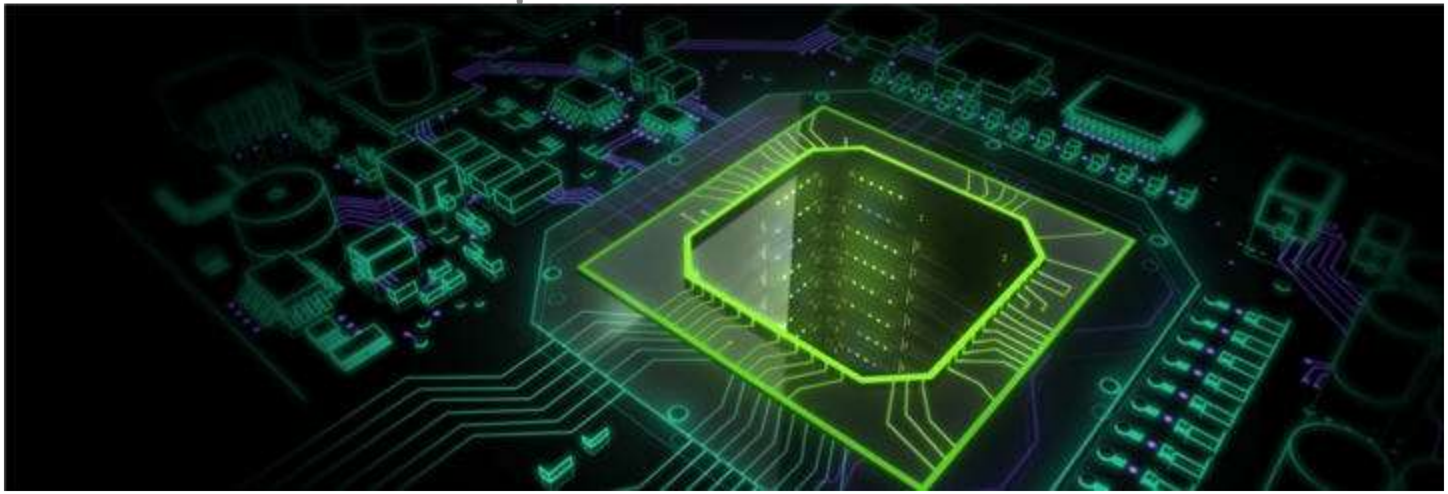
CSCI-GA.3033-012

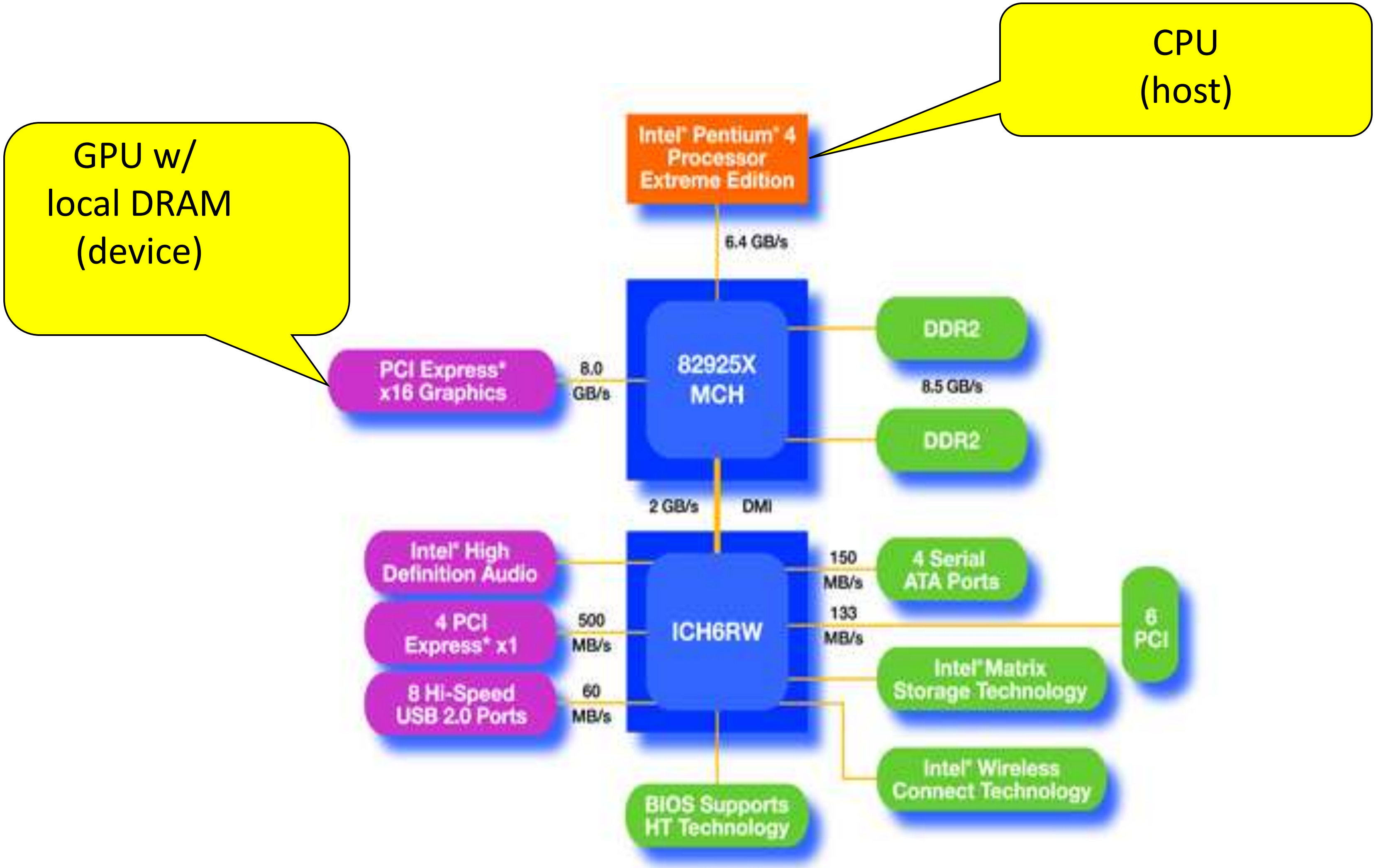
Graphics Processing Units (GPUs): Architecture and Programming

Lecture 4: CUDA Programming Model

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



Behind CUDA



GPU w/
local DRAM
(device)
CPU
(host)
Intel® Pentium® 4 Processor Extreme Edition
6.4 GB/s
82925X MCH
DDR2
8.5 GB/s
DDR2
PCI Express* x16 Graphics
8.0 GB/s
2 GB/s
DMI
ICH6RW
Intel® High Definition Audio
4 PCI Express* x1
500 MB/s
8 Hi-Speed USB 2.0 Ports
60 MB/s
150 MB/s
4 Serial ATA Ports
133 MB/s
6 PCI
Intel® Matrix Storage Technology
Intel® Wireless Connect Technology
BIOS Supports HT Technology

Parallel Computing on a GPU

- 8-series GPUs deliver 25 to 200+ GFLOPS
  on compiled parallel C applications
  - Available in laptops, desktops, and clusters

- GPU parallelism is doubling every year
- Programming model scales transparently

- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application
  data parallelism and thread parallelism

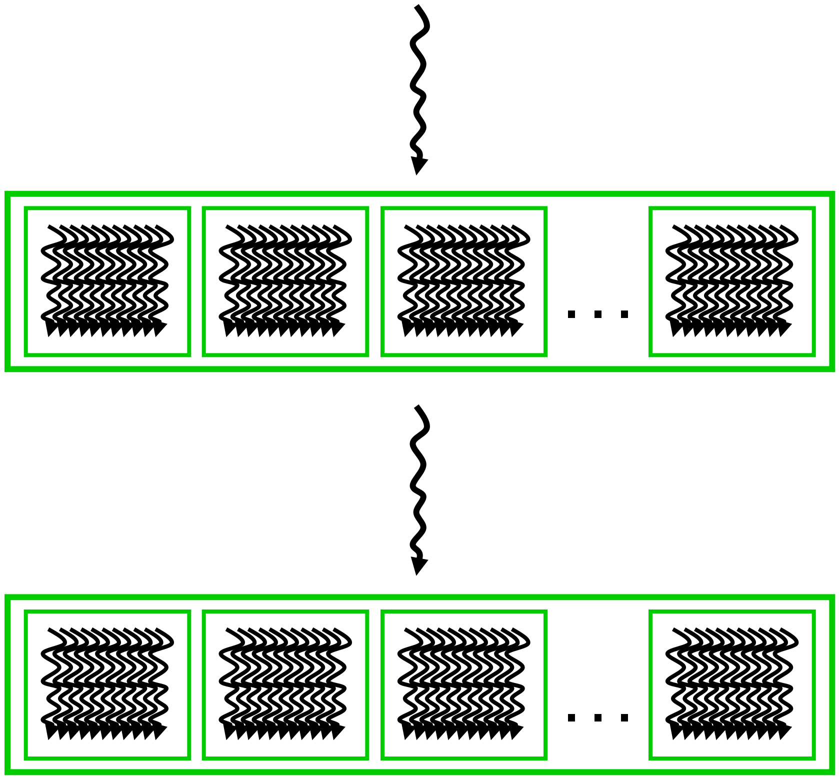
GeForce 8800

Tesla D870

Tesla S870

CUDA

- Compute Unified Device Architecture
- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code



Serial Code (host)

Parallel Kernel (device)

KernelA<<< nBlk, nTid >>>(args);

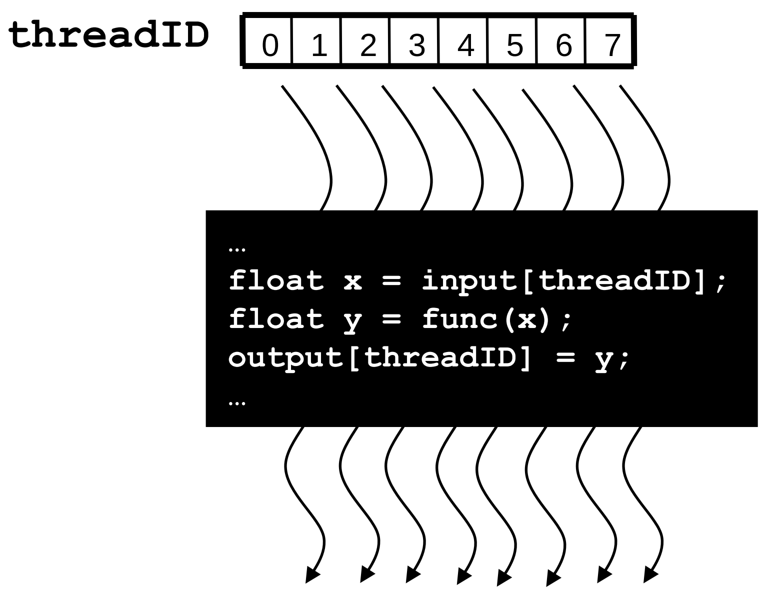
Serial Code (host)

Parallel Kernel (device)

KernelB<<< nBlk, nTid >>>(args);

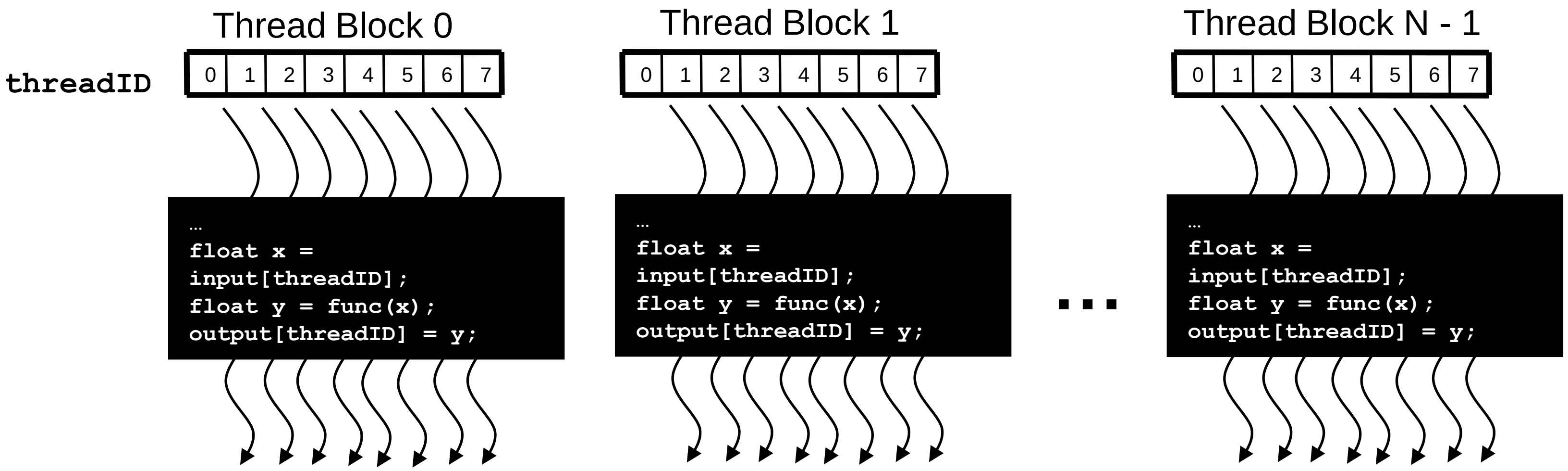
Parallel Threads

- A CUDA kernel is executed by an array of threads
  – All threads run the same code (SPMD)
  – Each thread has an ID that it uses to compute memory addresses and make control decisions



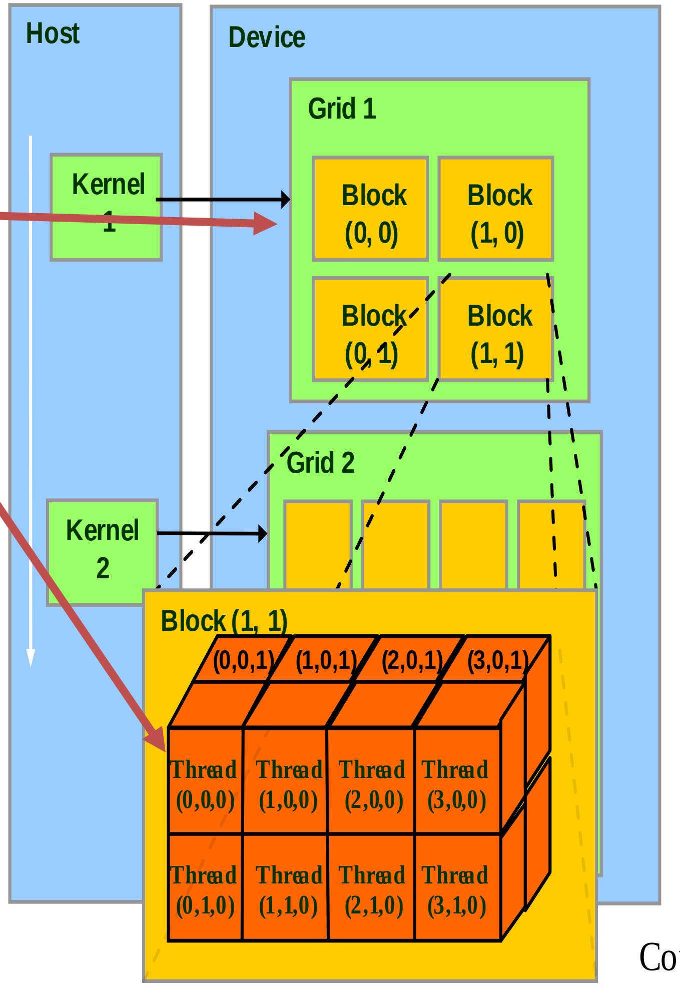
Thread Blocks

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate



IDs

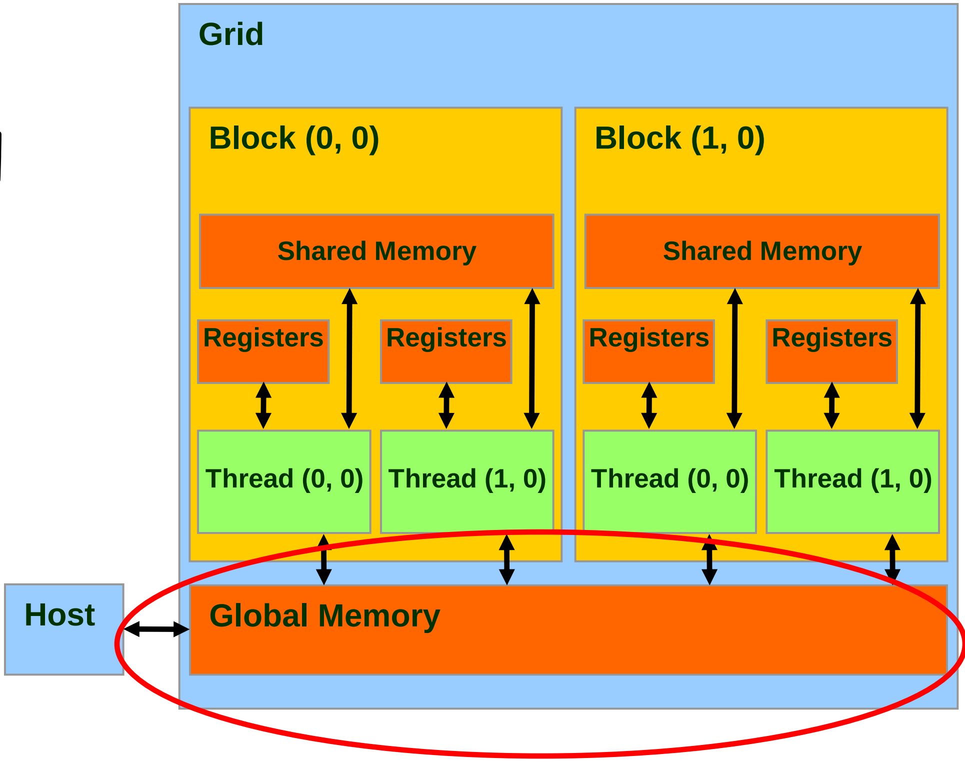
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

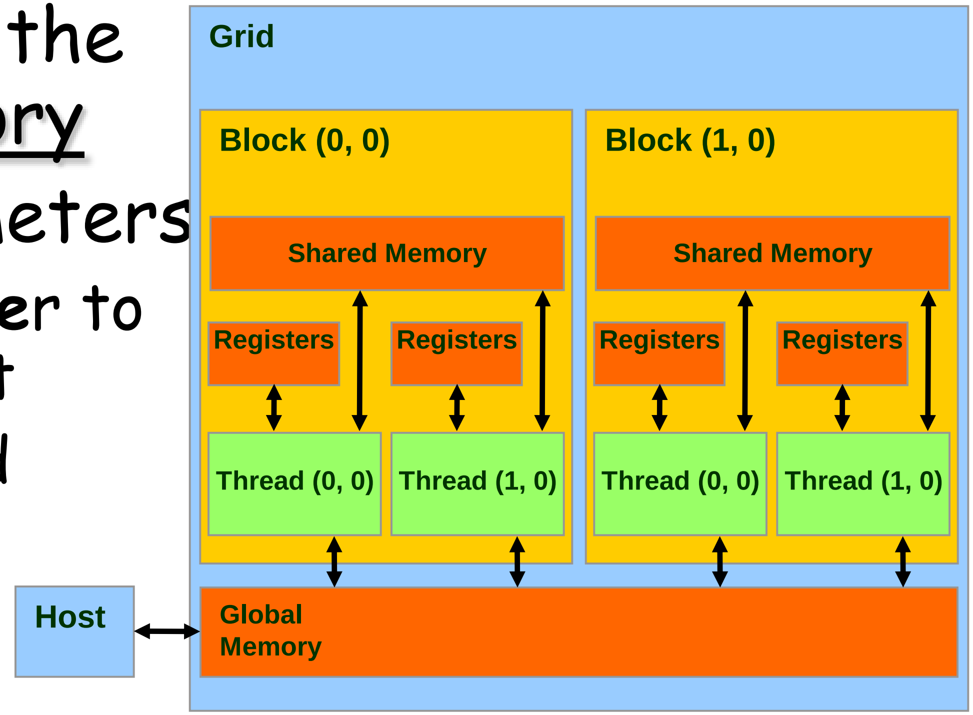
CUDA Memory Model

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
  - Constant and texture memory will come later



CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device <u>Global Memory</u>
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object

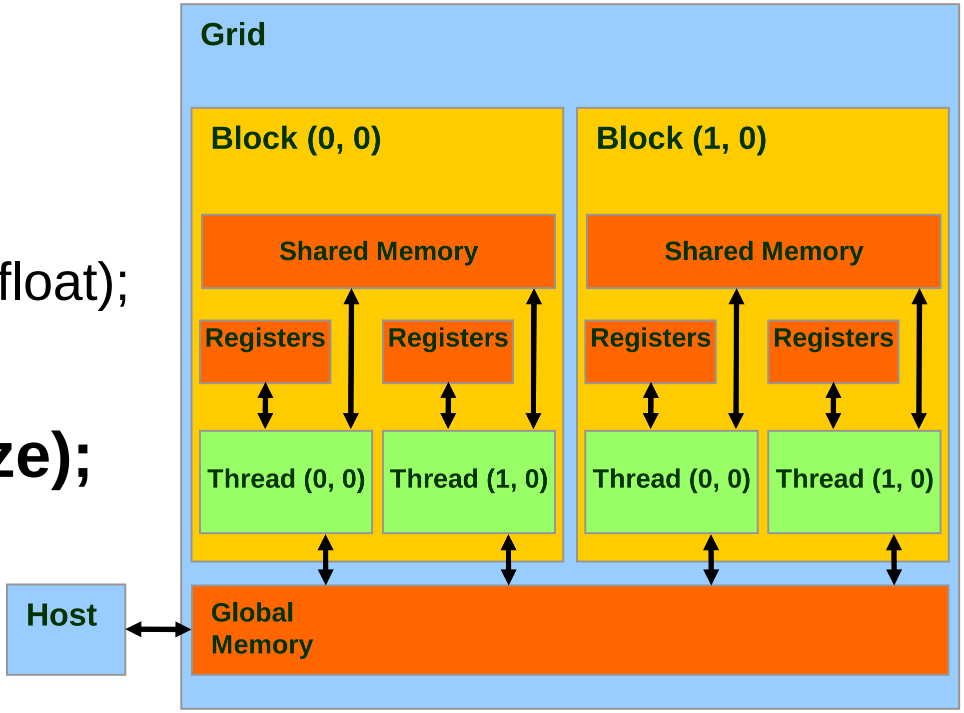


CUDA Device Memory Allocation

Example:

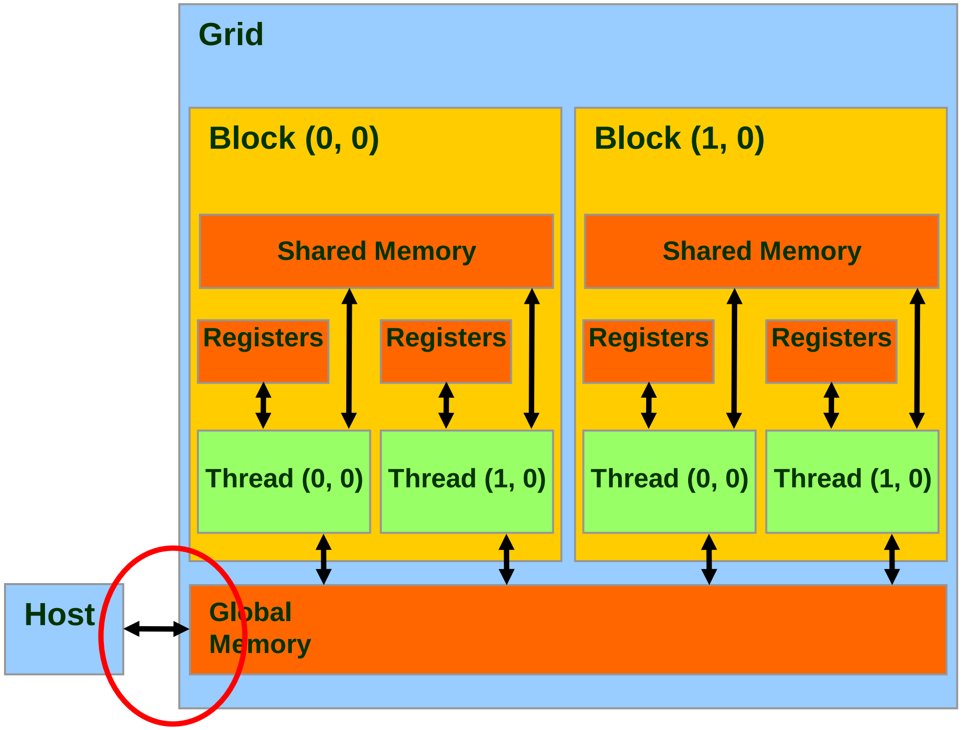
WIDTH = 64;
float* Md
int size = WIDTH * WIDTH * sizeof(float);

cudaMalloc((void)&Md, size);**
cudaFree(Md);



CUDA Device Memory Allocation

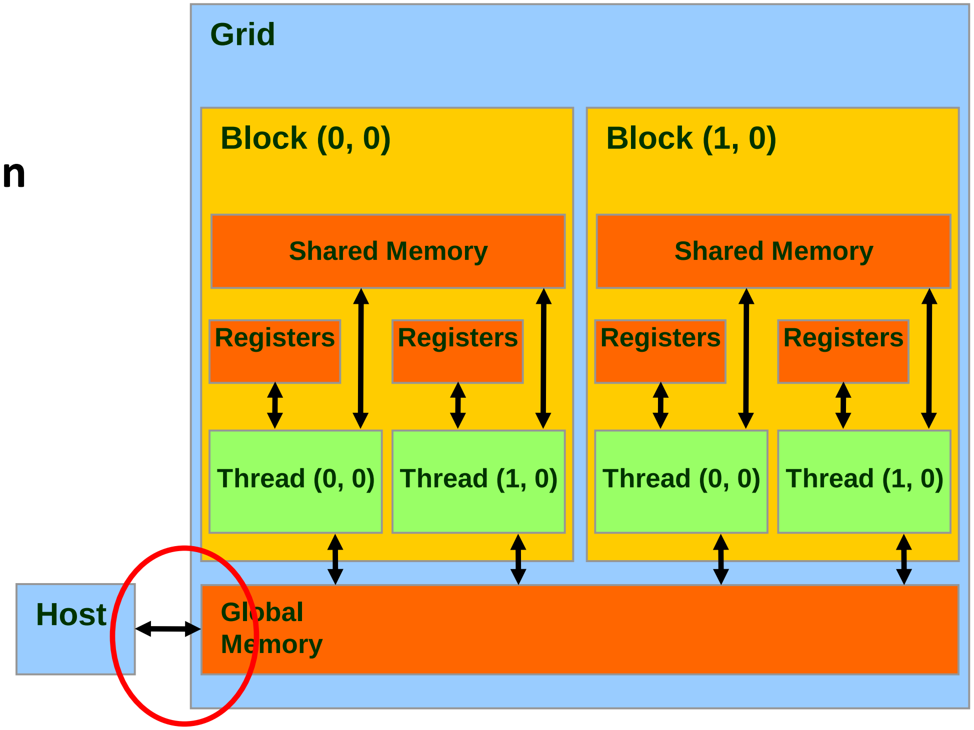
- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous transfer



CUDA Device Memory Allocation

Example:

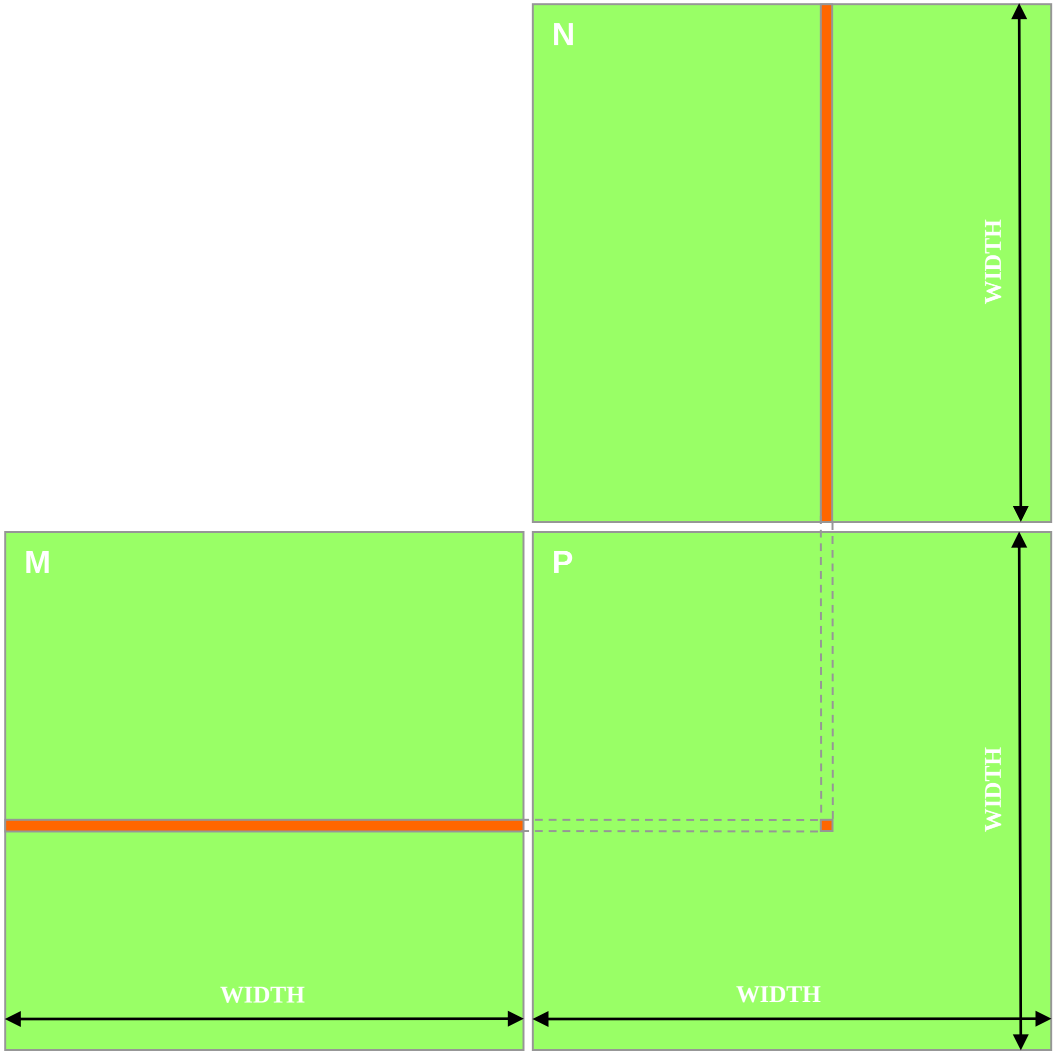
Destination pointer | Source pointer | Size in bytes | Direction




```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```

The *Hello World* of Parallel Programming: **Matrix Multiplication**

Data Parallelism:
We can safely perform many arithmetic operations on the data structures in a simultaneous manner.



The *Hello World* of Parallel Programming: **Matrix Multiplication**

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

M

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

C adopts raw-major placement approach
when storing 2D matrix in linear memory address.

The *Hello World* of Parallel Programming: **Matrix Multiplication**

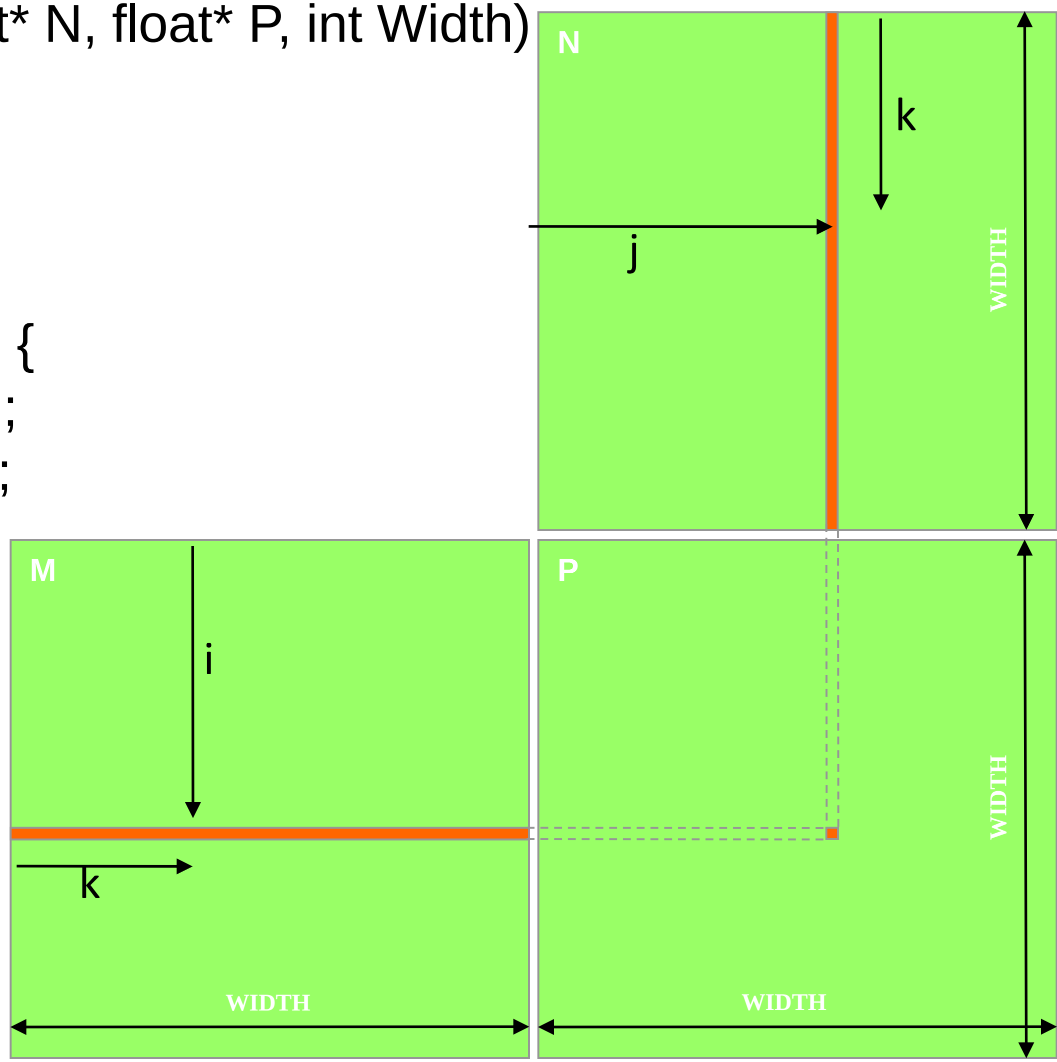
```
int main(void) {
1.   // Allocate and initialize the matrices M, N, P
     // I/O to read the input matrices M and N
....

2.   // M * N on the device
     MatrixMultiplication(M, N, P, Width);


3.   // I/O to write the output matrix P
     // Free matrices M, N, P
...
return 0;
}
```

A Simple main function: executed at the host

The *Hello World* of Parallel Programming: **Matrix Multiplication**

// Matrix multiplication on the (CPU) host

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



The *Hello World* of Parallel Programming: **Matrix Multiplication**

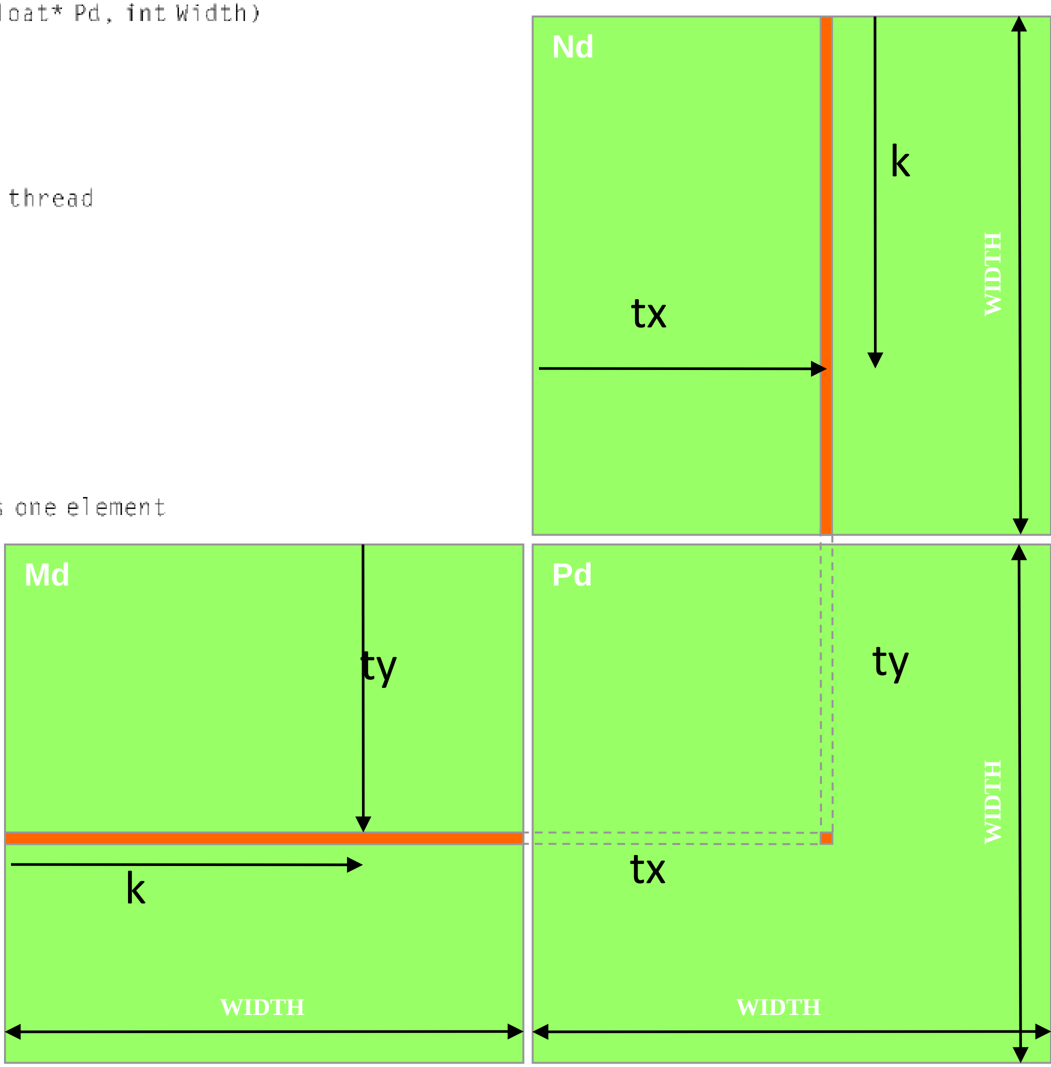
```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
1. // Allocate device memory for M, N, and P
   // copy M and N to allocated device memory locations

2. // Kernel invocation code - to have the device to perform
   // the actual matrix multiplication

3. // copy P from the device memory
   // Free device matrices
}
```

The *Hello World* of Parallel Programming: **Matrix Multiplication**

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;

1. // Transfer M and N to device memory
   cudaMalloc((void**) &Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
   cudaMalloc((void**) &Nd, size);
   cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

   // Allocate P on the device
   cudaMalloc((void**) &Pd, size);

2. // Kernel invocation code – to be shown later
   ...
3. // Transfer P from device to host
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
   // Free device matrices
   cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

The *Hello World* of Parallel Programming: **Matrix Multiplication**

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // Pvalue stores the Pd element that is computed by the thread
   float Pvalue = 0;

   for (int k = 0; k < Width; ++k)
   {
      float Mdelement = Md[ty * Width + k];
      float Ndelement = Nd[k * Width + tx];
      Pvalue += Mdelement * Ndelement;
   }

   // Write the matrix to device memory each thread writes one element
   Pd[ty * Width + tx] = Pvalue;
}
```



The Kernel Function

The *Hello World* of Parallel Programming: **Matrix Multiplication**

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No indirect function calls through pointers

Specifying Dimensions

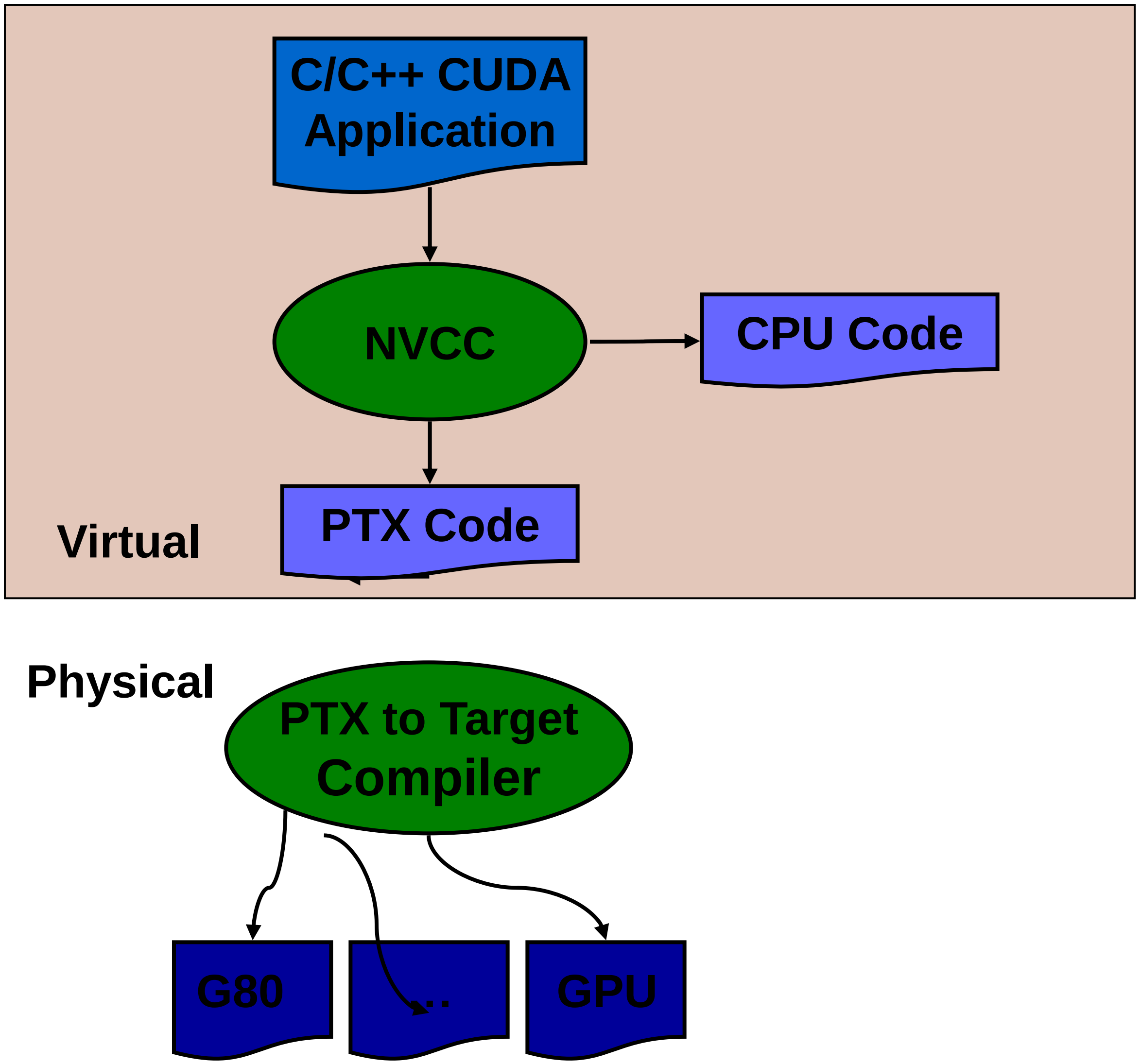
```
// Setup the execution configuration
   dim3 dimGrid(1, 1);
  dim3 dimBlock(Width, Width);
```

```
// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Important:

- dimGrid and dimBlock are user defined
- **gridDim** and **blockDim** are built-in predefined variable accessible in kernel functions

Tools


C/C++ CUDA Application
NVCC
CPU Code
PTX Code
Virtual
Physical
PTX to Target Compiler
G80
…
GPU

Conclusions

- We are done with chp 3 of the book.
- We looked at out first CUDA program
- What we learned today about CUDA:
  - KernelA<<< nBlk, nTid >>>(args)
  - cudaMalloc()
  - cudaFree()
  - cudaMemcpy()
  - gridDim and blockDim
  - threadIdx.x and threadIdx.y
  - dim3