudaError_t cudaStreamCreate(cudaStream_t* pStream)
```

cudaStreamDestroy : Destroys and cleans up an asynchronous stream

```
cudaError_t cudaStreamDestroy(cudaStream_t stream)
```

In case the device is still doing work in the stream when cudaStreamDestroy() is called, the function will return immediately and the resources associated with stream will be released automatically once the device has completed all work in stream.

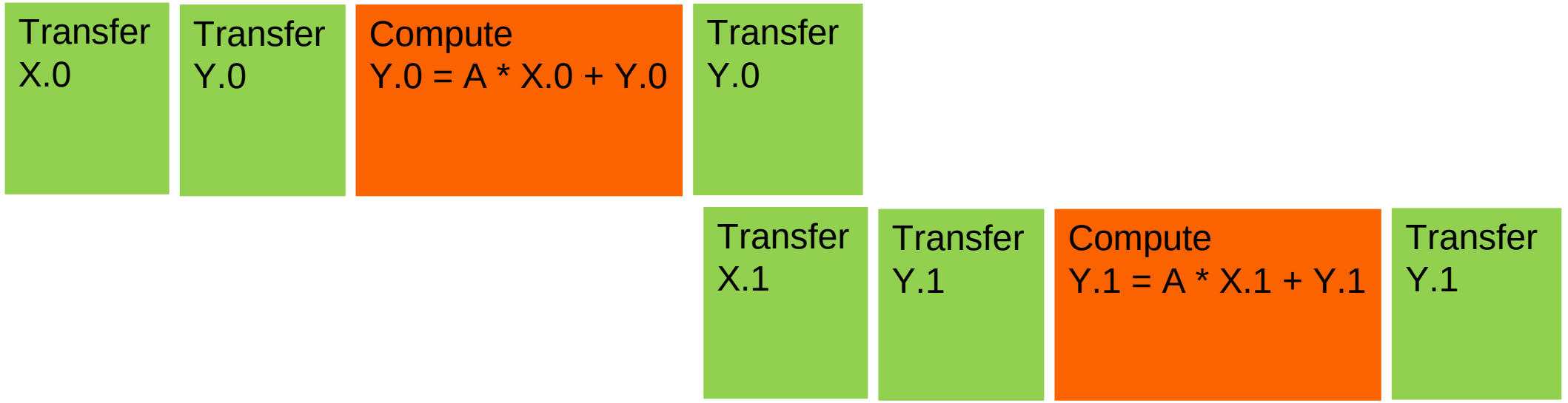
2-Stream *SAXPY* Host Code



```
#define N 20480

int main(void) {
    float *x, *y, a, *dx[2], *dy[2];
    int i, j, segLength = 1024;
    size_t size = N*sizeof(float);
    size_t segSize = segLength*sizeof(float);

    // allocate pinned memory on host
    cudaHostAlloc((void **) &x, size, cudaHostAllocDefault);
    cudaHostAlloc((void **) &y, size, cudaHostAllocDefault);
    // initialize x and y on host (skipped)

    // allocate device memory for each stream
    for (i = 0; i < 2; ++i ) {
        cudaMalloc((void **) &dx[i], segSize);
        cudaMalloc((void **) &dy[i], segSize);
    }

    // create 2 streams
    cudaStream_t stream[2];
    for (i = 0; i < 2; ++i)
        cudaStreamCreate(&stream[i]);
```

```
    for (i = 0; i < N; i += 2) {
        for (j = 0; j < 2; j++) {
            // copy host memory to device memory
            cudaMemcpyAsync(dx[j], x + (i+j)*segLength, segSize,
                            cudaMemcpyHostToDevice, stream[j]);
            cudaMemcpyAsync(dy[j], y + (i+j)*segLength, segSize,
                            cudaMemcpyHostToDevice, stream[j]);
            // launch the kernel function
            saxpy<<<segLength/256,256,0,stream[j]>>>(a, dx[j], dy[j]);
            // copy device memory to host memory
            cudaMemcpyAsync(y + (i+j)*segLength, dy[j], segSize,
                            cudaMemcpyDeviceToHost, stream[j]);
        }
    }

    cudaDeviceSynchronize();
    // destroy and clean up the 2 streams
    for (int i = 0; i < 2; ++i) cudaStreamDestroy(stream[i]);
    // deallocate memory
    cudaFree(dx); cudaFree(dy);
    cudaFreeHost(x); cudaFreeHost(y)
    return 0;
}
```

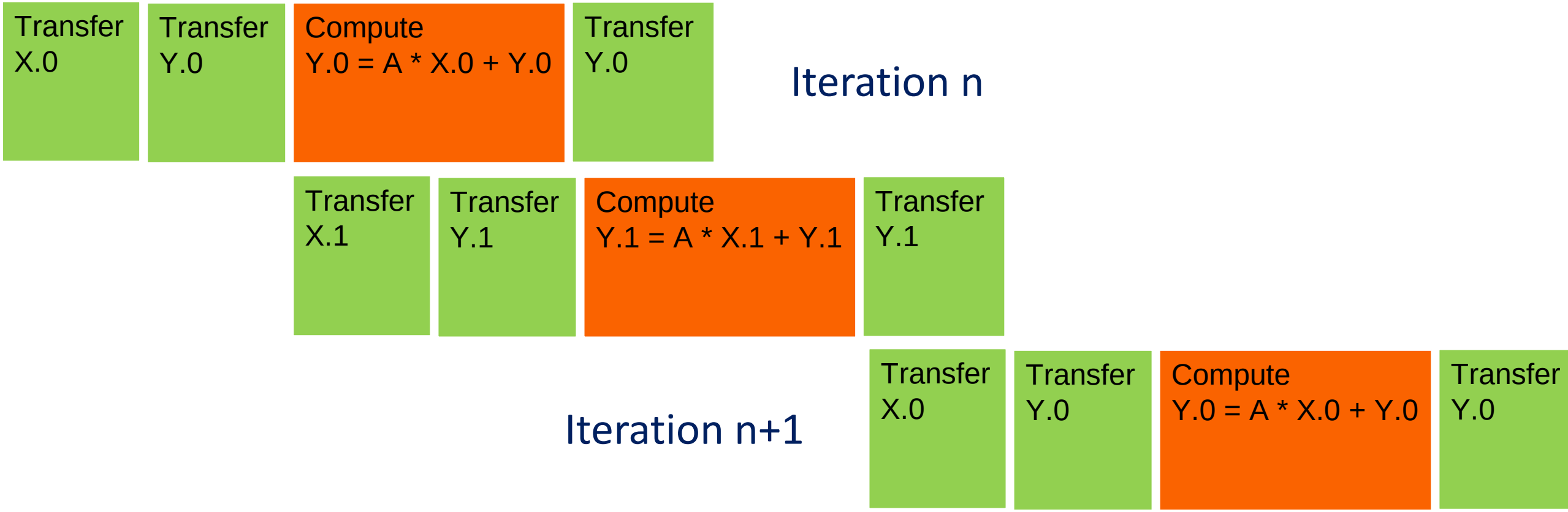
Not quite the overlap we want



Better 2-Stream SAXPY Host Code

```
for (i = 0; i < N; i += 2) {
    cudaMemcpyAsync(dx[0], x + i*segLength, segSize,
                    cudaMemcpyHostToDevice, stream[0]);
    cudaMemcpyAsync(dy[0], y + i*segLength, segSize,
                    cudaMemcpyHostToDevice, stream[0]);
    cudaMemcpyAsync(dx[1], x + (i+1)*segLength, segSize,
                    cudaMemcpyHostToDevice, stream[1]);
    cudaMemcpyAsync(dy[1], y + (i+1)*segLength, segSize,
                    cudaMemcpyHostToDevice, stream[1]);

    saxpy<<<segLength/256,256,0,stream[0]>>>(a, dx[0], dy[0]);
    saxpy<<<segLength/256,256,0,stream[1]>>>(a, dx[1], dy[1]);

    cudaMemcpyAsync(y + i*segLength, dy[0], segSize,
                    cudaMemcpyDeviceToHost, stream[0]);
    cudaMemcpyAsync(y + (i+1)*segLength, dy[1], segSize,
                    cudaMemcpyDeviceToHost, stream[1]);
}
```

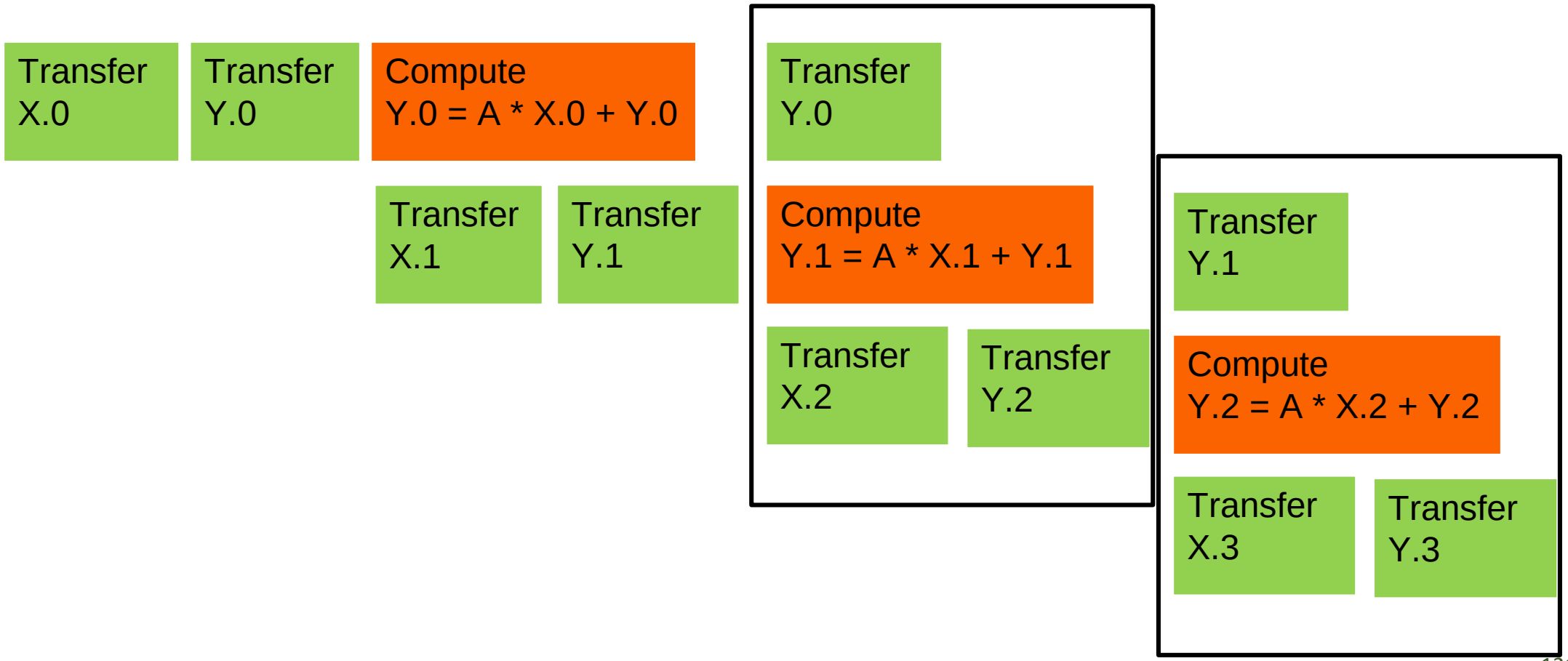
Better, but not quite the overlap we want



Ideal, Pipelined Timing

Will need at least 3 buffers for each original X and Y

Code is more complicated (left as an exercise for the more adventurous)



Imperatives for Efficient CUDA Code

- Expose abundant fine-grained parallelism
  - need thousands of threads for full utilization
- Maximize on-chip work
  - on-chip memory is orders of magnitude faster
- Minimize execution divergence
  - SIMT execution of threads in 32-thread warps
- Minimize memory divergence
  - warp loads and consumes a complete 128-byte cache line

Optimizing GPU Performance

- Understand the GPU architecture
- Understand how applications maps to architecture
- Use lots of threads and blocks
- Often better to redundantly compute in parallel
- Access memory in local regions
- Leverage high memory bandwidth
- Enable global memory coalescing
- Optimize memory copies
- Keep data in GPU device memory
- Experiment and measure

nvprof

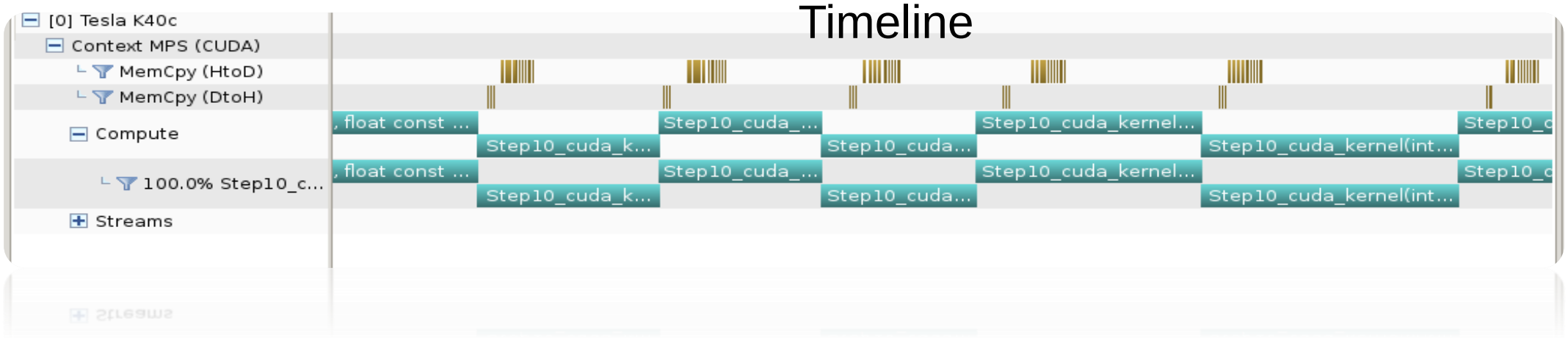
- **nvprof** is the command line profiling tool provided with the CUDA Toolkit

  http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview
- **nvprof** enables the collection of a timeline of CUDA-related activities on both CPU and GPU, including:
  - kernel execution
  - memory transfers
  - memory set
  - CUDA API calls and events or metrics for CUDA kernels
- No need to recompile!
- Profiling results are displayed in the console after the profiling data is collected, and may also be saved for later viewing by either nvprof or the Visual Profiler.

Nvidia Visual Profiler

- Nvidia Visual Profiler (nvvp) displays a timeline of the application's activity on both CPU and GPU

  http://docs.nvidia.com/cuda/profiler-users-guide/#visual



- Nvidia Visual Profiler does not require any application changes; however, by making some simple modifications and additions, you can greatly increase its usability and effectiveness.

What about OpenCL?

- OpenCL is a standardized, cross-platform API designed to support **portable** parallel application development on heterogeneous computing systems, including multicore CPUs, GPUs, DSPs, FPGAs and other processors
- OpenCL has a more complex platform and device management model than CUDA
- OpenCL's data parallel execution model mirrors CUDA, but with different terminology

OpenCL parallelism concepts	**CUDA equivalent**
Kernel	Kernel
Host program	Host program
NDRange (index space)	Grid
Work group	Block
Work item	Thread

- OpenCL 1.0 was released in August 2009, 1.1 in 2010, 1.2 in 2011, 2.0 in 2013, 2.1 in 2015 and 2.2 in 2017. But Nvidia driver only supports OpenCL 1.1.

Further Readings

- **CUDA Toolkit Documentation:** http://docs.nvidia.com/cuda/
- **CUDA C Programming Guide:** http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- **CUDA C Best Practice Guide:** http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf
- **Programming Massively Parallel Processors: A Hands-on Approach**, 3rd Edition, *by* David B. Kirk & Wen-mei W. Hwu, Morgan Kaufmann 2016 https://www.amazon.com/Programming-Massively-Parallel-Processors-Hands/dp/0128119861
- **Parallel Programming with OpenACC**, *by* Rob Farber, Elsevier 2017 https://www.sciencedirect.com/science/book/9780124103979
- **PGI CUDA Fortran Programming Guide and Reference:** https://www.pgroup.com/doc/pgicudafortug.pdf
- **CUDA Fortran for Scientists and Engineers**, *by* Fatica & Ruetsch, Elsevier 2014 http://www.sciencedirect.com/science/book/9780124169708

Further Readings (cont'd)

- The Evolution of GPUs for General Purpose Computing, *by* Ian Buck
- Optimizing Parallel Reduction in CUDA, *by* Mark Harris
- Multi-GPU Programming with MPI, by Jiri Kraus
- Nvidia Tesla P100 Architecture Whitepaper
- Nvidia Fermi Architecture Whitepaper
- GPU Gems
- GPU Gems 2
- GPU Gems 3
- Benchmarking GPUs to Tune Dense Linear Algebra, *by* Vasily Volkov & James W. Demmel