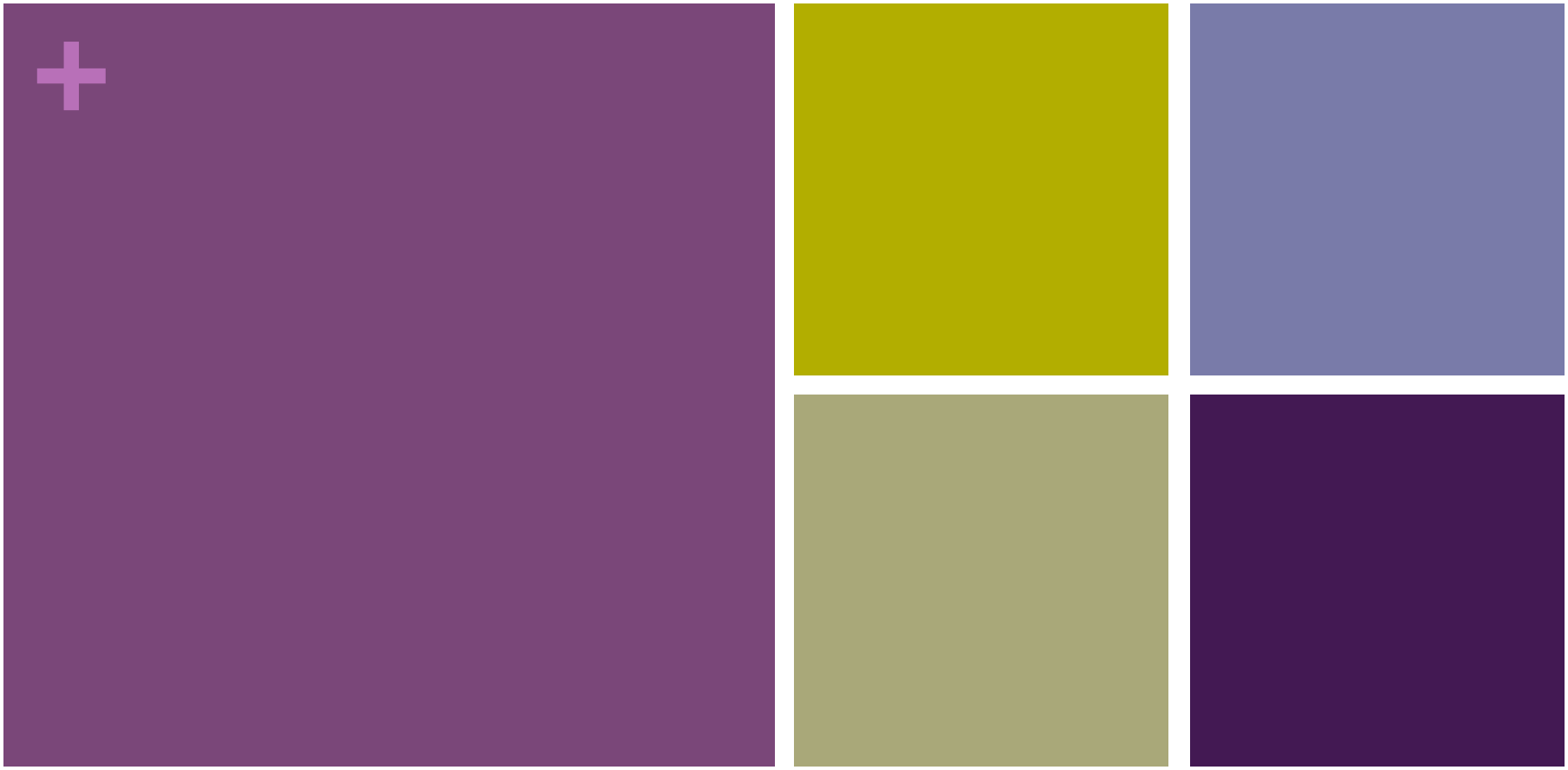


Data Parallel Execution and CUDA Memories

Abhijit Bendale

03/11/2014


TIME FOR REVIEW

CUDA Software Development

Overview

CUDA Optimized Libraries:
math.h, FFT, BLAS, …

Integrated CPU + GPU
C Source Code

NVIDIA C Compiler

NVIDIA Assembly
for Computing (PTX)

CPU Host Code

CUDA
Driver

Profiler

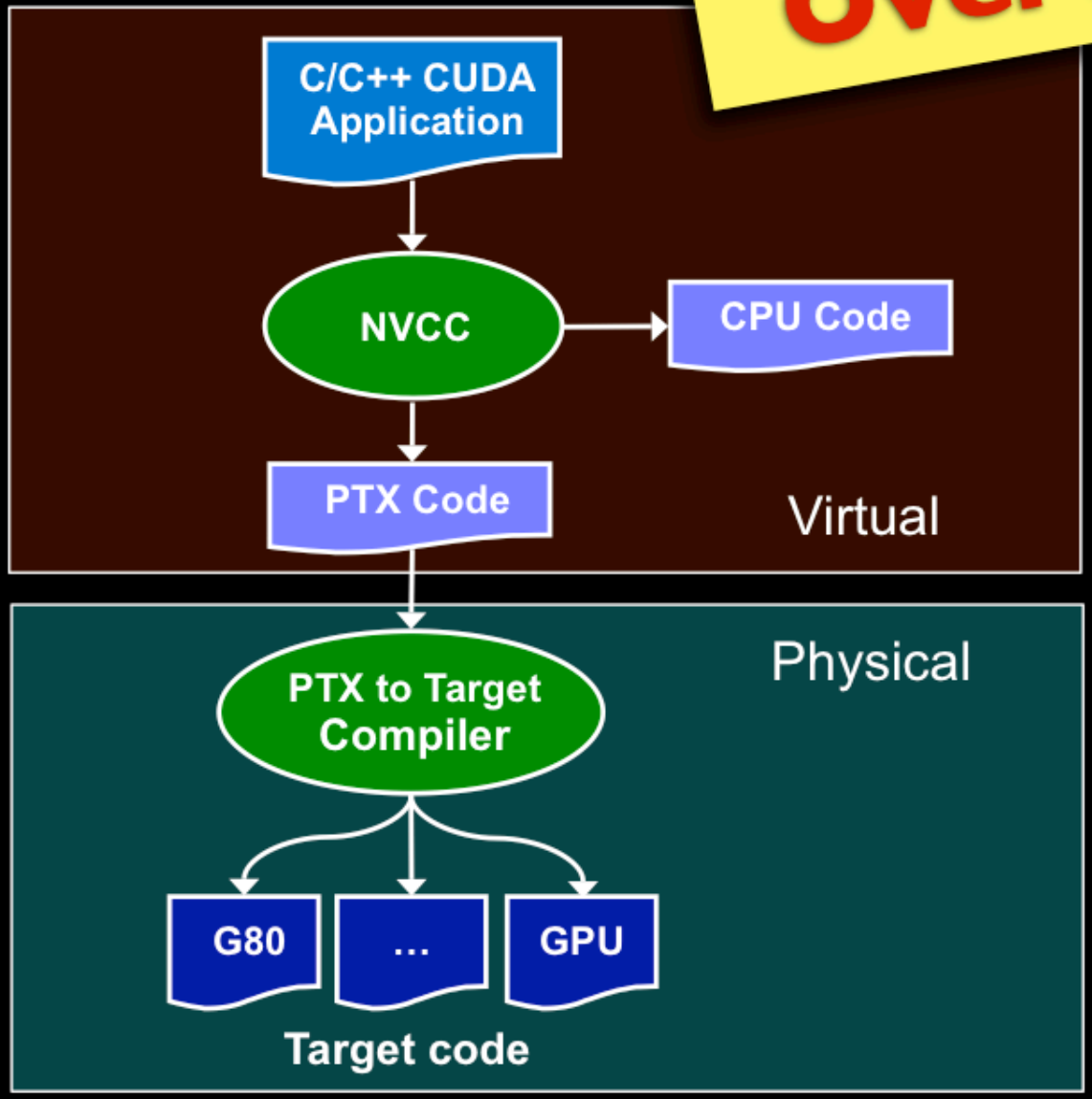
Standard C Compiler

GPU

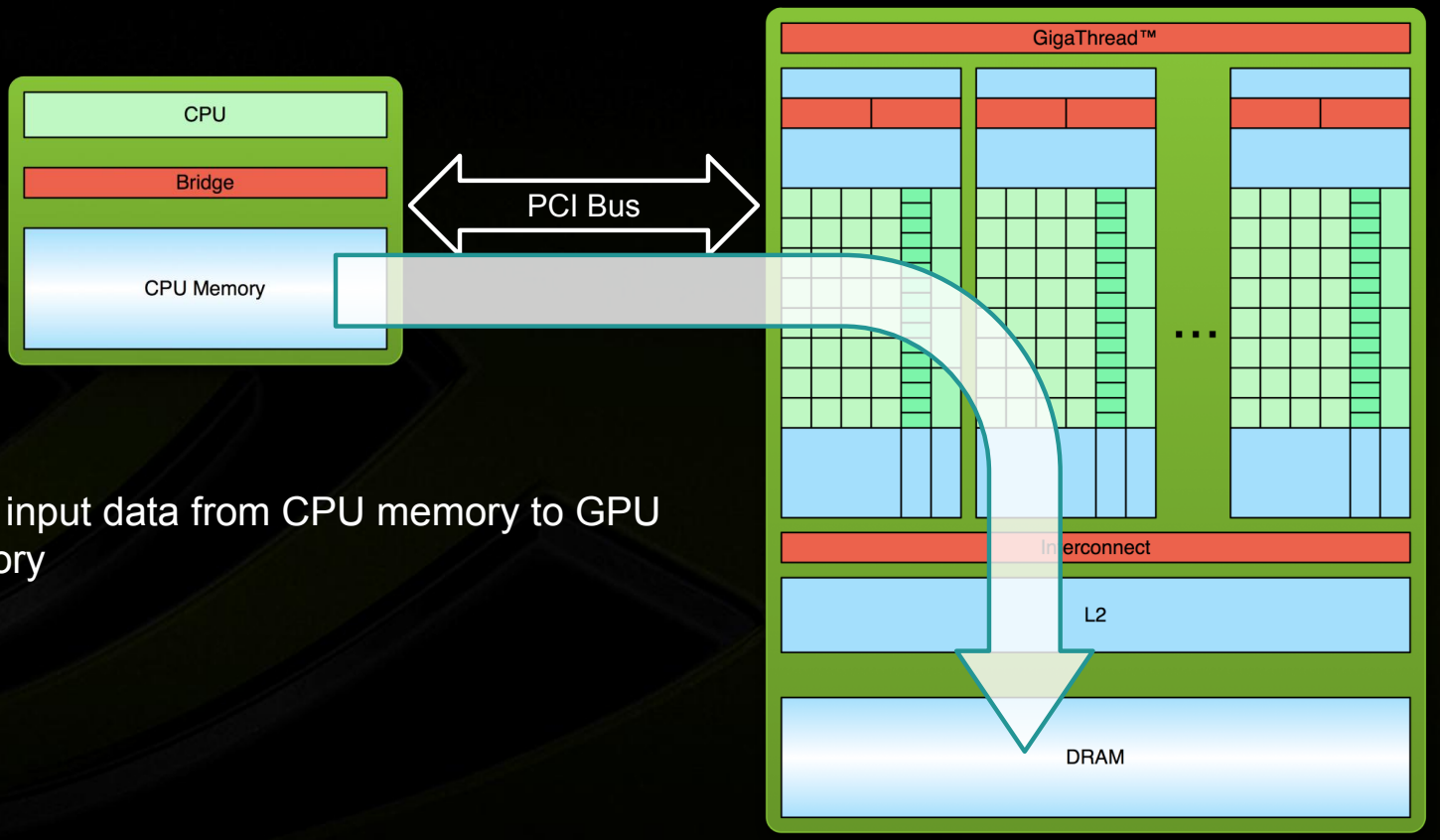
CPU

Compiling CUDA Code

Overview

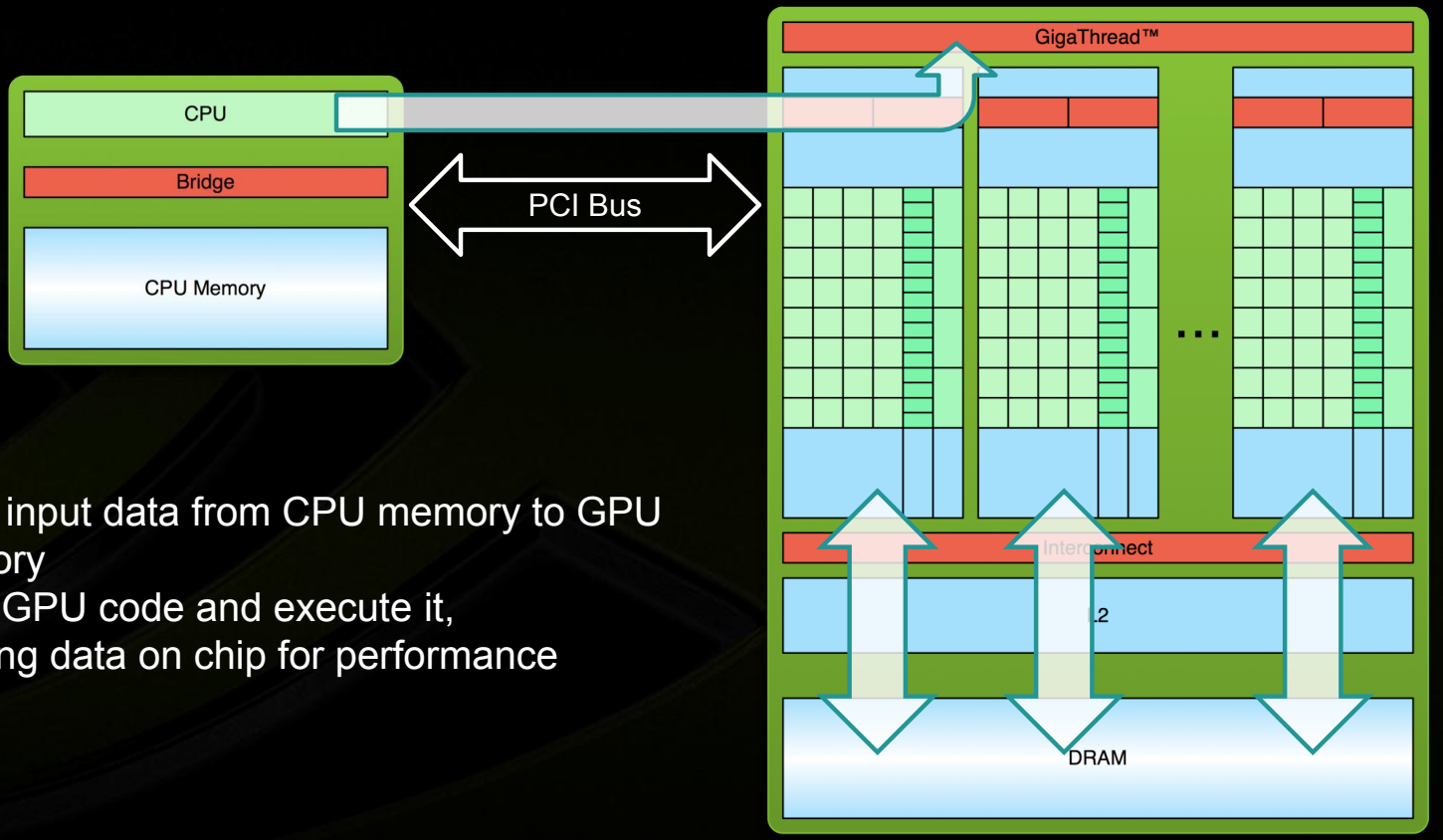


Simple Processing Flow



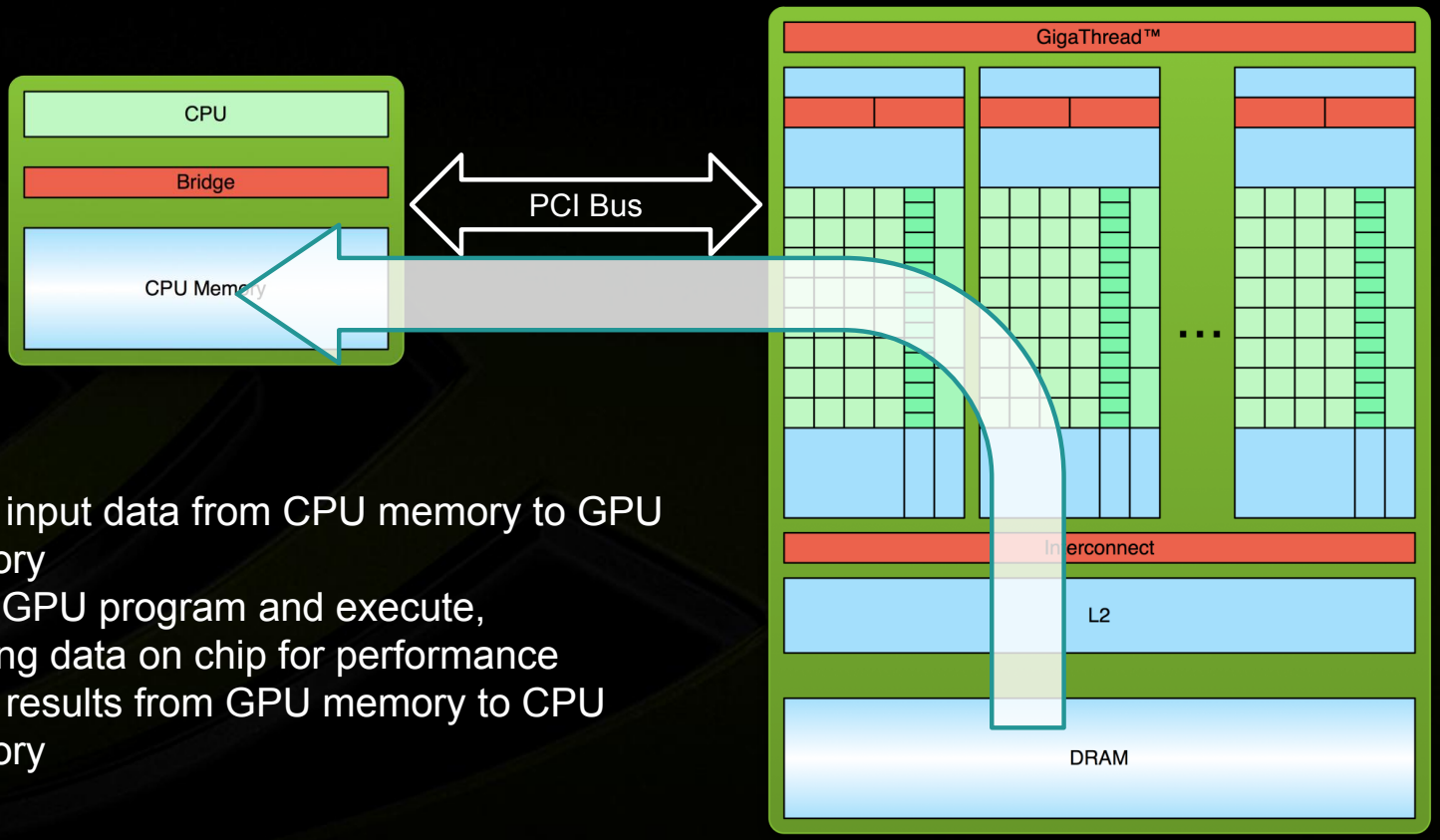
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it, caching data on chip for performance

Simple Processing Flow

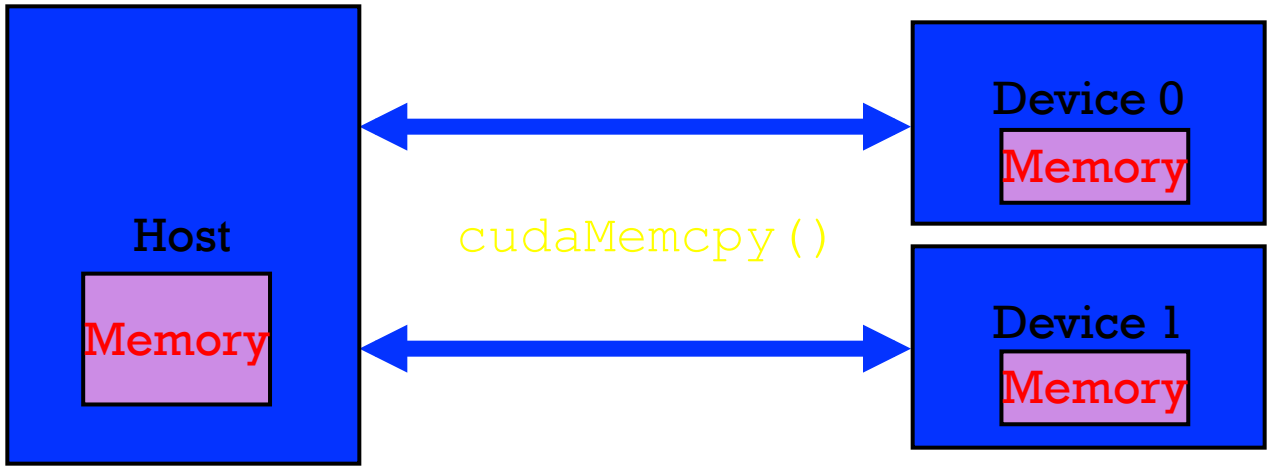


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Memory Model

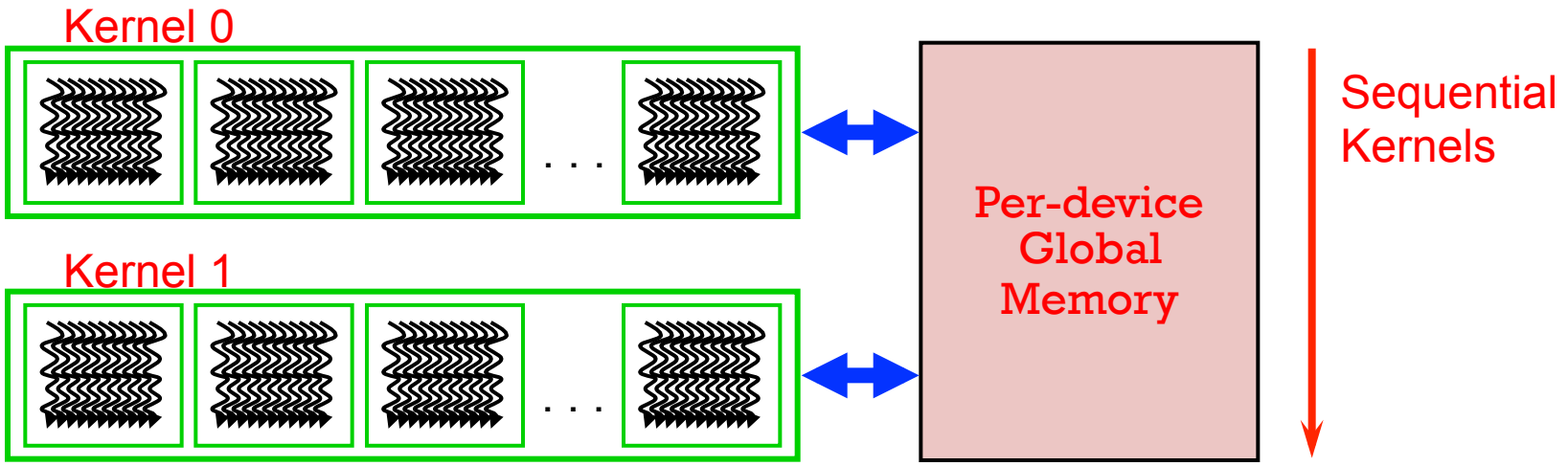
```
float *h_A, *h_B, *h_C
malloc(); // assign memory
free();   // free memory
```

```
float *d_A, *d_B, *d_C
cudaMalloc()
cudaFree()
```



We need to assign memory in the device (GPU) for the Variables that we wish to use in the device

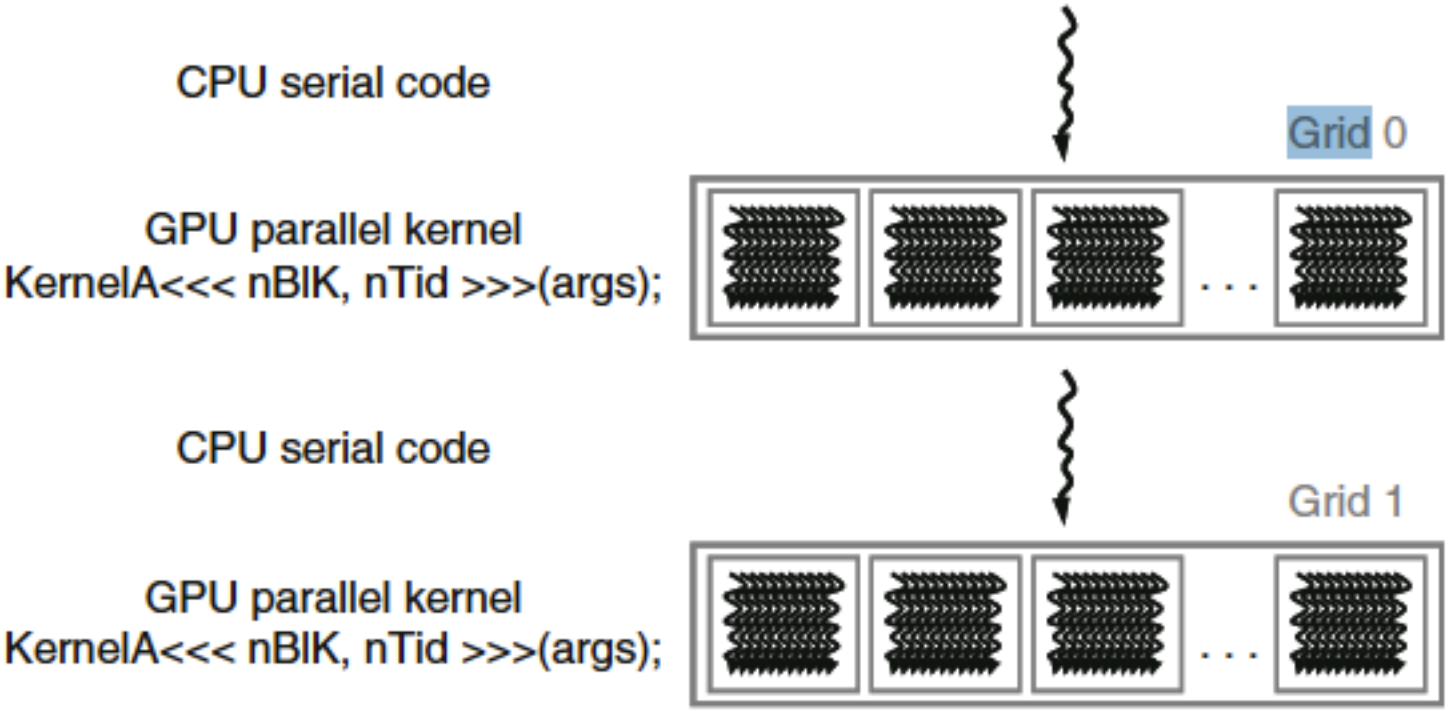
Memory Model



All the blocks within the device have access to global memory of the device

Grid, Block, Thread, Kernel..

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```



All threads that are generated by a kernel during an invocation are collectively called a grid

FIGURE 3.2

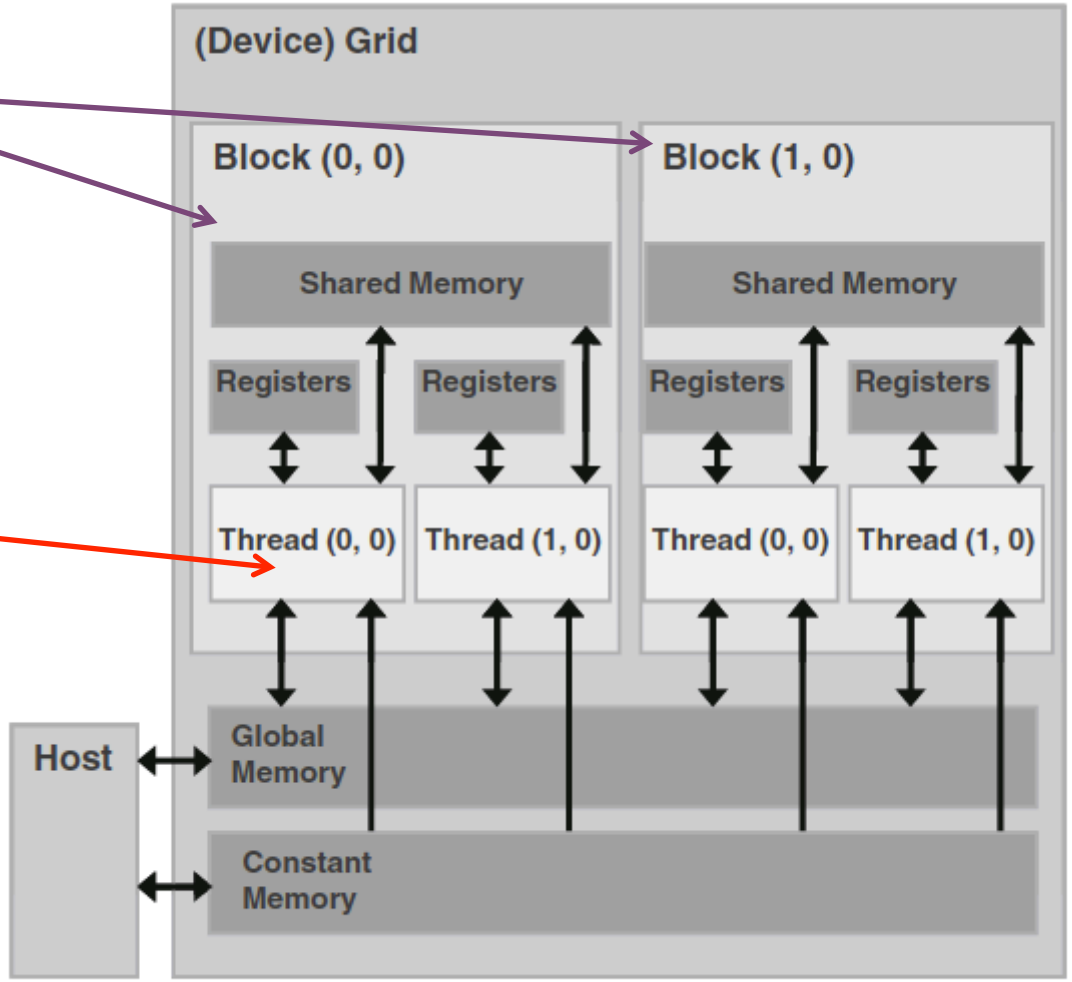
Execution of a CUDA program.

Grid, Block, Thread, Kernel..

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```

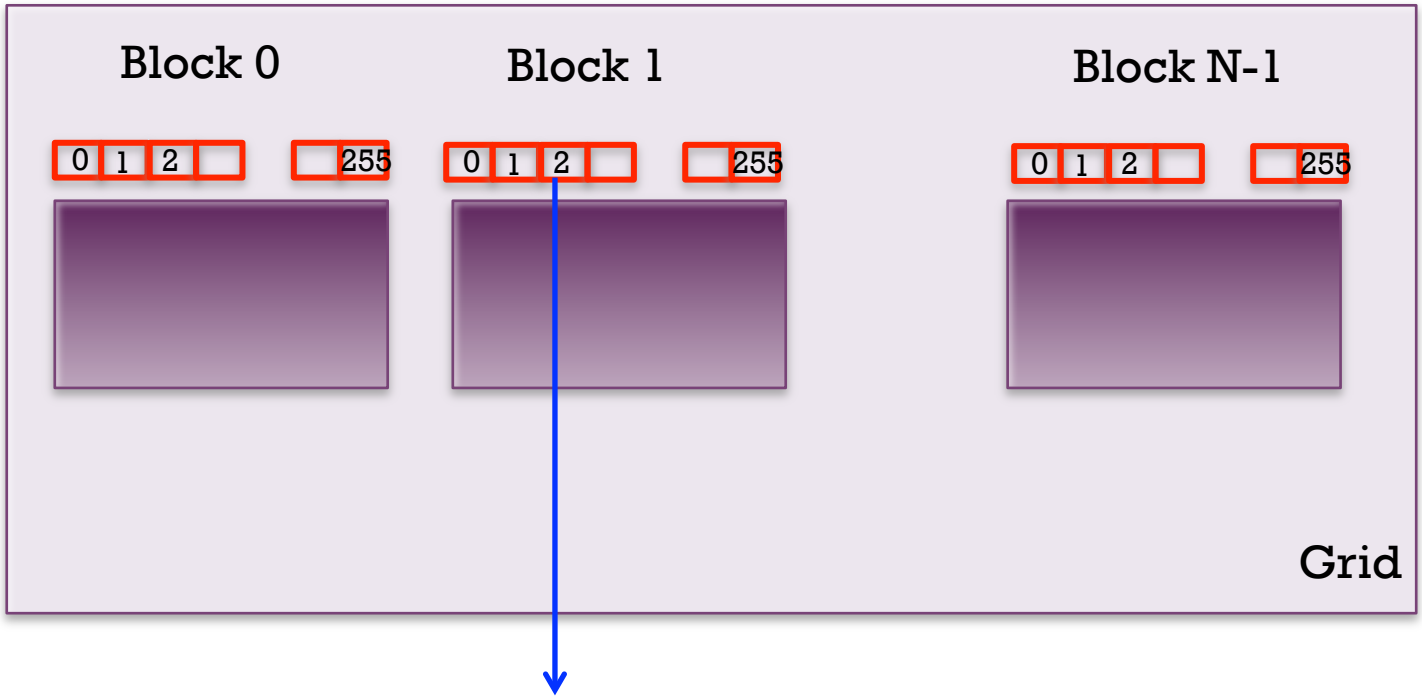
A grid consists
Of multiple blocks.
Each block has finite
Size (usually in
Increments of 32,
since 32 threads form
a **warp**).

Each block can execute
many threads



Grid, Block, Thread, Kernel..

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```



Block = 1, Block Dimension = 256, Thread id = 2

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
 258  =      2       + 256 * 1
```

A more complete version of vecAdd()

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
     if (i < n)
         C[i] = A[i] + B[i];
}
```

Device (GPU) co

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    cudaMalloc((void**) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**) &d_C, size);

    vecAddKernel <<< ceil(n/256.0), 256 >>> (d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    //Free device (GPU) memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

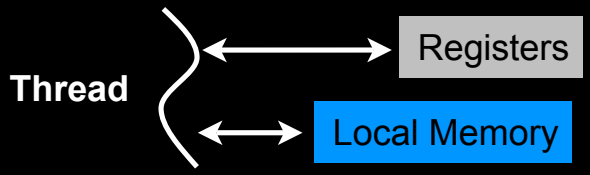
Launching the kernel code

Host (CPU) code

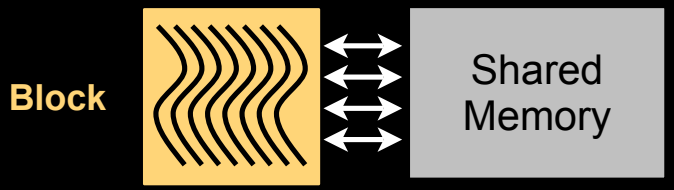
Kernel Memory Access

Basics

- **Per-thread**

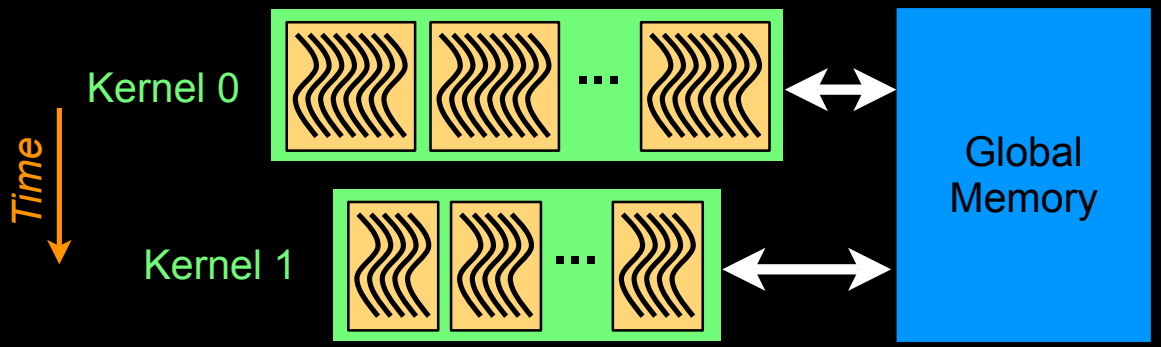


- **Per-block**



- On-chip, small
- Fast

- **Per-device**



- Off-chip, large
- Uncached
- Persistent across kernel launches
- Kernel I/O

Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C functions

- Pointers are just addresses
  - Can’ t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa

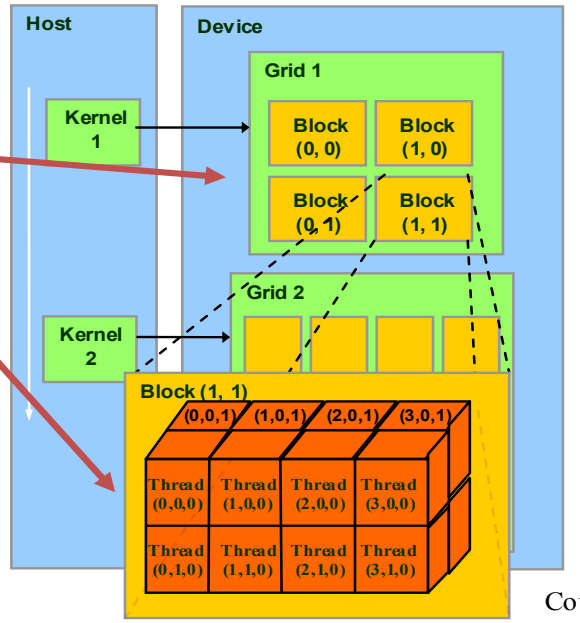
Data Parallel Execution Model

- Fine-grained, data-parallel threads are fundamental means of parallel execution in CUDA
- Each thread uses a unique co-ordinate given by threadId {x,y,z}
- We will now study
  - Organization of threads
  - Resource Assignment to threads
  - Synchronization of threads
  - Scheduling of threads in a grid

Threads and dimensions

Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



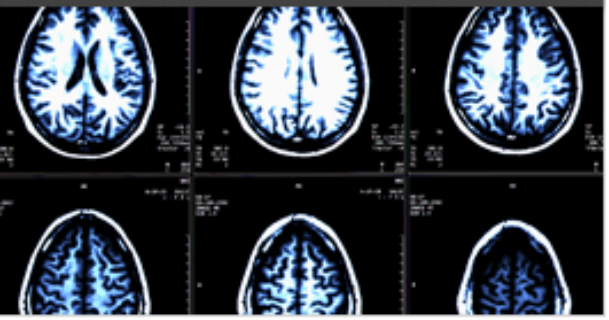
Courtesy: NDVIA

Organization

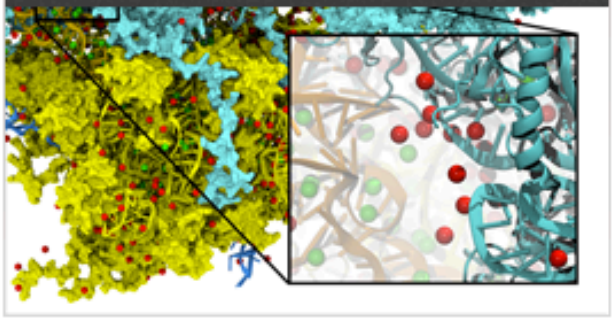
- Grid is a 3D array of blocks. Each block is a 3D array of threads.
- The exact organization of a grid is determined by configuration parameters <<< >>> of kernel launch statement
- <<< No of Blocks in grid, Number of threads per block >>>
- E.g. if we have to execture 4096 threads
  - dim3 dimBlock(128, 1, 1) → Create 1D grid of 128 blocks
  - dim3 dimGrid(32, 1, 1) → each block has 32 threads
  - vecAddKernel <<< dimBlock, dimGrid >>> (….);
  - 128*32 = 4096

Multi-Dimensional Data

MEDICAL IMAGING



BIOINFORMATICS

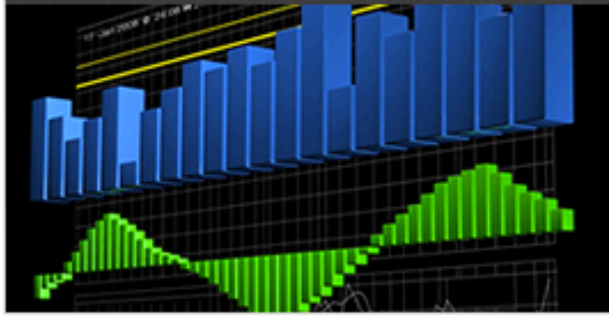


SUPERCOMPUTING CENTERS

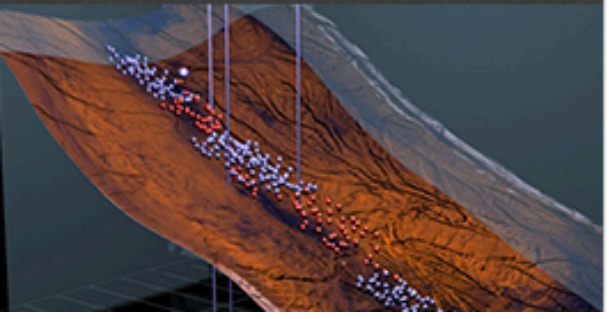
CAD / CAM / CAE

COMPUTATIONAL FLUID DYNAMICS

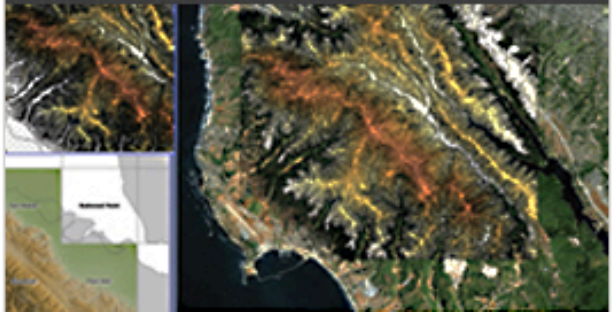
COMPUTATIONAL FINANCE



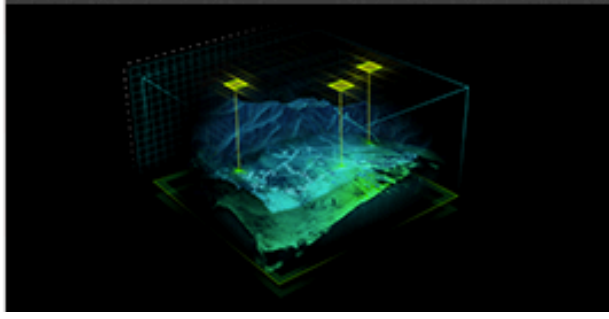
SEISMIC EXPLORATION



GIS



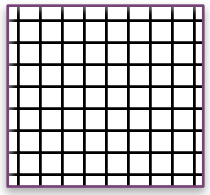
DEFENSE



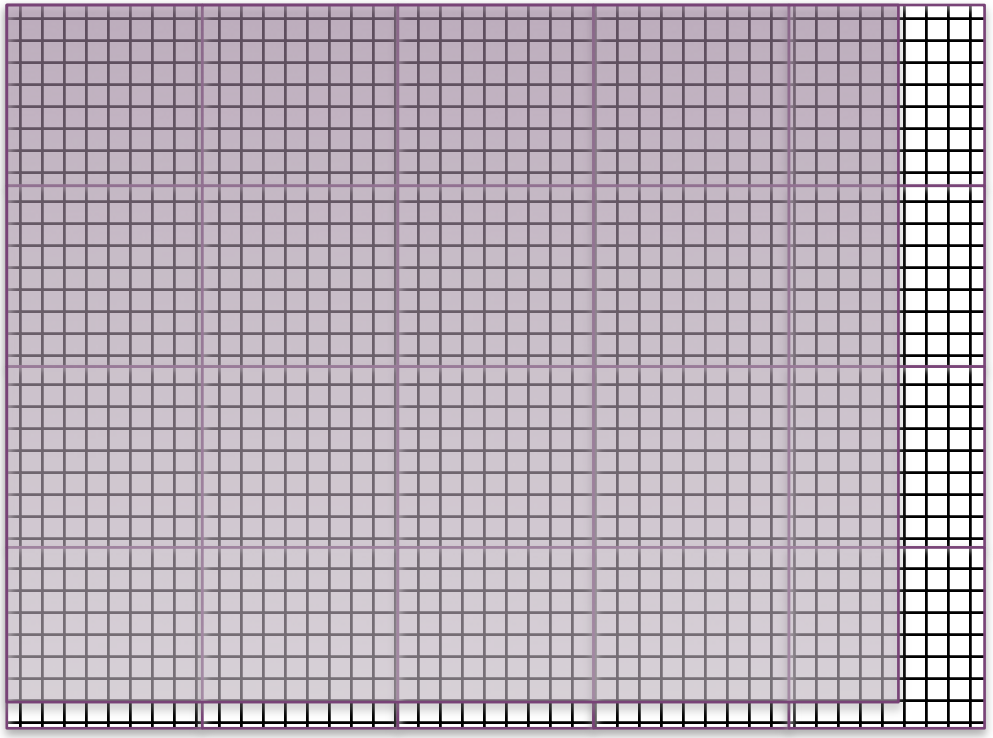
Processing Multi-Dimensional Data

- The choice of 1D, 2D, 3D thread organization is done based on the nature of data
- 1D data : vector manipulation
- 2D Data: Image processing
- 3D Data: MRI Scans

A 2D Example



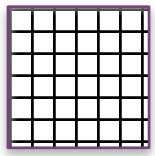
16 x 16 blocks



Process an image of size 76 x 62 pixels.
Block size: 16x16, Total blocks needed = 5x4 = 20

Goal : Double the value of each pixel

A 2D Example



16 x 16
blocks

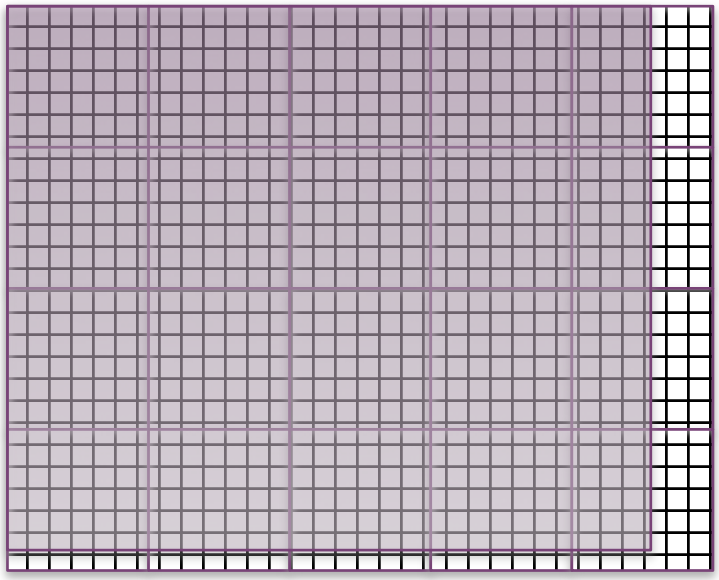


Image size
76 x 62

```
dim3 dimBlock(ceil(n/16.0), ceil(m/16.0), 1);
dim3 dimGrid(16, 16, 1);
pictureKernel<<< dimGrid, dimBlock>> (d_Pin, d_Pout, n, m)
```

gridDim.x = 5, gridDim.y = 4, blockDim.x = 16, blockDim.y = 16
Total number of threads generated = 76 x 62 = 4712

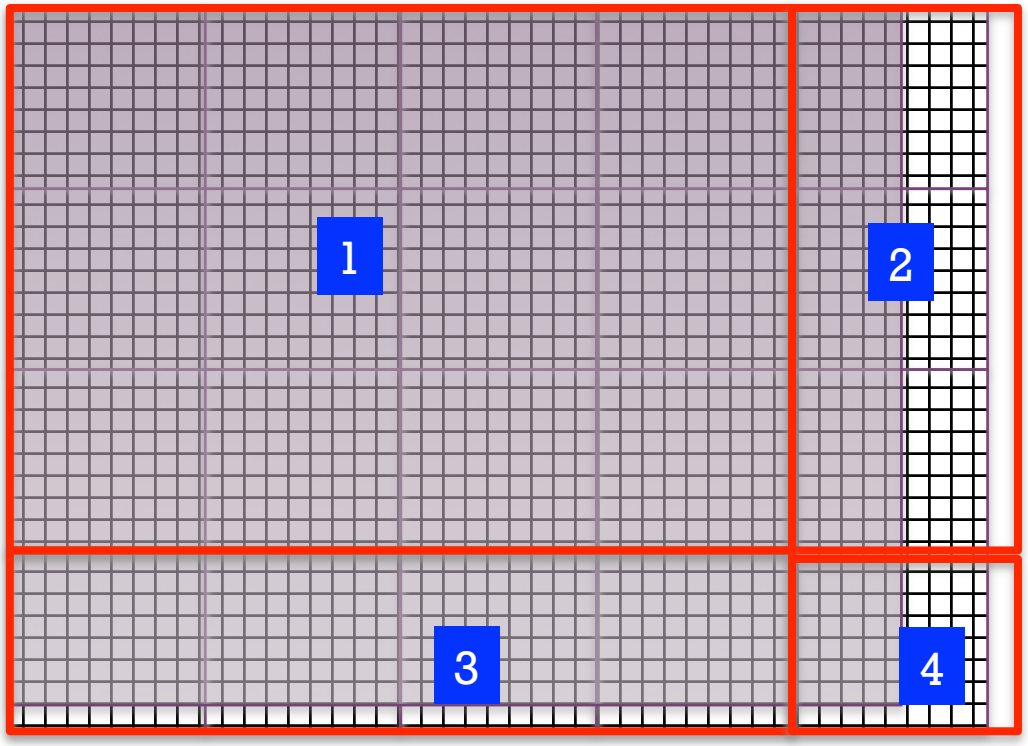
Picture Kernel Code

```
__global__ void PictureKernel(float* d_pin, float* d_Pout, int m, int n){

        // Calculate the row number
        int Row = blockIdx.y * blockDim.y + threadIdx.y;

        // Calculate the column number
        int Col = blockIdx.x * blockDum.x + threadIdx.x;

        //each thread computes one element of d_Pout if in range
        if( (Row < m) && (Col < n))
                d_Pout[Row*n + Col] = 2 * d_Pin[Row*n + Col];

}
```

Source code of PictureKernel() showing 2D thread mapping to a data pattern

```
if( (Row < m) && (Col < n))
```

rows and cols
both in range

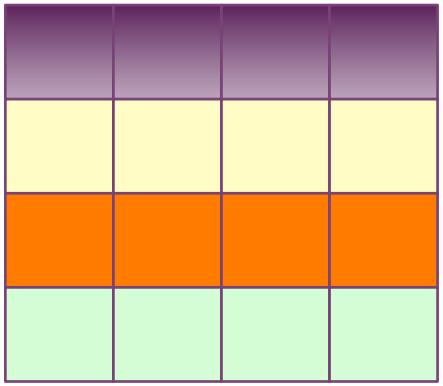


Some cols
Out of range

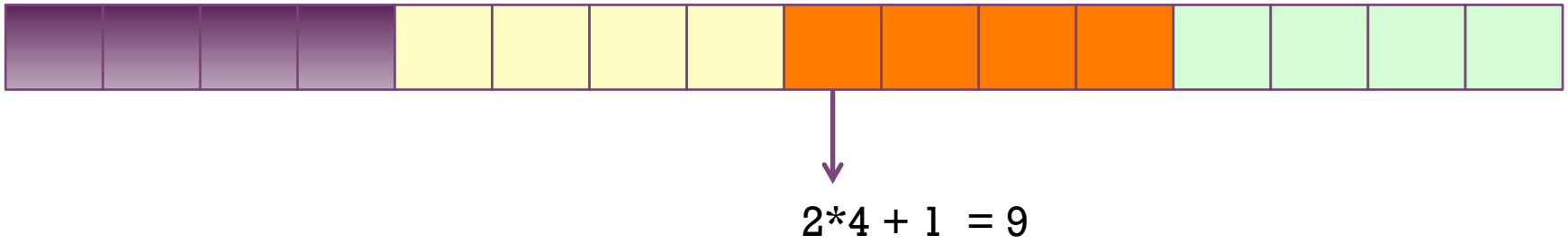
Some rows
And cols out
Of range

Some rows out of range

Row Major indexing



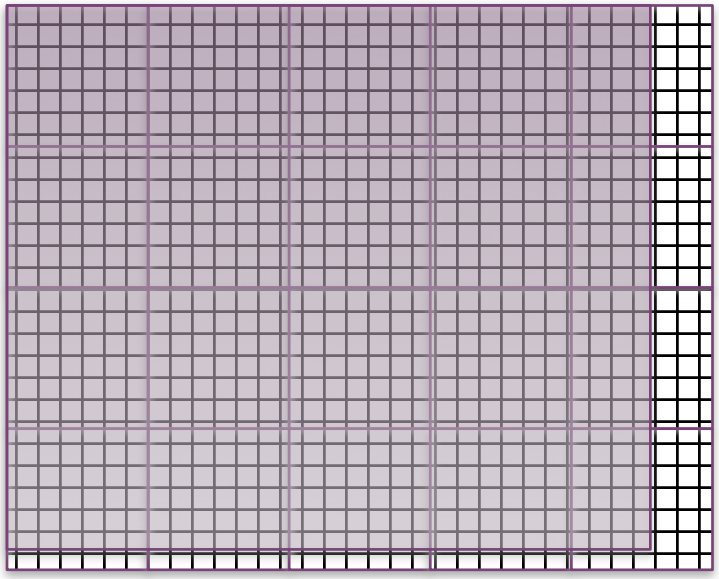
index = row * width + col

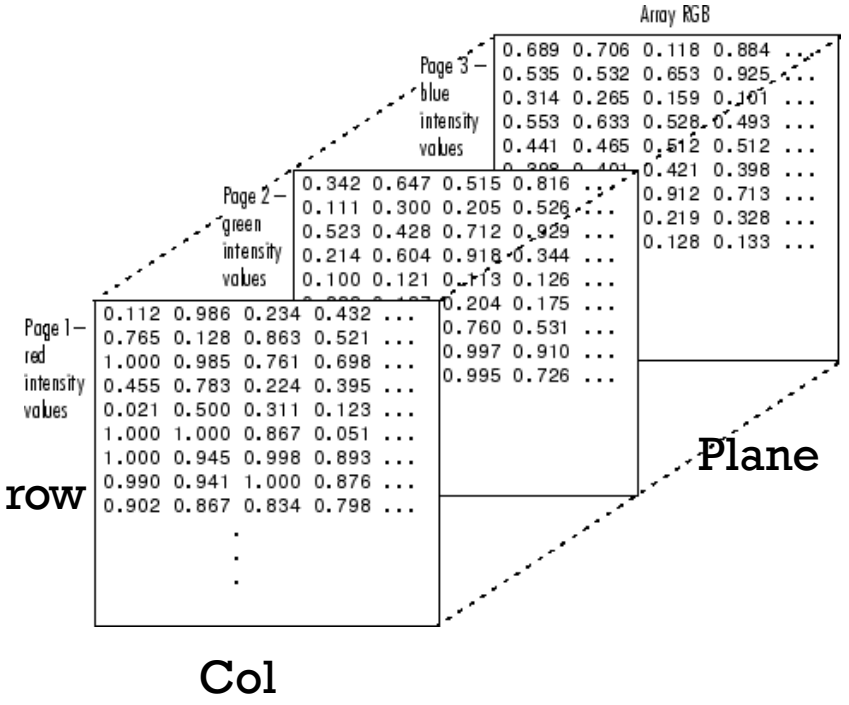


2*4 + 1 = 9

Helpful in dynamic memory allocation.
Address are considered in continuous locations based on datatype
E.g. int = 4 bytes

Handling Data in 3D





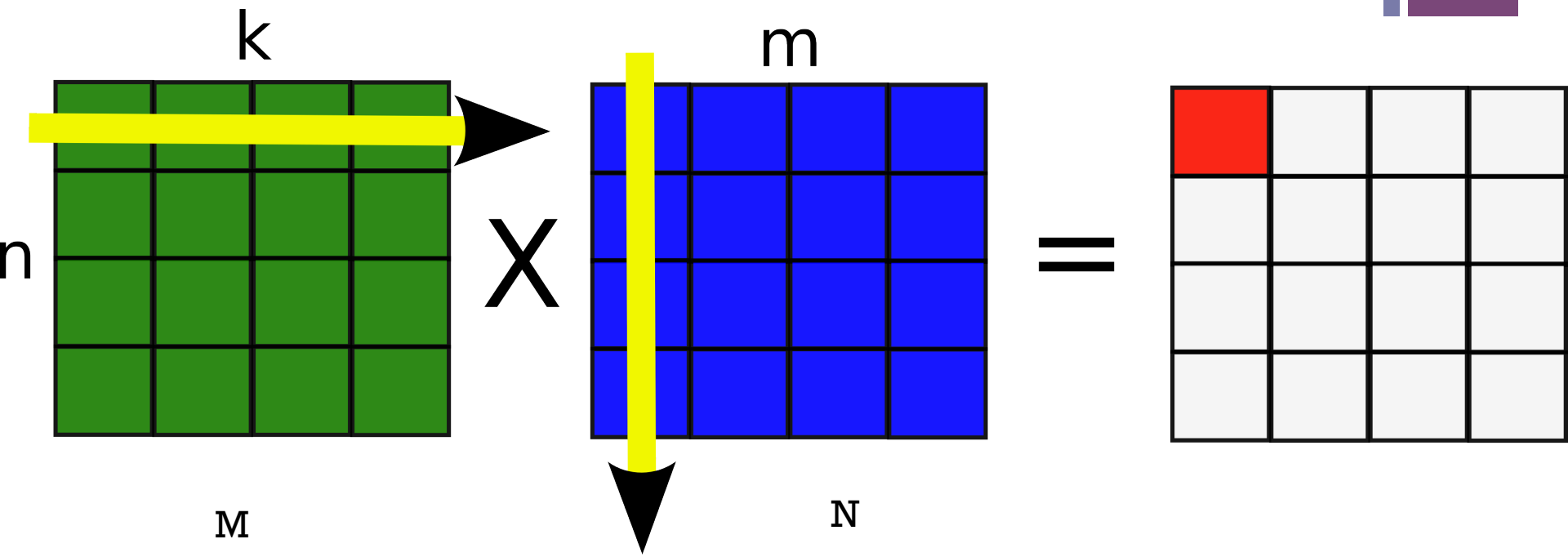
The concept easily extends in 3D. Just have to keep track of Addition dimension

int Plane = blockIdx.z * blockDim.z + threaIdx;

The linearized access to array P will be in the form
P[Plane * m * n + Row*n + Col]

Thus we have to keep track of 3 variables: Plane, row, col

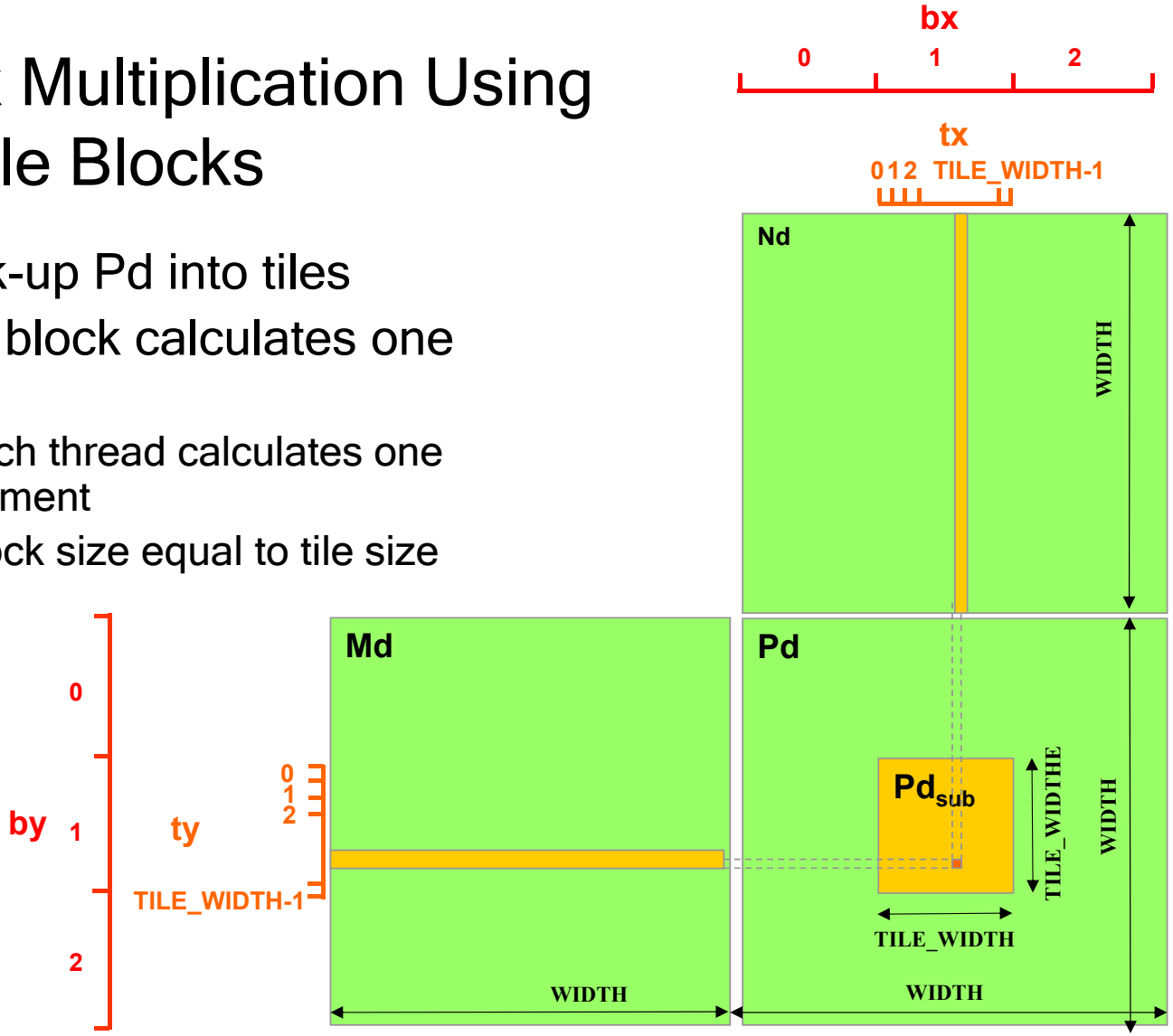
Matrix Multiplication Example



We will consider square matrices only for clarity

Matrix Multiplication Using Multiple Blocks

- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal to tile size



Tile Width = Block width (difference betn book editions)

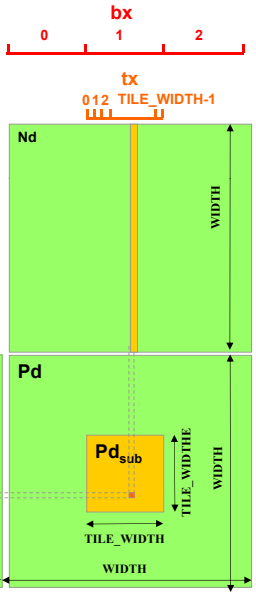
Matrix Multiplication : Thread to Data Mapping

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, int m, int n){

        // Calculate the row number
        int Row = blockIdx.y * blockDim.y + threadIdx.y;

        // Calculate the column number
        int Col = blockIdx.x * blockDum.x + threadIdx.x;

        //each thread computes one element of d_Pout if in range
        if( (Row < m) && (Col < n)){
           float P_value = 0;
           // each thread computes one element of the block submatrix
           for(int k =0; k < width; k++){
                    Pvalue += d_M[Row * Width + k] * d_N[k * Width + Col];
           }
        }
        d_P[Row * width + Col] = Pvalue;

}
```

Why thread to data mapping?

Matrix Multiplication Using Multiple Blocks

- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal to tile size

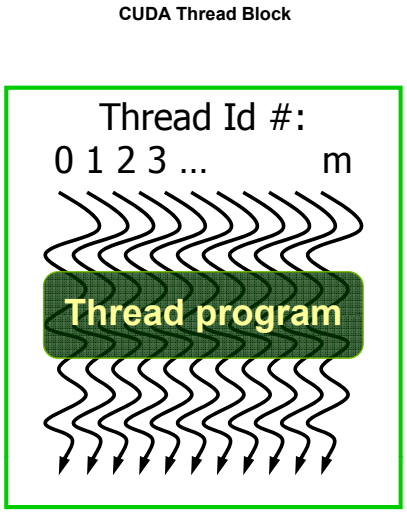


- Divide the computation in tiles/blocks.
- Some block dimensions might be better than others (remember our 76 x 62 image example ?)
- Find the optimal Block/Tile Sizes: "Autotuning" to gain maximum performance gain
- E.g. if we wanted to process matrix of size 1000 x 1000
  - #define BLOCK_SIZE 16 // will generate 64 x 64 blocks
  - #define BLOCK_SIZE 32 // will generate 32 x 32 blocks

Which one to use of the above 2 configuration? (Determined by other parameters as well (like number of streaming multiprocessors etc)

CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocs!

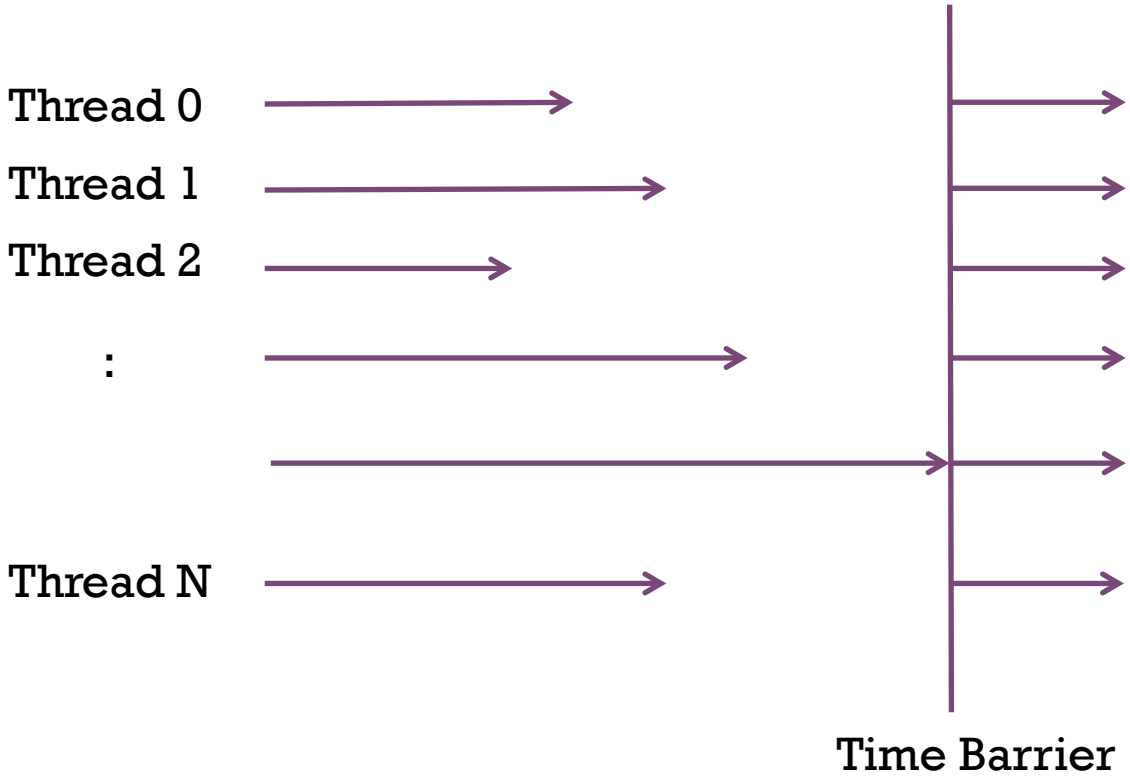


Courtesy: John Nickolls, NVIDIA

Thread Synchronization

- To ensure that all threads in a block have completed a phase of their execution of the kernel before any of them can move on to the next phase
- CUDA follows barrier synchronization: Wait till all threads from the block have completed execution before context/ task switch

Barrier Synchronization



Wait till all the threads complete the given task before proceeding with next operation

Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared
- Allowed in conditional code only if the conditional is uniform across the entire thread block

Host Synchronization

Basics

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- cudaMemcpy() is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete

Host Synchronization Example

Basics

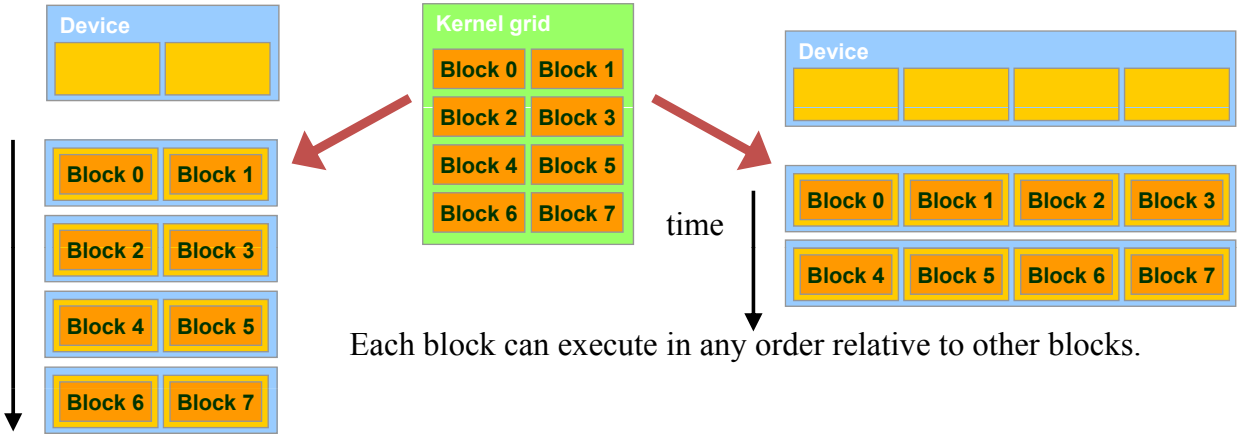
```
// copy data from host to device
cudaMemcpy(a_d, a_h, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
inc_gpu<<<ceil(N/(float)blocksize), blocksize>>>(a_d, N);

// run independent CPU code
run_cpu_stuff();

// copy data from device back to host
cudaMemcpy(a_h, a_d, numBytes, cudaMemcpyDeviceToHost);
```

Thread Synchronization: Points to understand

- CUDA allows thread synchronization within the block but not across blocks
- This means blocks do not have time-dependency on one another: Can be executed in any order
- This flexibility allows scalable implementations
- Enables execution of same code at wide range of speeds (hence same code can be applied to different hardware) e.g. different wait times for resources
- The ability to execute the same application code on hardware with a different number of execution resouces is referred to as transparent scalability

Transparent Scalability

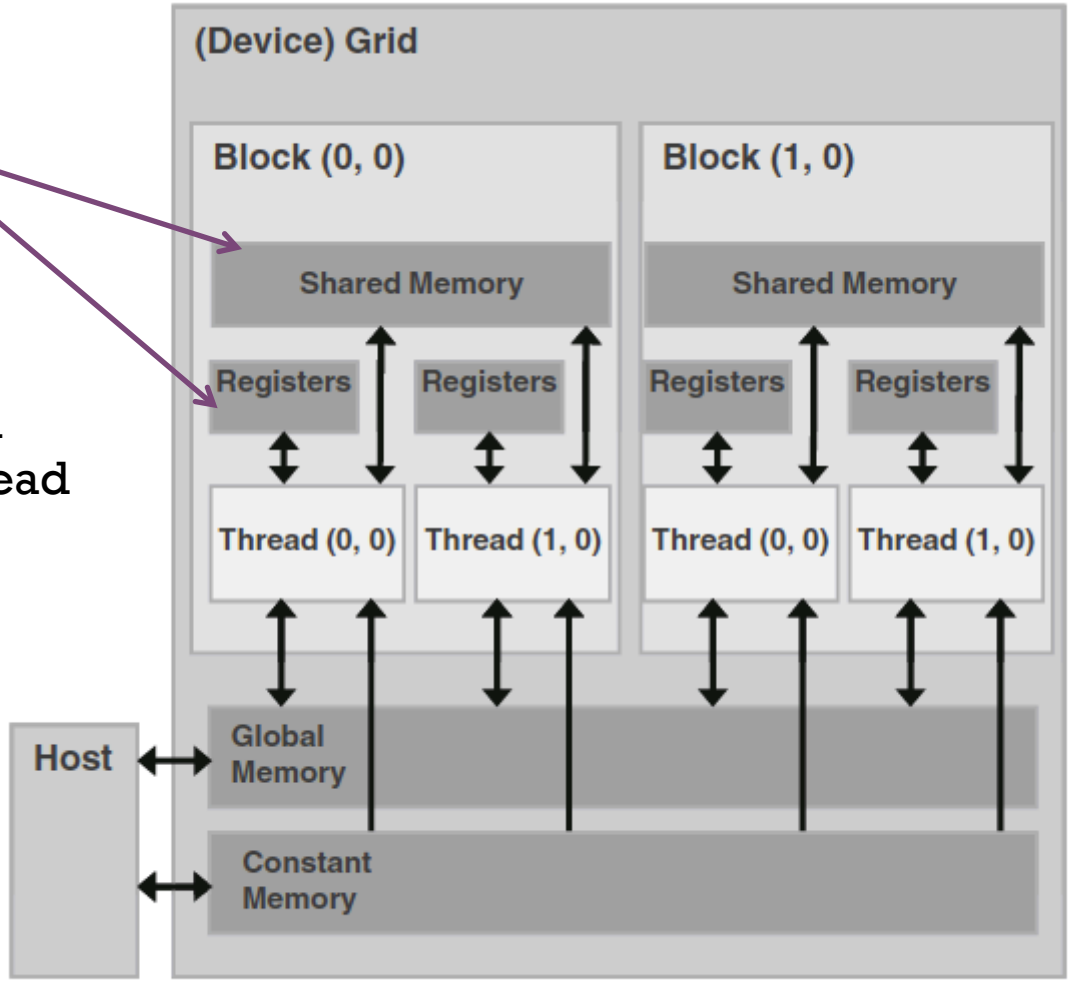
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors



Assigning Resources

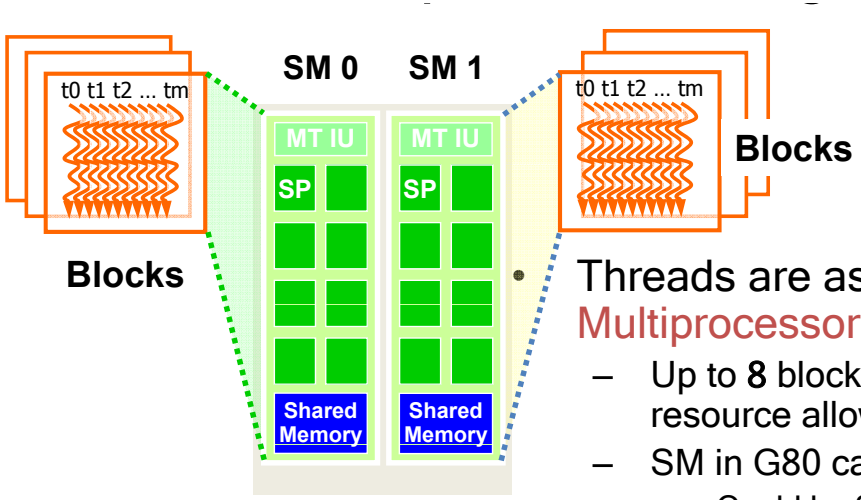
A thread needs resources: memory, registers etc.

These resources hold data/code for the thread to execute



Executing thread blocks

SMs execute the operations in a block

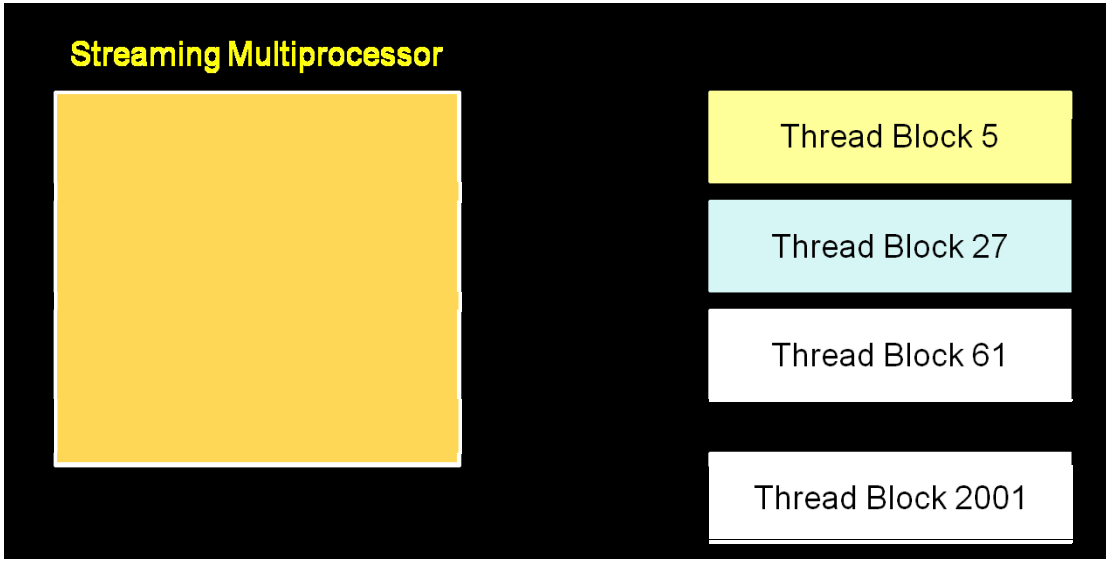


Shared memory contains data/ code for execution for all the blocks within a SM

- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to 8 blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

Execution resources are organized into streaming multiprocessors SMs

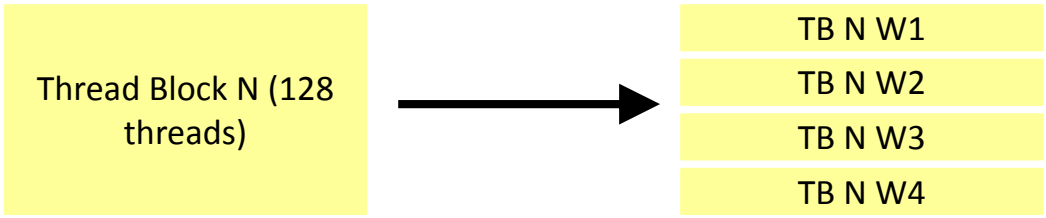
Scheduling Blocks onto SMs



- HW Schedules thread blocks onto available SMs
  - No guarantee of ordering among thread blocks
  - HW will schedule thread blocks as soon as a previous thread block finishes

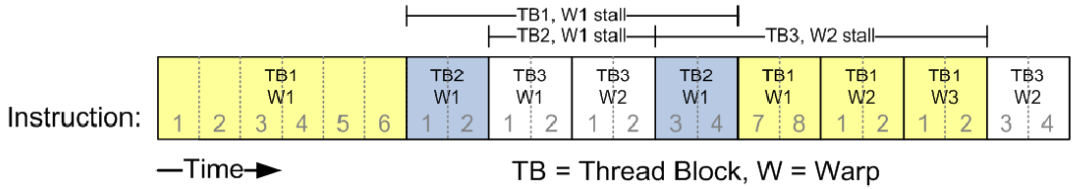
Mapping of Thread Blocks

- Each thread block is mapped to one or more warps
- The hardware schedules each warp independently



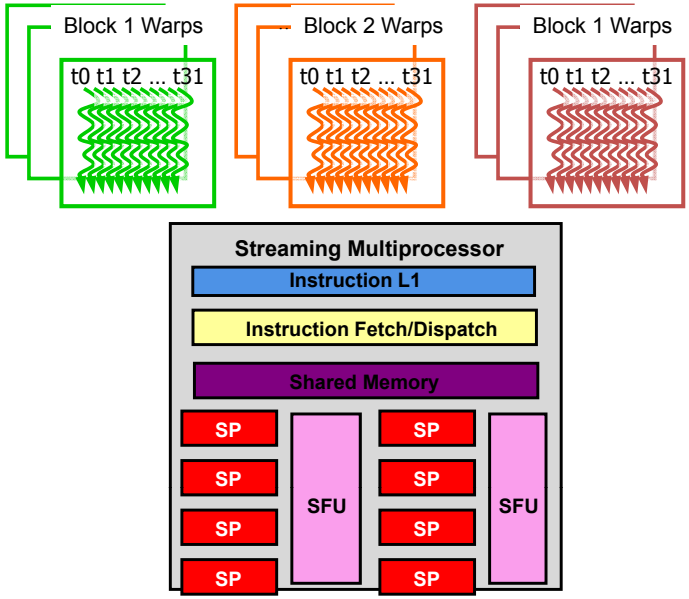
Thread Scheduling Example

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected



Thread Scheduling

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
- 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



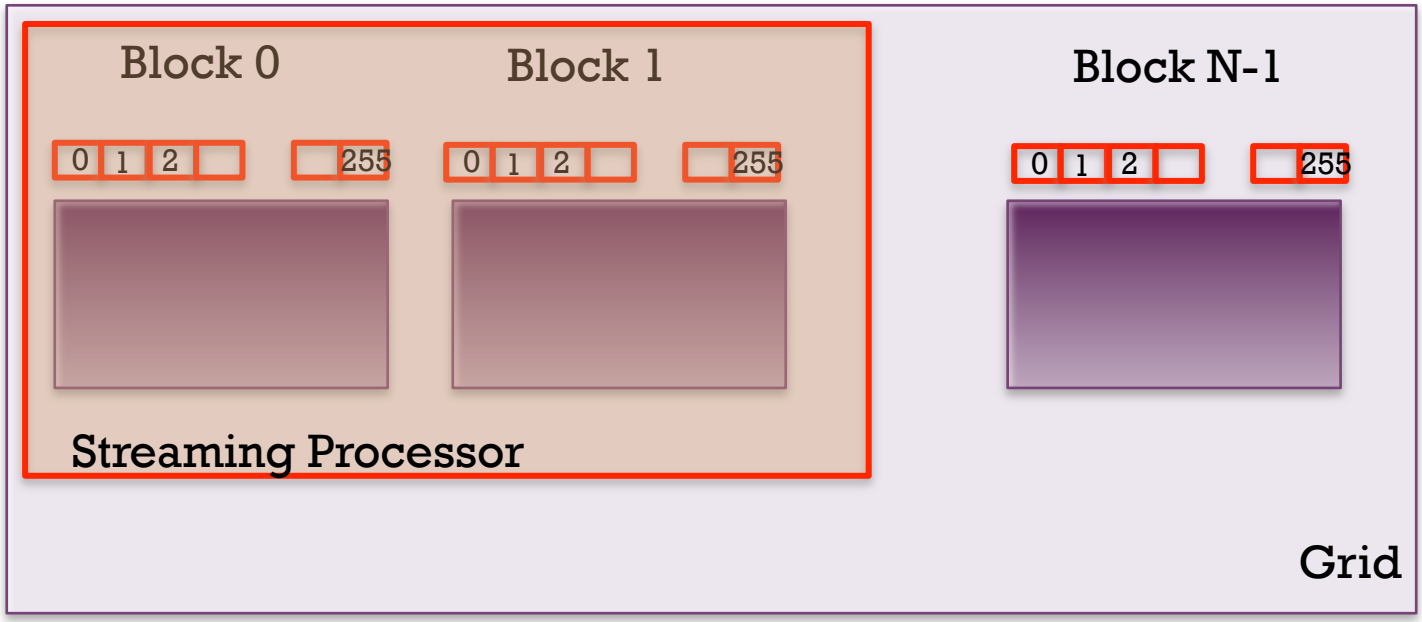
G80 Example: Thread Scheduling (Cont.)

- 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 on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

G80 Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!

Recap



Grid: Total number of threads
Block: Organization for threads
Streaming Processors: Resource allocation for Threads
Grids, blocks, threads can have multiple dimensions

Device and Resource Query

- How to find device configurations?
- How many SMs? How many threads per block?

```
cudaDeviceProp dev_prop;
for(i = 0; I < dev_count; i++){
          cudaGetDeviceProperties(&dev_prop, i);
          //decide if device has sufficient resources
}

// cudaDeviceProp is a C structure
```

cudaDeviceProp Struct Reference

[Data types used by CUDA Runtime]

Data Fields

int	asyncEngineCount
int	canMapHostMemory
int	clockRate
int	computeMode
int	concurrentKernels
int	deviceOverlap
int	ECCEnabled
int	integrated
int	kernelExecTimeoutEnabled
int	l2CacheSize
int	major
int	maxGridSize [3]
int	maxSurface1D
int	maxSurface1DLayered [2]
int	maxSurface2D [2]
int	maxSurface2DLayered [3]
int	maxSurface3D [3]
int	maxSurfaceCubemap
int	maxSurfaceCubemapLayered [2]
int	maxTexture1D
int	maxTexture1DLayered [2]
int	maxTexture1DLinear
int	maxTexture2D [2]
int	maxTexture2DGather [2]
int	maxTexture2DLayered [3]
int	maxTexture2DLinear [3]
int	maxTexture3D [3]
int	maxTextureCubemap
int	maxTextureCubemapLayered [2]
int	maxThreadsDim [3]
int	maxThreadsPerBlock
int	maxThreadsPerMultiProcessor
int	memoryBusWidth
int	memoryClockRate
size_t	memPitch
int	minor
int	multiProcessorCount
char	name [256]
int	pciBusID
int	pciDeviceID
int	pciDomainID
int	regsPerBlock
size_t	sharedMemPerBlock
size_t	surfaceAlignment
int	tccDriver
size_t	textureAlignment
size_t	texturePitchAlignment
size_t	totalConstMem
size_t	totalGlobalMem
int	unifiedAddressing
int	warpSize

Device Management

- CPU can query and select GPU devices
  - cudaGetDeviceCount(int* count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *current_device)
  - cudaGetDeviceProperties(cudaDeviceProp* prop, int device)
  - cudaChooseDevice(int *device, cudaDeviceProp* prop)
- Multi-GPU setup:
  - device 0 is used by default
  - one CPU thread can control one GPU
    - multiple CPU threads can control the same GPU
      - calls are serialized by the driver

Kepler GK110 supports the new CUDA Compute Capability 3.5. (For a brief overview of CUDA see ***Appendix A - Quick Refresher on CUDA***). The following table compares parameters of different Compute Capabilities for Fermi and Kepler GPU architectures:

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

Specifications

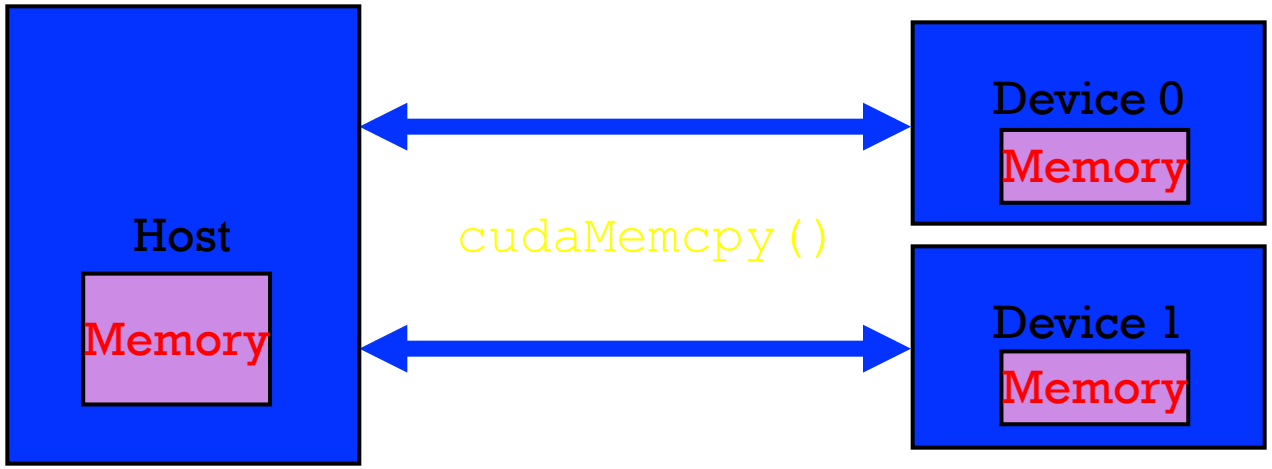
Note: The below specifications represent this GPU as incorporated into NVIDIA's reference graphics card design. Graphics card specifications may vary by Add-in-card manufacturer. Please refer to the Add-in-card manufacturers' website for actual shipping specifications.

GTX 780 GPU Engine Specs:	
CUDA Cores	2304
Base Clock (MHz)	863
Boost Clock (MHz)	900
Texture Fill Rate (billion/sec)	160.5
GTX 780 Memory Specs:	
Memory Speed	6.0 Gbps
Standard Memory Config	3072 MB
Memory Interface	GDDR5
Memory Interface Width	384-bit
Memory Bandwidth (GB/sec)	288.4
GTX 780 Support:	
Important Technologies	GPU Boost 2.0, PhysX, TXAA, NVIDIA G-SYNC-ready, SHIELD-ready
Other Supported Technologies	3D Vision, CUDA, DirectX 11, Adaptive VSync, FXAA, 3D Vision Surround, SLI-ready
OpenGL	4.3
Bus Support	PCI Express 3.0
Certified for Windows 7, Windows 8, Windows Vista, or Windows XP	Yes
3D Vision Ready	Yes
3D Gaming	Yes
Blu Ray 3D	Yes
3D Vision Live (Photos and Videos)	Yes

CUDA Memories

```
float *h_A, *h_B, *h_C
malloc(); // assign memory
free();   // free memory
```

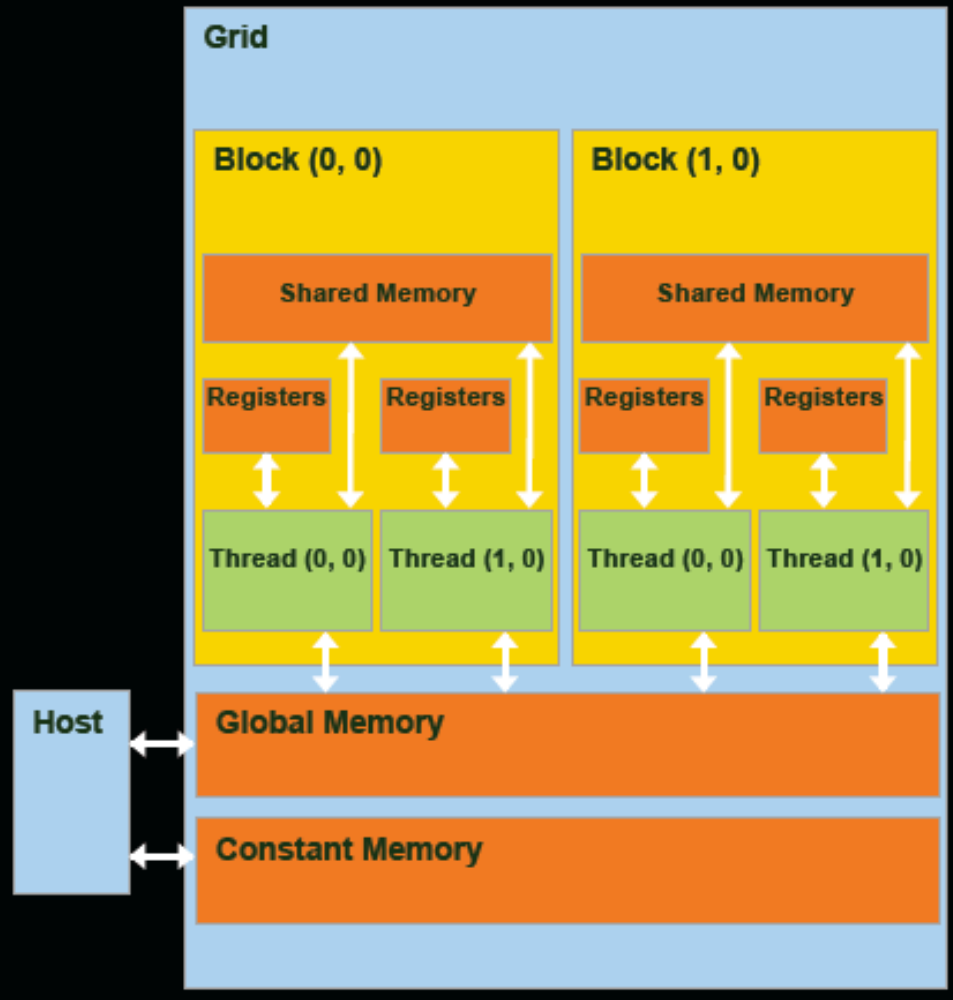
```
float *d_A, *d_B, *d_C
cudaMalloc()
cudaFree()
```



We need to assign memory in the device (GPU) for the Variables that we wish to use in the device

Hardware Implementation of CUDA Memories

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read/only per-grid constant memory



Importance of Memory Access Efficiency

```
for(int k = 0; k < Width; k++)
        Pvalue += d_M[Row * Width + k] + d_N[k*Width + Col]
```

Every iteration has 2 global memory access for one floating point addition and one floating point multiplication.
Thus it has *Compute to global memory access ratio (CGMA) is 1:1*

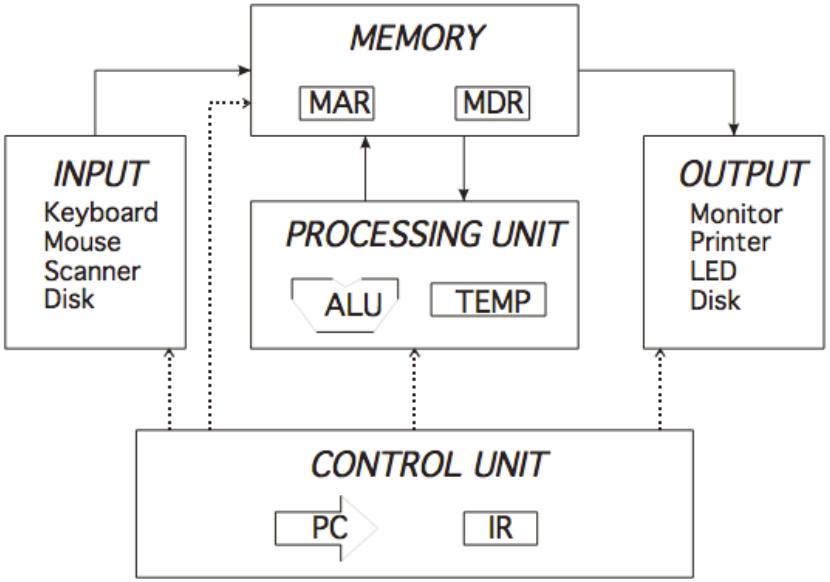
It has major performance implications:
Eg: Memory Bandwidth: 200 GB/s
Floating point size: 4 Bytes. Therefore 50 Gigs single precision operands/sec
i.e. it will execute at the max 50 GFLOPS.

Peak performance usually at 1500 GFLOPS (1.5 TFLOPS)
Only way to get around this is to increase CGMA ratio i.e

REDUCE MEMORY ACCESS

Lets understand threads in detail

- Thread is a virtualized von Neuman processor
- In von Neuman model, code of program is stored in memory, PC keeps track of particular point of the program, IR has instructions, Registers and memory holds value of variables and data structure



Processing Units and Threads

- Modern processors are designed to allow context switching, where multiple threads can time-share processor
- During context switch, intermediate values are saved in registers/memory
- GPUs allow multiple processors, single instruction i.e all processors execute same instructions. Hence, resource sharing between threads is important.
- The reason why threads are organized into blocks/warps

CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	constant	grid	application

- **"automatic" scalar variables without qualifier reside in a register**
  - **compiler will spill to thread local memory**
- **"automatic" array variables without qualifier reside in thread-local memory**

CUDA Variable Type Performance

Variable declaration	Memory	Penalty
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

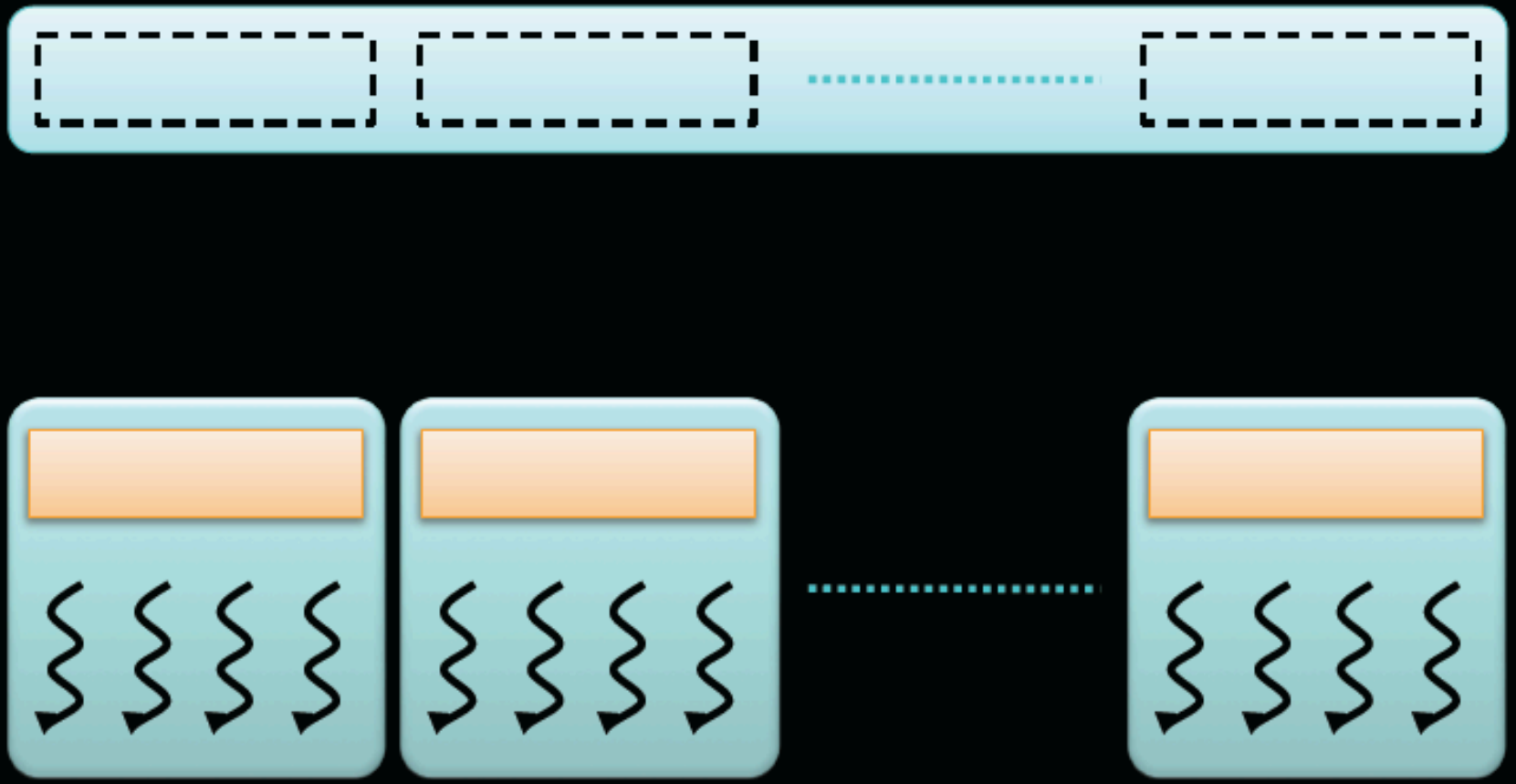
A Common Programming Strategy

- Global memory resides in device memory (DRAM)
  - Much slower access than shared memory
- Tile data to take advantage of fast shared memory:
  - Generalize from `adjacent_difference` example
  - Divide and conquer

A Common Programming Strategy

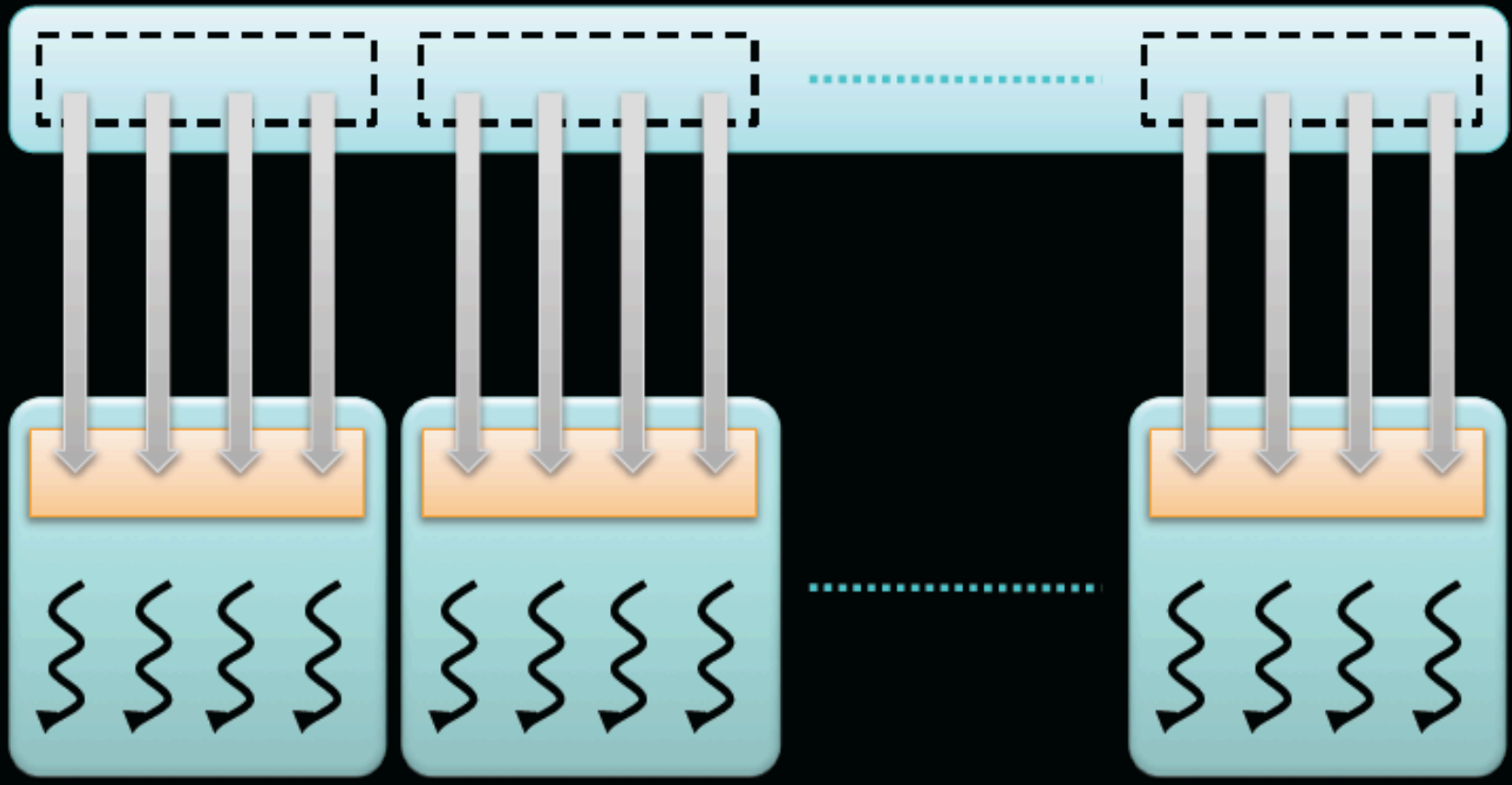
- Partition data into subsets that fit into shared memory

A Common Programming Strategy



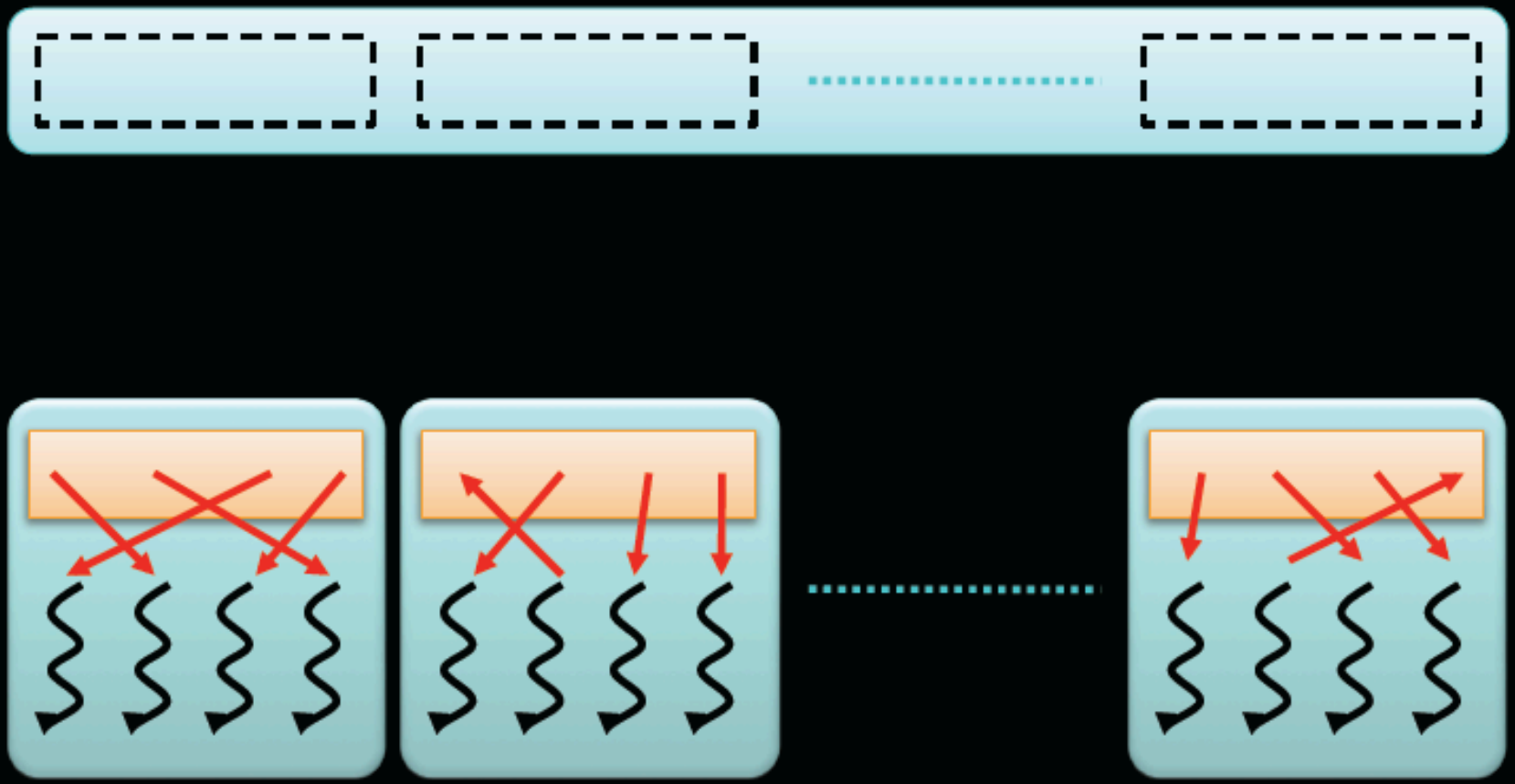
- Handle each data subset with one thread block

A Common Programming Strategy



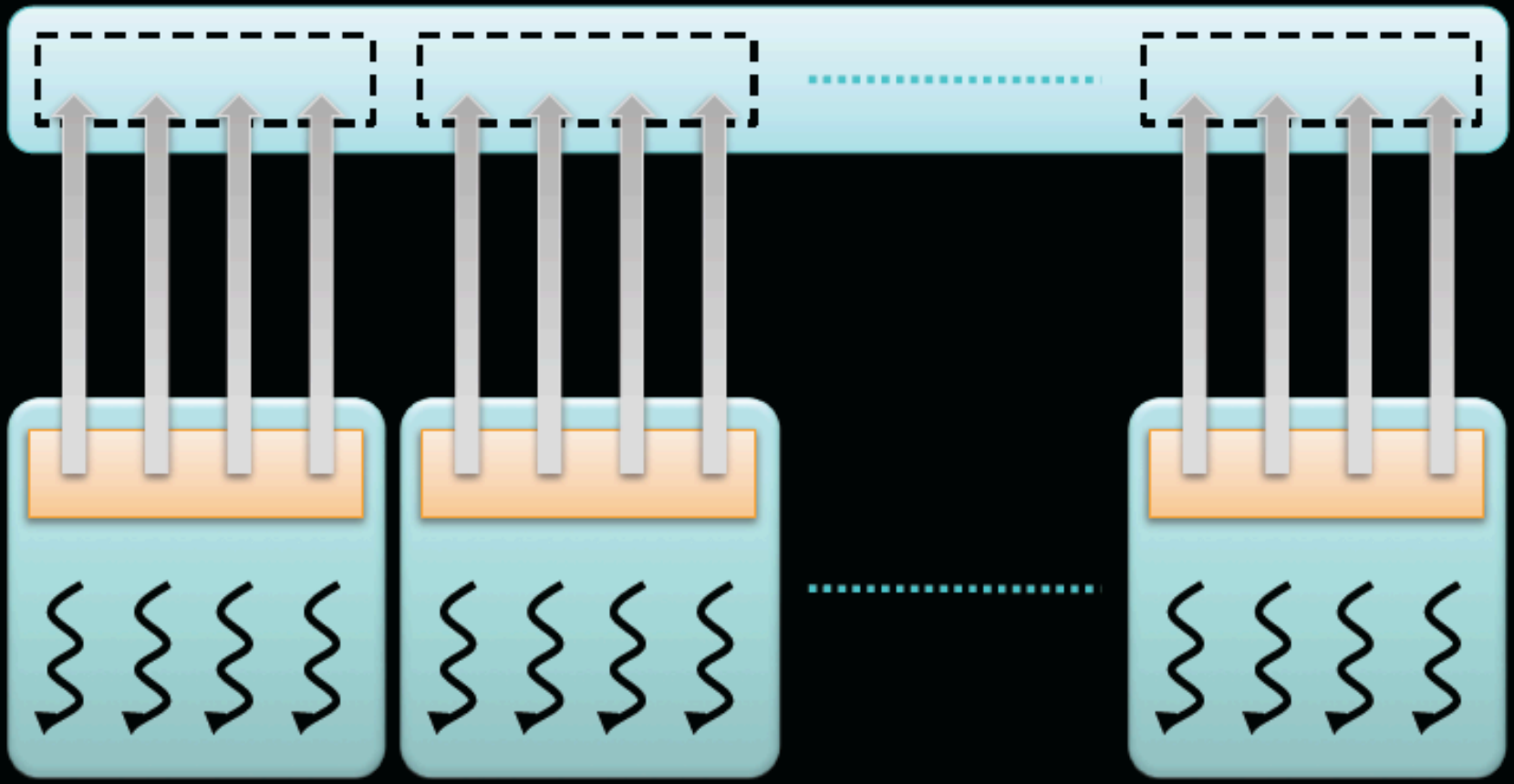
- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

A Common Programming Strategy



- Perform the computation on the subset from shared memory

A Common Programming Strategy

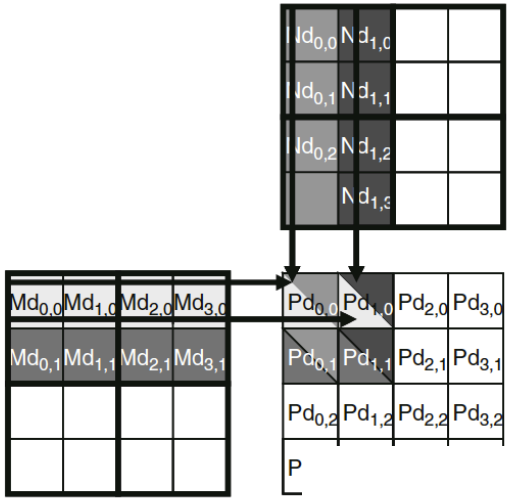


- Copy the result from shared memory back to global memory

A Common Programming Strategy

- Carefully partition data according to access patterns
- Read-only → `__constant__` memory (fast)
- R/W & shared within block → `__shared__` memory (fast)
- R/W within each thread → registers (fast)
- Indexed R/W within each thread → local memory (slow)
- R/W inputs/results → `cudaMalloc`'ed global memory (slow)

Tiled Matrix Multiplication

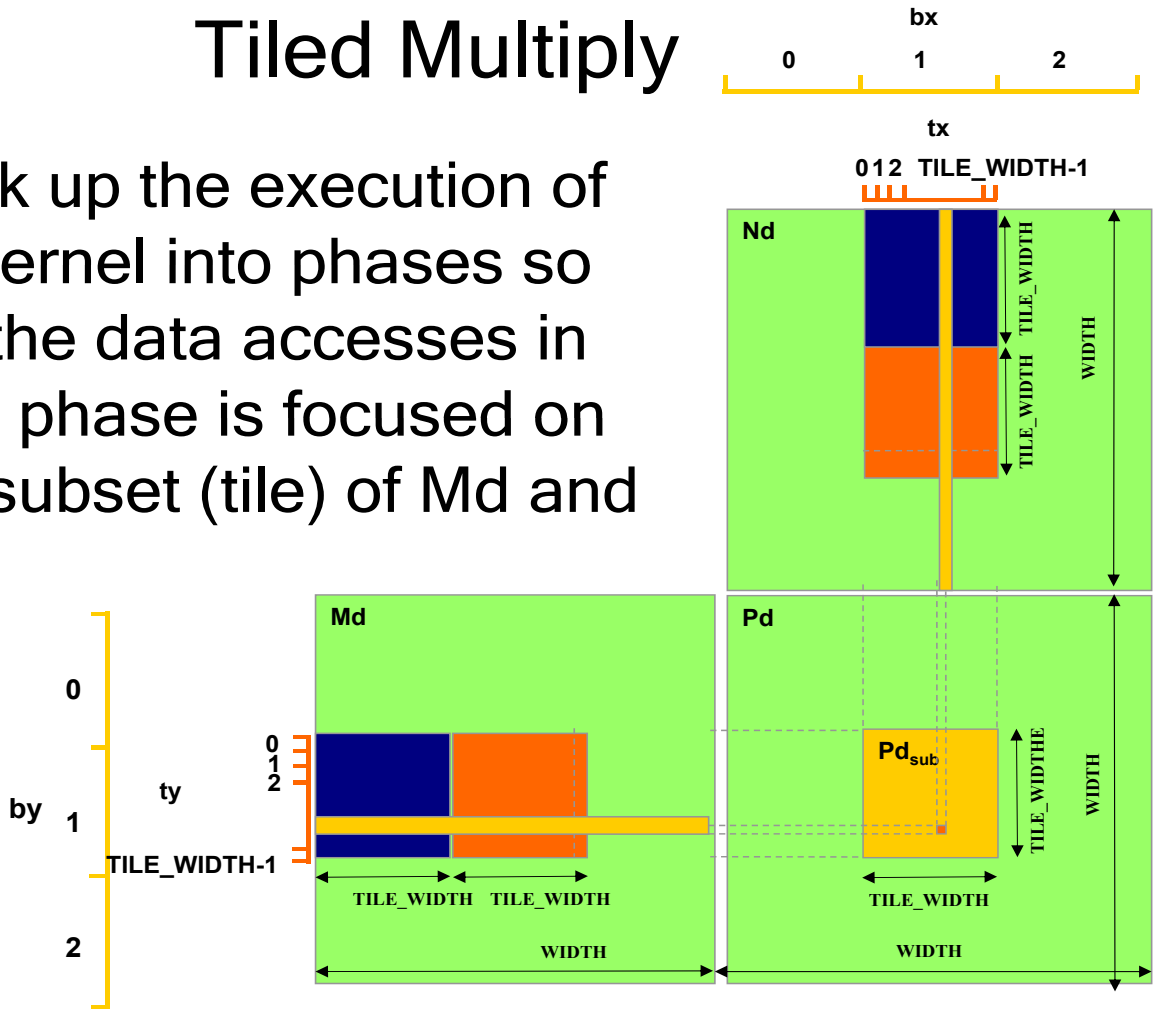


	Phase 1			Phase 2		
$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + Mds_{11}*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

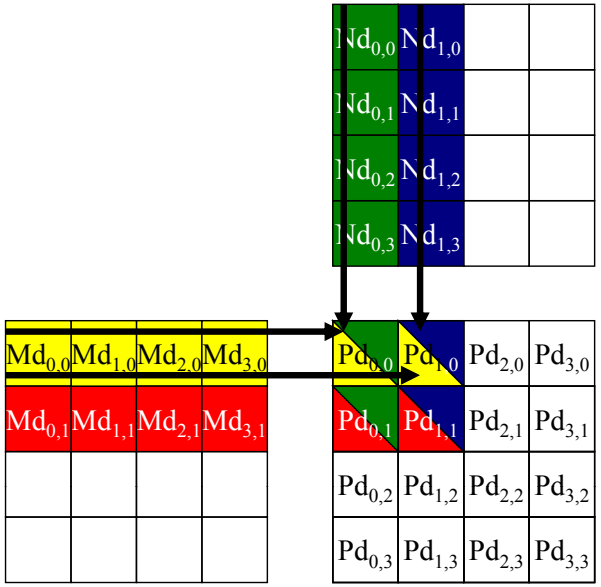
time →

Tiled Multiply

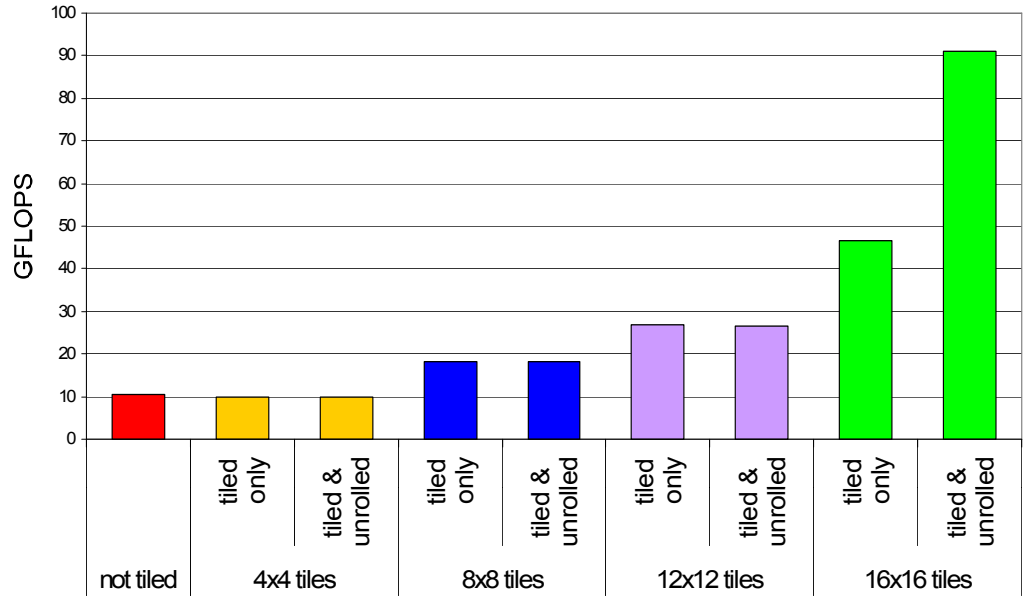
- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd



A Small Example: Multiplication



Tiling Size Effects



```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.   float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collaborative loading of Md and Nd tiles into shared memory
9.    Mds[ty][tx] = Md[Row*Width  + (m*TILE_WIDTH  +  tx)];
10.   Nds[ty][tx] = Nd[(m*TILE_WIDTH  + ty)*Width + Col];
11.   __syncthreads();

12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.     Pvalue += Mds[ty][k] * Nds[k][tx];
14.   __syncthreads();
    }
15. Pd[Row*Width + Col] = Pvalue;
}
```

Points to Remember about CUDA Memory

- CUDA offers local memories/registers. Using these efficiently reduces access to global memory and improves performance. But it requires algorithm re-design
- Blocks, Registers have limited memory. If data exceeds this shared memory requirement, data has to be split into multiple pieces
- The ability to reason about hardware limitation when developing an application is key aspect of computational thinking
- Tiled algorithms often increase performance. But key to it is to exploit data locality

Accelerating MATLAB with CUDA

- Massimiliano Fatica
- NVIDIA
- mfatica@nvidia.com

Won-Ki Jeong

University of Utah

wkjeong@cs.utah.edu

Overview

MATLAB can be easily extended via MEX files to take advantage of the computational power offered by the latest NVIDIA GPUs (GeForce 8800, Quadro FX5600, Tesla).

Programming the GPU for computational purposes was a very cumbersome task before CUDA. Using CUDA, it is now very easy to achieve impressive speed-up with minimal effort.

This work is a proof of concept that shows the feasibility and benefits of using this approach.

MEX file

- Even though MATLAB is built on many well-optimized libraries, some functions can perform better when written in a compiled language (e.g. C and Fortran).

- MATLAB provides a convenient API for interfacing code written in C and FORTRAN to MATLAB functions with MEX files.

- MEX files could be used to exploit multi-core processors with OpenMP or threaded codes or like in this case to offload functions to the GPU.

NVMEX

- Native MATLAB script cannot parse CUDA code

- New MATLAB script nvmex.m compiles CUDA code (.cu) to create MATLAB function files

- Syntax similar to original mex script:

```
>> nvmex –f nvmexopts.bat filename.cu –IC:\cuda\include
        –LC:\cuda\lib -lcudart
```

Available for Windows and Linux from:

http://developer.nvidia.com/object/matlab_cuda.html

Mex files for CUDA

A typical mex file will perform the following steps:

1. Convert from double to single precision
2. Rearrange the data layout for complex data
3. Allocate memory on the GPU
4. Transfer the data from the host to the GPU
5. Perform computation on GPU (library, custom code)
6. Transfer results from the GPU to the host
7. Rearrange the data layout for complex data
8. Convert from single to double
9. Clean up memory and return results to MATLAB

Some of these steps will go away with new versions of the library (2,7) and new hardware (1,8)

CUDA MEX example

Additional code in MEX file to handle CUDA

```
/*Parse input, convert to single precision and to interleaved complex format */
        .....
/* Allocate array on the GPU */
        cufftComplex *rhs_complex_d;
        cudaMalloc( (void **) &rhs_complex_d,sizeof(cufftComplex)*N*M);
/* Copy input array in interleaved format to the GPU */
        cudaMemcpy( rhs_complex_d, input_single, sizeof(cufftComplex)*N*M,                cudaMemcpyHostToDevice);
/* Create plan for CUDA FFT  NB: transposing dimensions*/
        cufftPlan2d(&plan, N, M, CUFFT_C2C) ;
/* Execute FFT on GPU */
        cufftExecC2C(plan, rhs_complex_d, rhs_complex_d, CUFFT_INVERSE) ;
/* Copy result back to host */
        cudaMemcpy( input_single, rhs_complex_d, sizeof(cufftComplex)*N*M,                cudaMemcpyDeviceToHost);
/* Clean up memory and plan on the GPU */
        cufftDestroy(plan); cudaFree(rhs_complex_d);
/*Convert back to double precision and to split complex format */
```

Initial study

- Focus on 2D FFTs.
- FFT-based methods are often used in single precision (for example in image processing)
- Mex files to overload MATLAB functions, no modification between the original MATLAB code and the accelerated one.
- Application selected for this study:

  solution of the Euler equations in vorticity form using a pseudo-spectral method.

Implementation details:

Case A) FFT2.mex and IFFT2.mex

Mex file in C with CUDA FFT functions.

Standard mex script could be used.

Overall effort: few hours

Case B) Szeta.mex: Vorticity source term written in CUDA

Mex file in CUDA with calls to CUDA FFT functions.

Small modifications necessary to handle files with a .cu suffix

Overall effort: ½ hour (starting from working mex file for 2D FFT)

Configuration

Hardware:

AMD Opteron 250 with 4 GB of memory

NVIDIA GeForce 8800 GTX

Software:

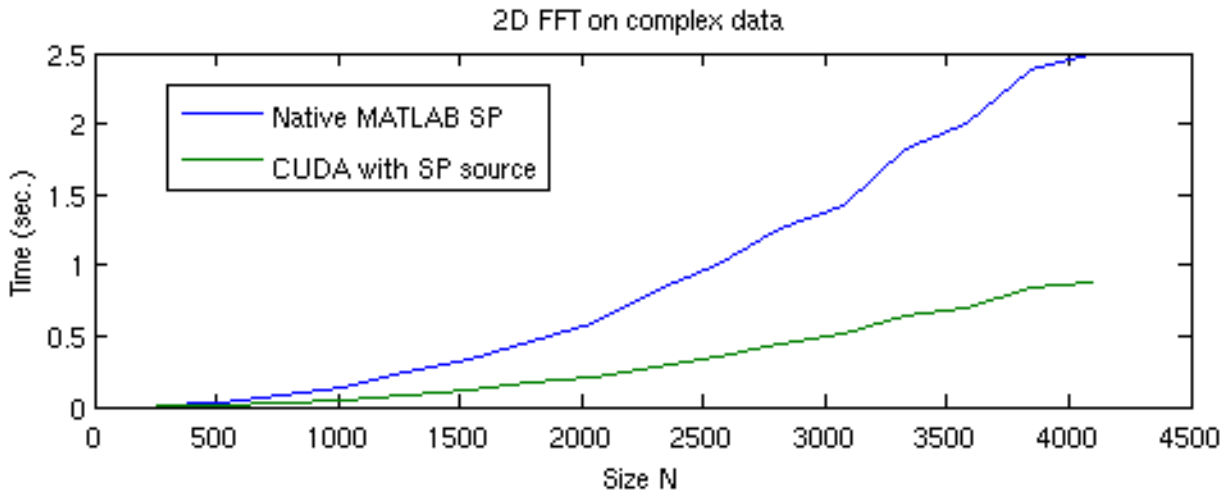
Windows XP and Microsoft VC8 compiler

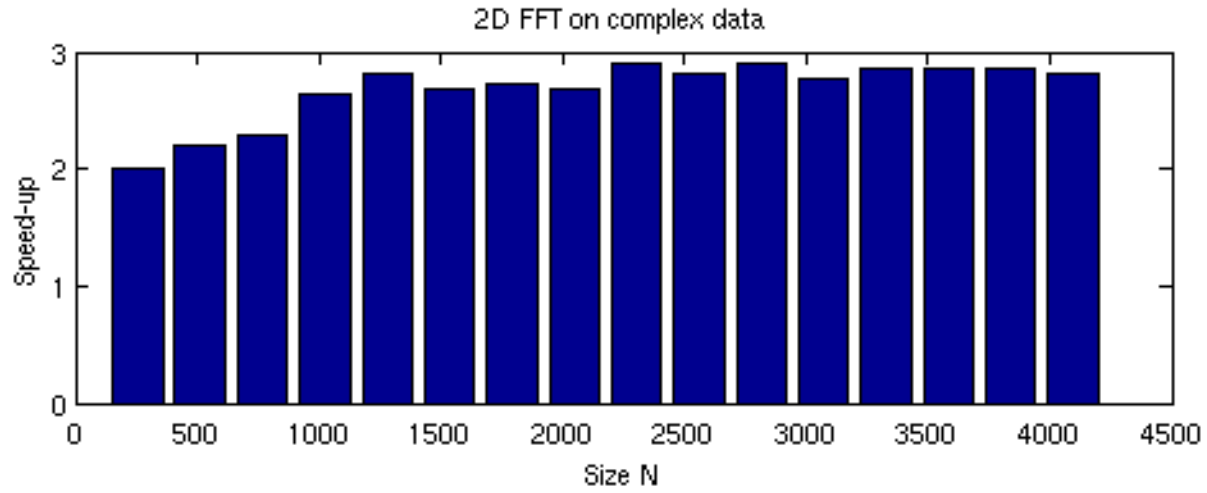
RedHat Enterprise Linux 4 32 bit, gcc compiler

MATLAB R2006b

CUDA 1.0

FFT2 performance





Vorticity source term

http://www.amath.washington.edu/courses/571-winter-2006/matlab/Szeta.m

```
function S = Szeta(zeta,k,nu4)



% Pseudospectral calculation of vorticity source term

% S = -(- psi_y*zeta_x + psi_x*zeta_y) + nu4*del^4 zeta

% on a square periodic domain, where zeta = psi_xx + psi_yy is an NxN matrix

% of vorticity and k is vector of Fourier wavenumbers in each direction.

% Output is an NxN matrix of S at all pseudospectral gridpoints

        zetahat = fft2(zeta);

        [KX KY] = meshgrid(k,k);

% Matrix of (x,y) wavenumbers corresponding

% to Fourier mode (m,n)

        del2 = -(KX.^2 + KY.^2);

        del2(1,1) = 1; % Set to nonzero to avoid division by zero when inverting

% Laplacian to get psi

        psihat = zetahat./del2;

        dpsidx = real(ifft2(1i*KX.*psihat));

        dpsidy = real(ifft2(1i*KY.*psihat));
```

Caveats

The current CUDA FFT library only supports interleaved format for complex data while MATLAB stores all the real data followed by the imaginary data.

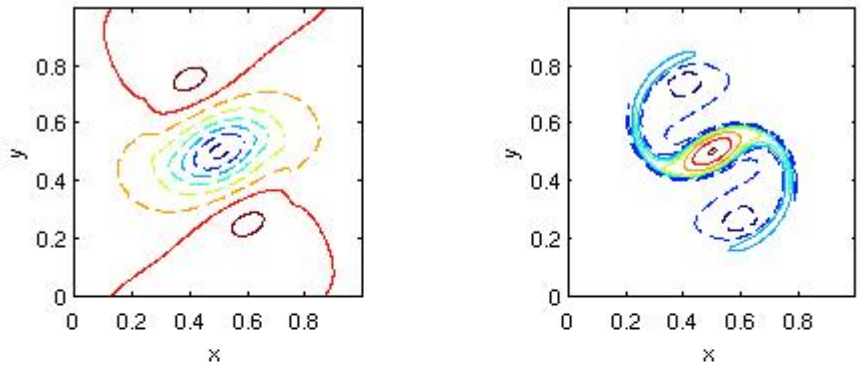
Complex to complex (C2C) transforms used

The accelerated computations are not taking advantage of the symmetry of the transforms.

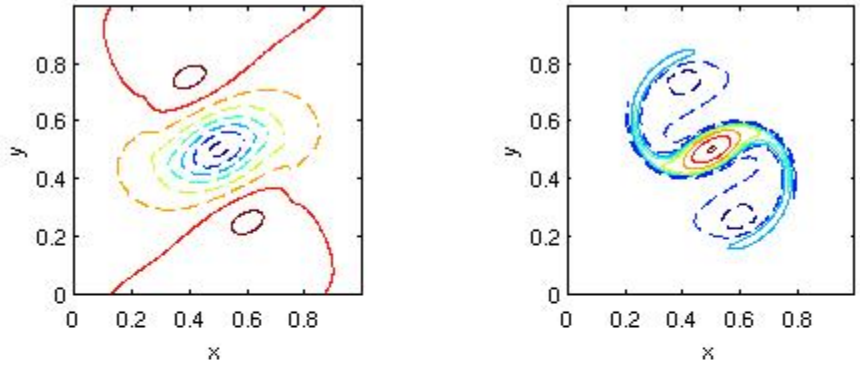
The current GPU hardware only supports single precision (double precision will be available in the next generation GPU towards the end of the year). Conversion to/from single from/to double is consuming a significant portion of wall clock time.

Advection of an elliptic vortex

256x256 mesh, 512 RK4 steps, Linux, MATLAB file
http://www.amath.washington.edu/courses/571-winter-2006/matlab/FS_vortex.m



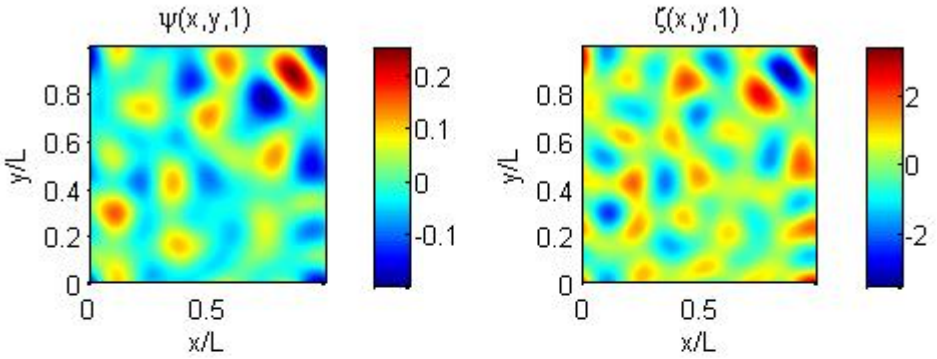
MATLAB
168 seconds



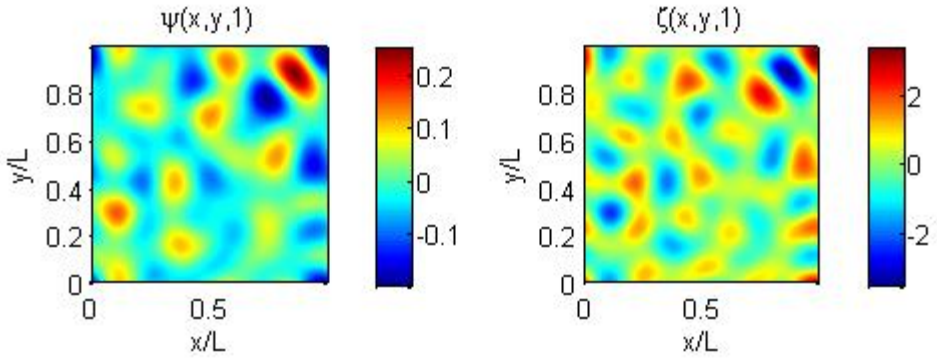
MATLAB with CUDA
(single precision FFTs)
14.9 seconds (11x)

Pseudo-spectral simulation of 2D Isotropic turbulence.

512x512 mesh, 400 RK4 steps, Windows XP, MATLAB file
http://www.amath.washington.edu/courses/571-winter-2006/matlab/FS_2Dturb.m

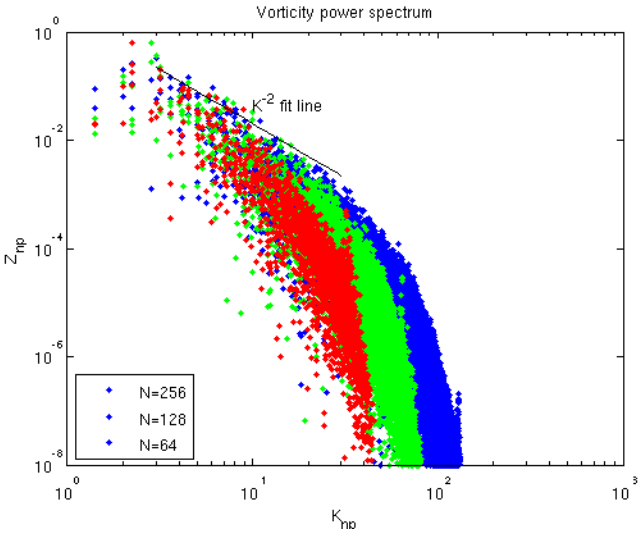


MATLAB
992 seconds

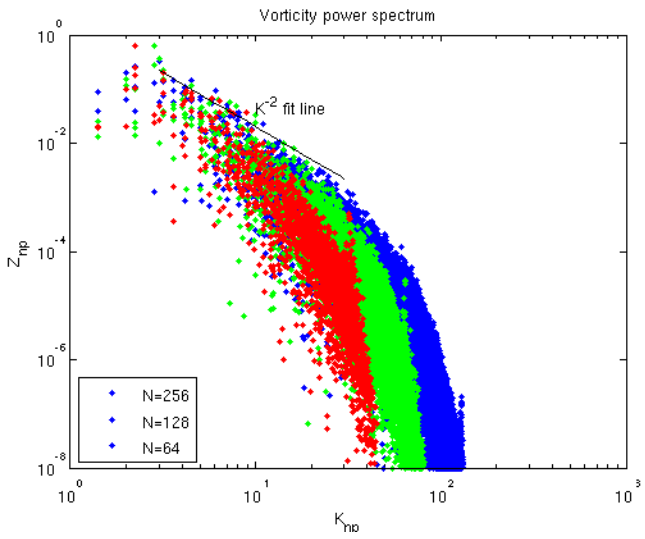


MATLAB with CUDA
(single precision FFTs)
93 seconds

- Power spectrum of vorticity is very sensitive to fine scales. Result from original MATLAB run and CUDA accelerated one are in excellent agreement



MATLAB run



CUDA accelerated MATLAB run

Timing details

1024x1024 mesh, 400 RK4 steps on Windows, 2D isotropic turbulence

	Runtime Opteron 250	Speed up	Runtime Opteron 2210	Speed up
PCI-e Bandwidth: Host to/from device	1135 MB/s 1003 MB/s		1483 MB/s 1223 MB/s	
Standard MATLAB	8098 s		9525s	
Overload FFT2 and IFFT2	4425 s	1.8x	4937s	1.9x
Overload Szeta	735 s	11.x	789s	12.X
Overload Szeta , FFT2 and IFFT2	577 s	14.x	605s	15.7x