ECE 498AL

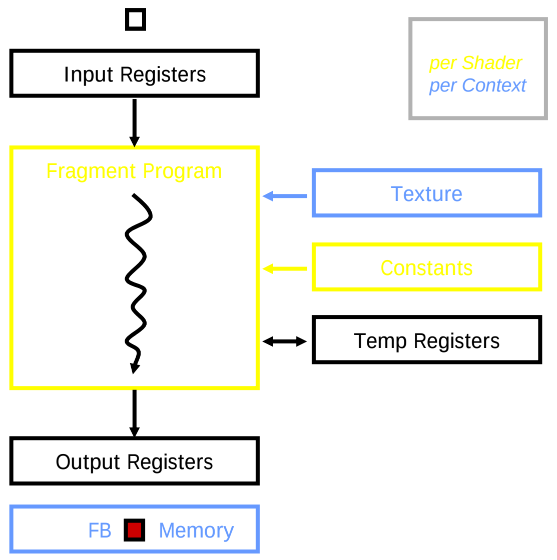
Lecture 2: The CUDA Programming Model

What is (Historical) GPGPU ?

- General Purpose computation using GPU and graphics API in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

Previous GPGPU Constraints

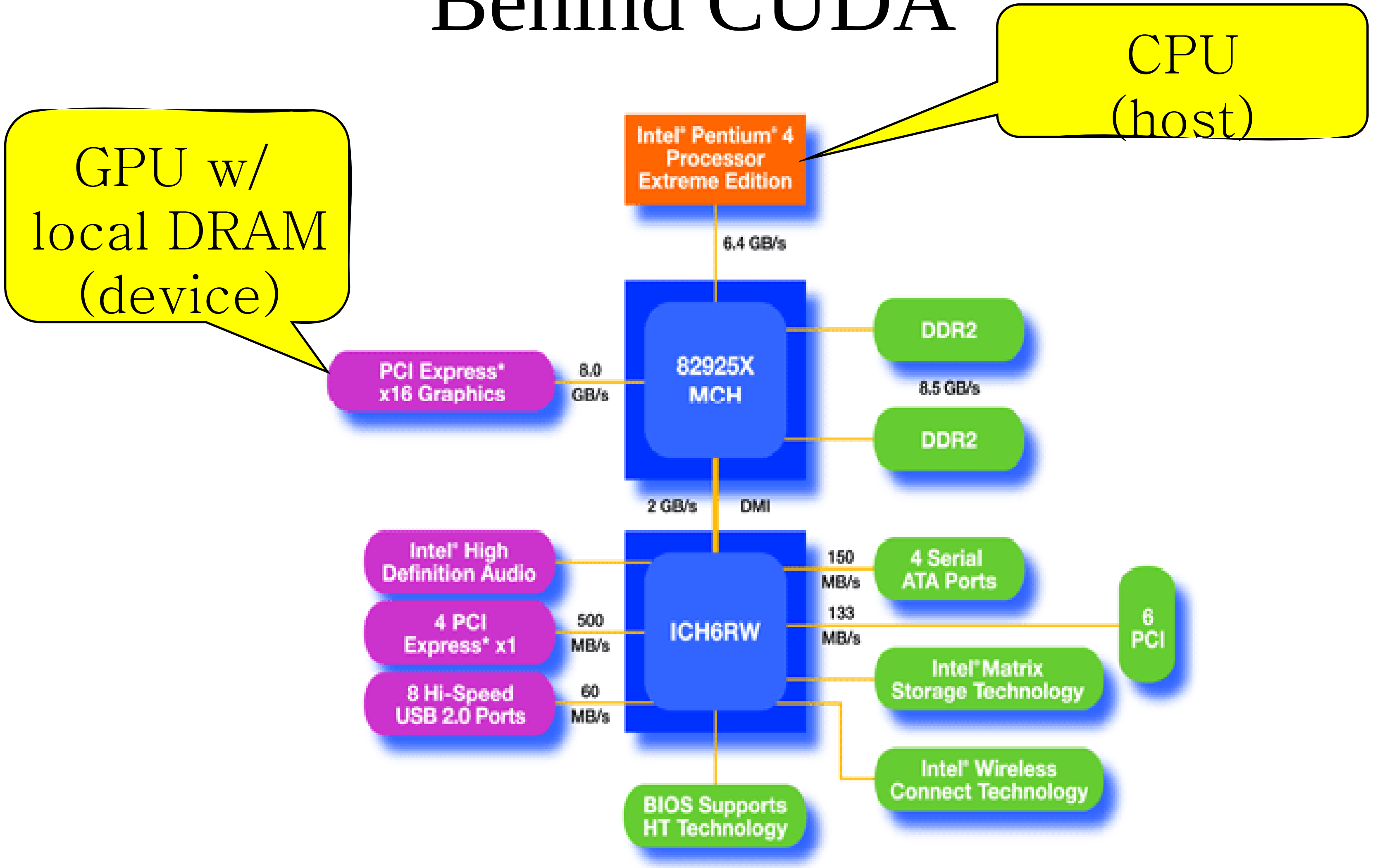
- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Lack of Integer & bit ops
- Communication limited
  - Between pixels
  - Scatter a[i] = p



CUDA

- "Compute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute – graphics-free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

An Example of Physical Reality Behind CUDA


CPU (host)
GPU w/ local DRAM (device)
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

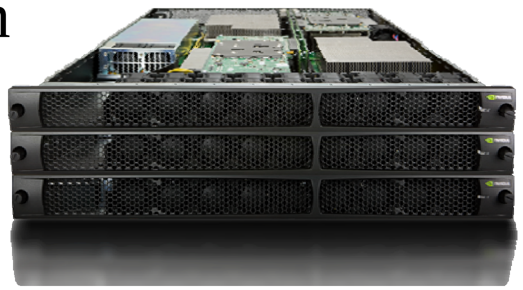
- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters

- 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

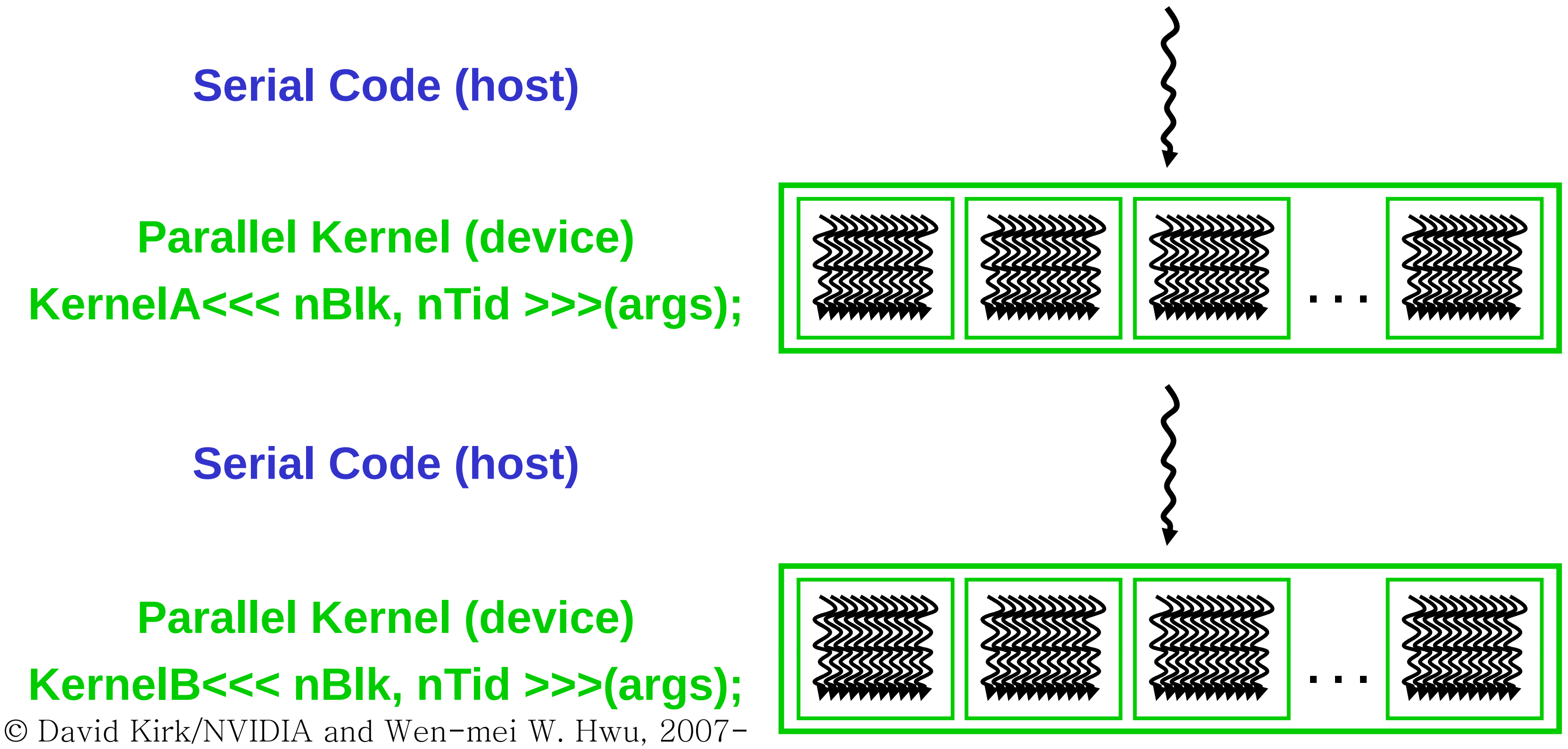

Tesla S870

Overview

- CUDA programming model – basic concepts and data types
- CUDA application programming interface - basic
- Simple examples to illustrate basic concepts and functionalities
- Performance features will be covered later

CUDA – C with no shader limitations!

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

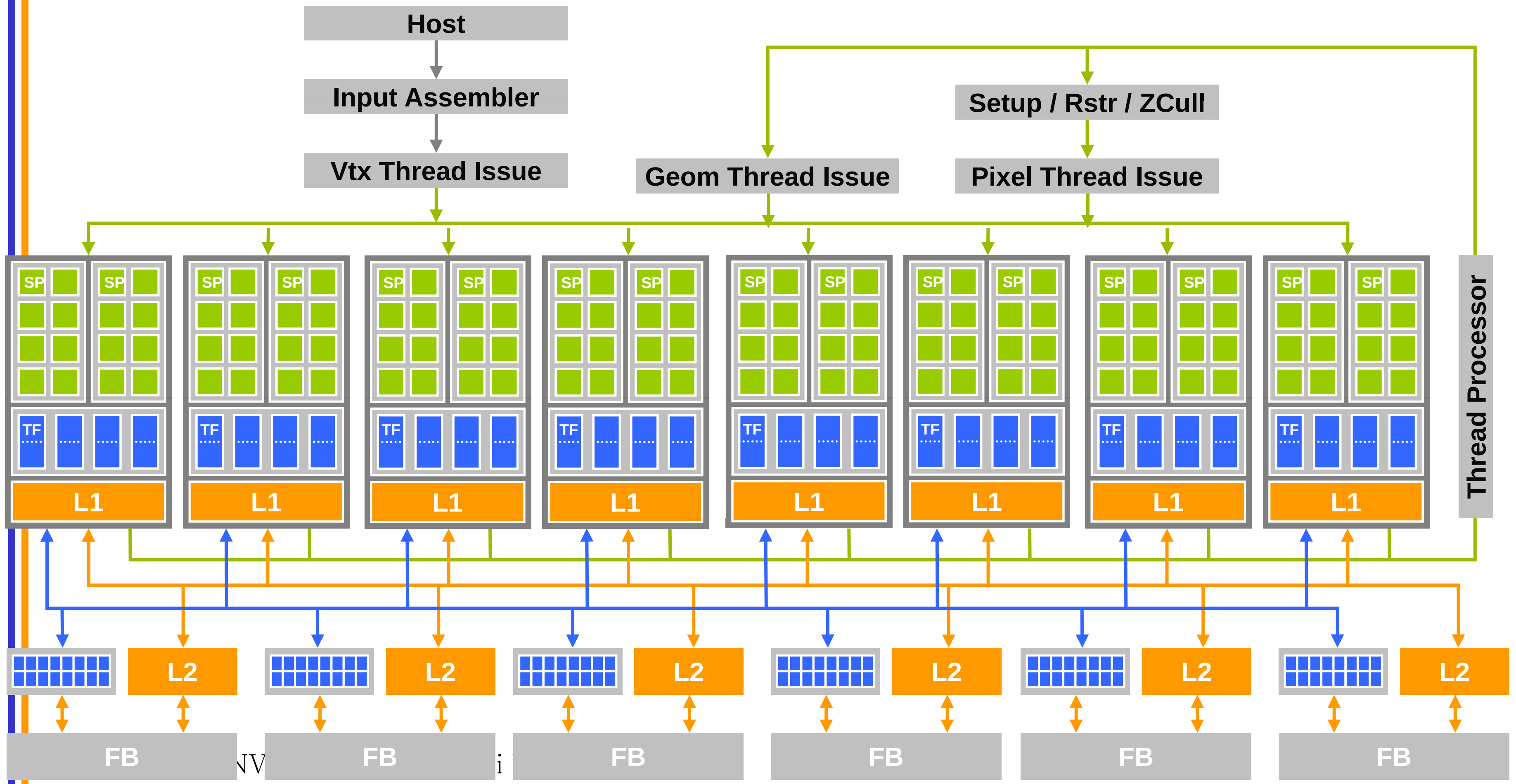


CUDA Devices and Threads

- A compute device
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device kernels which run on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

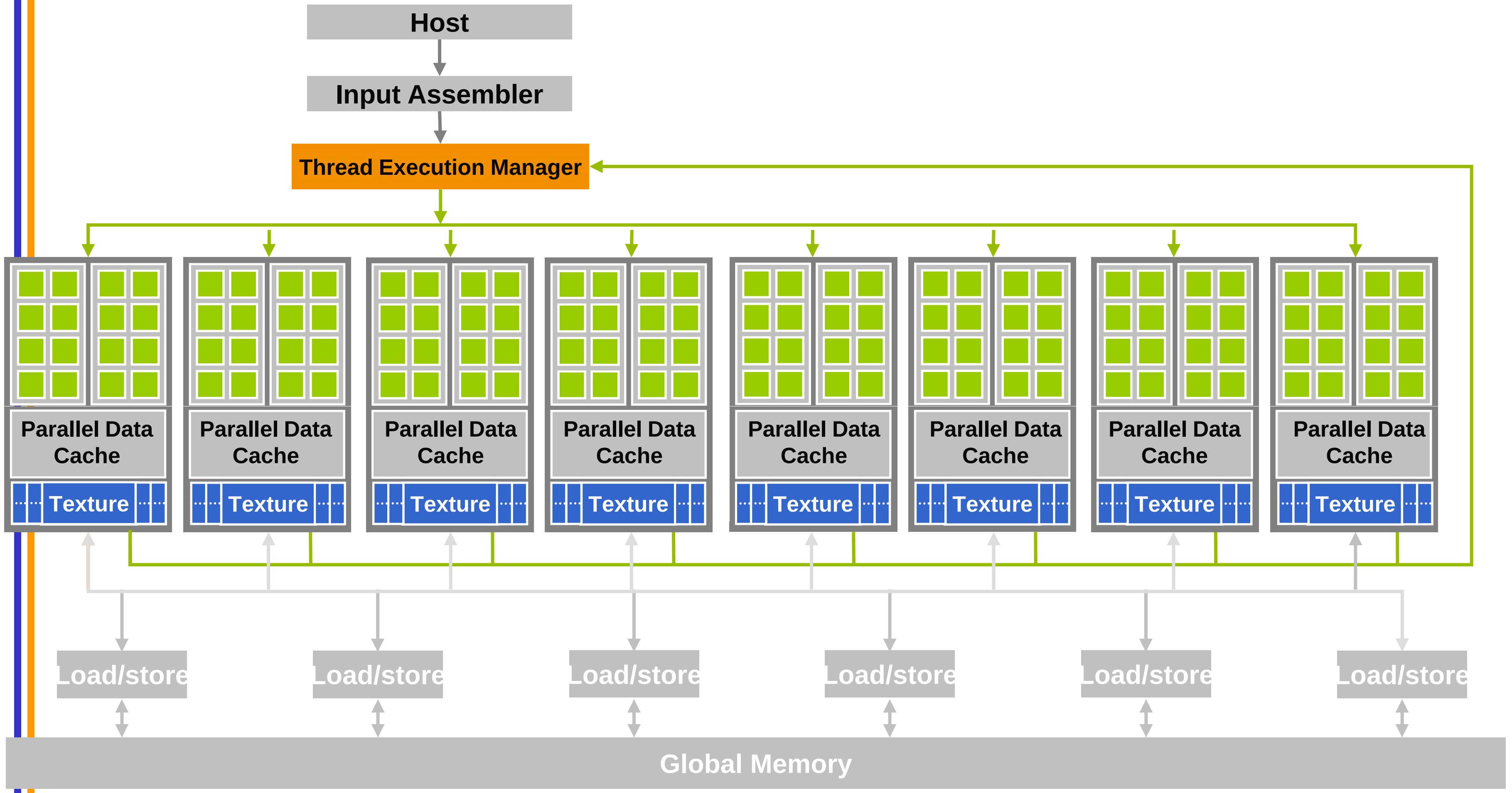
G80 – Graphics Mode

- The future of GPUs is programmable processing
- So – build the architecture around the processor



G80 CUDA mode – A **Device** Example

- Processors execute computing threads
- New operating mode/HW interface for computing

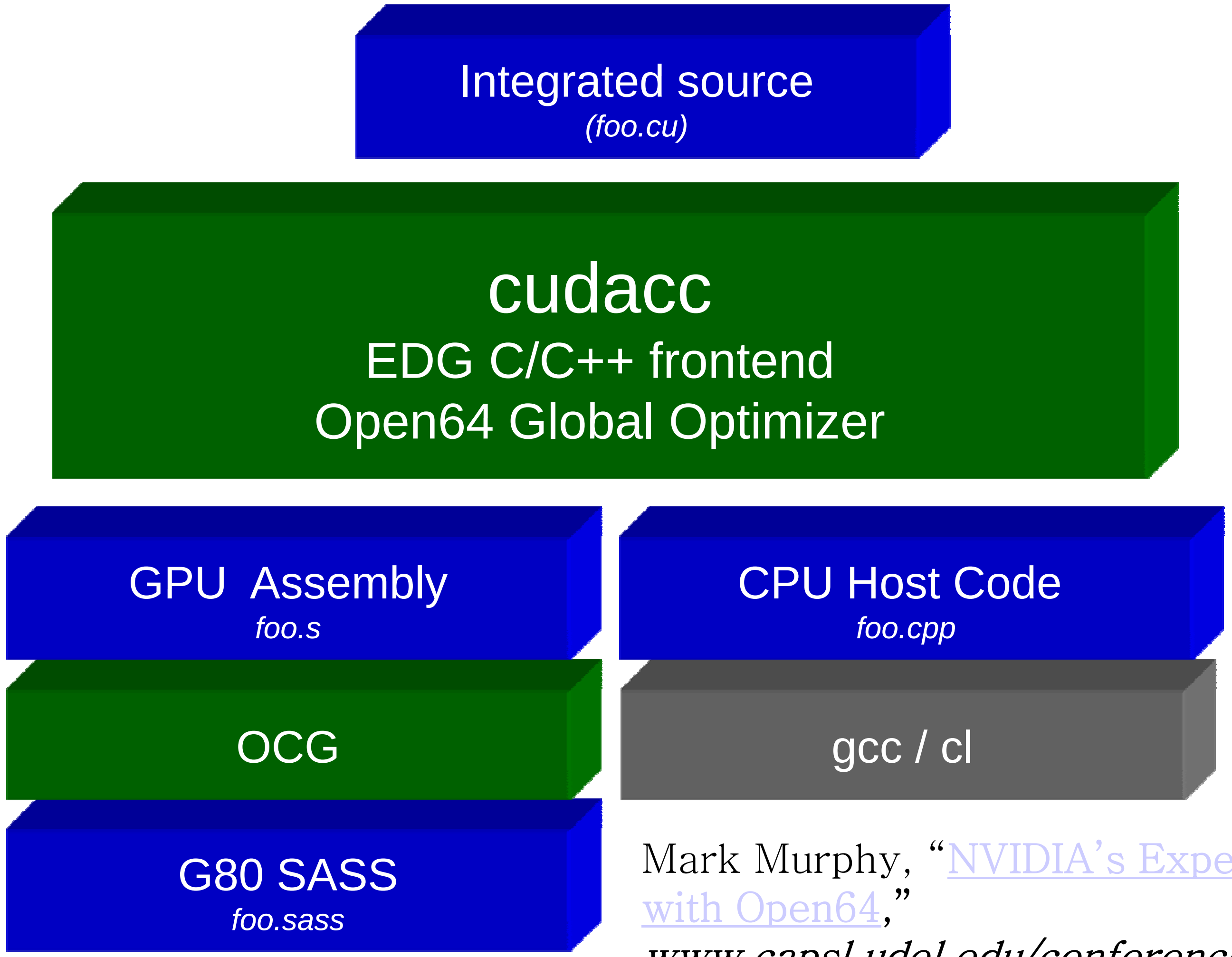


Extended C

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

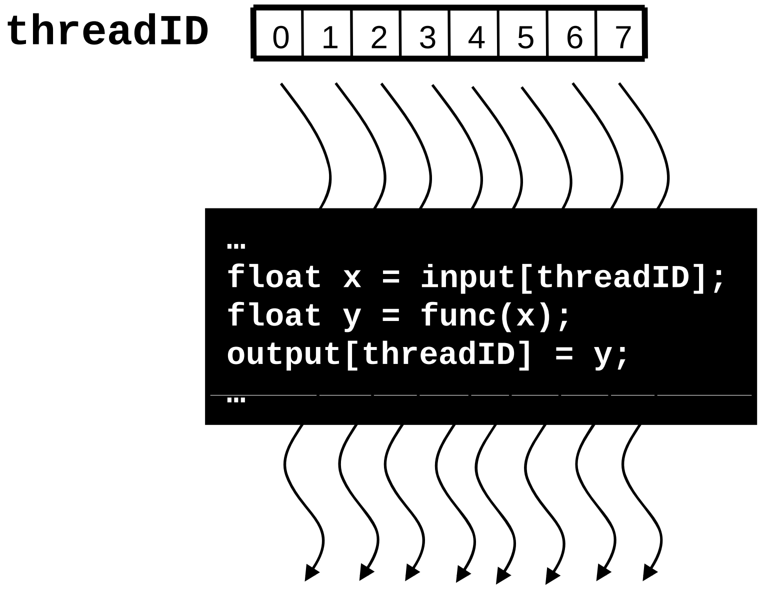
Extended C



Mark Murphy, "NVIDIA's Experience with Open64," *www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc*

Arrays of Parallel Threads

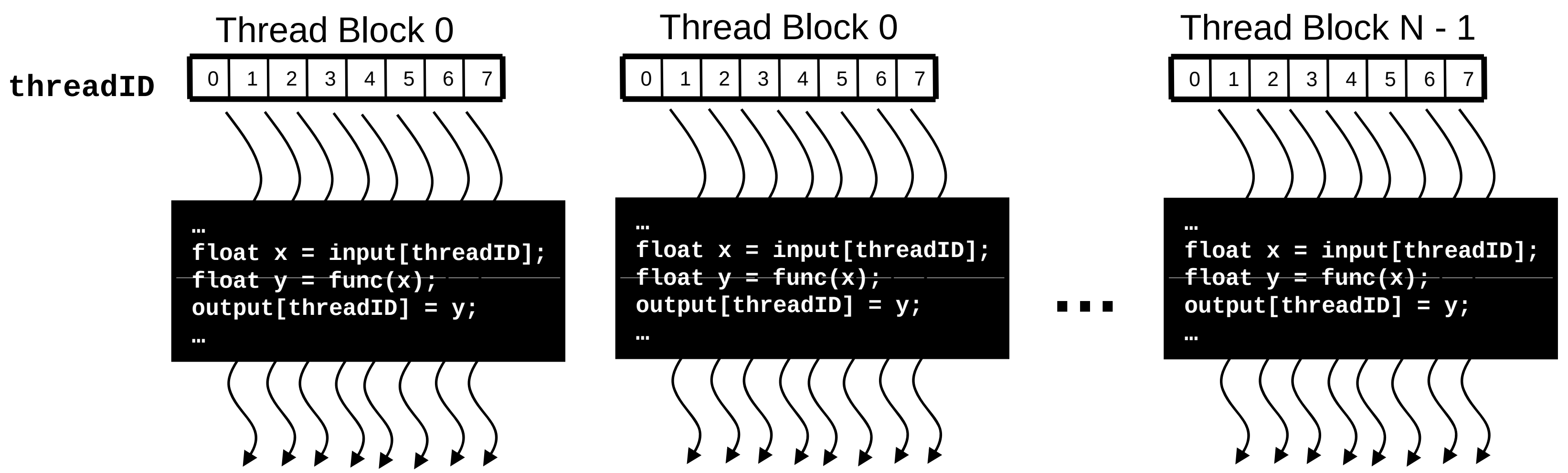
- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

Thread Blocks: Scalable Cooperation

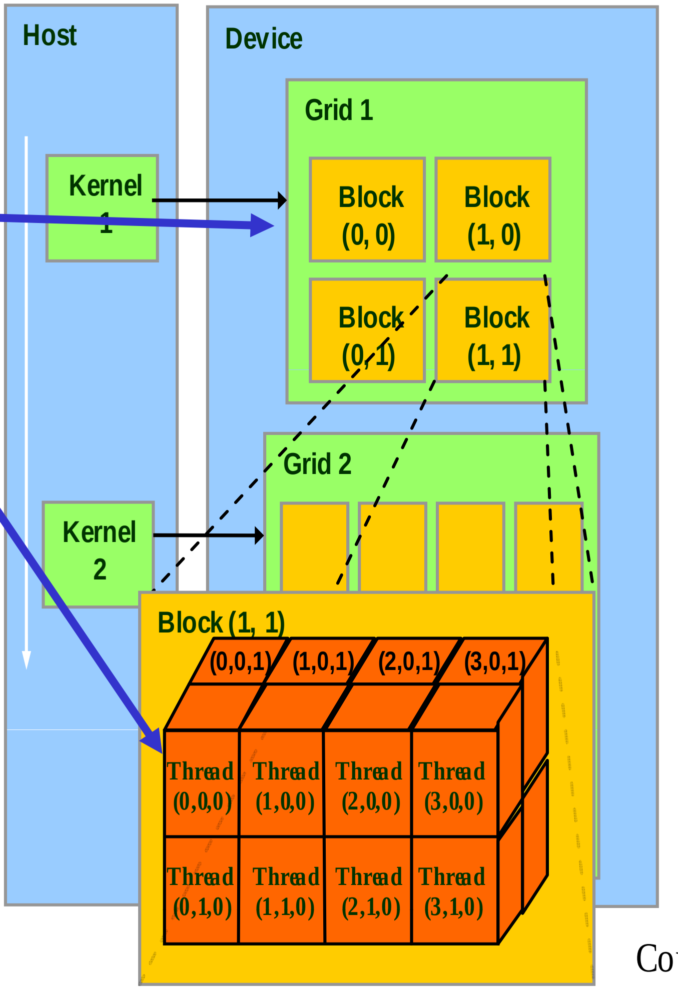
- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate



```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

Block IDs and Thread IDs

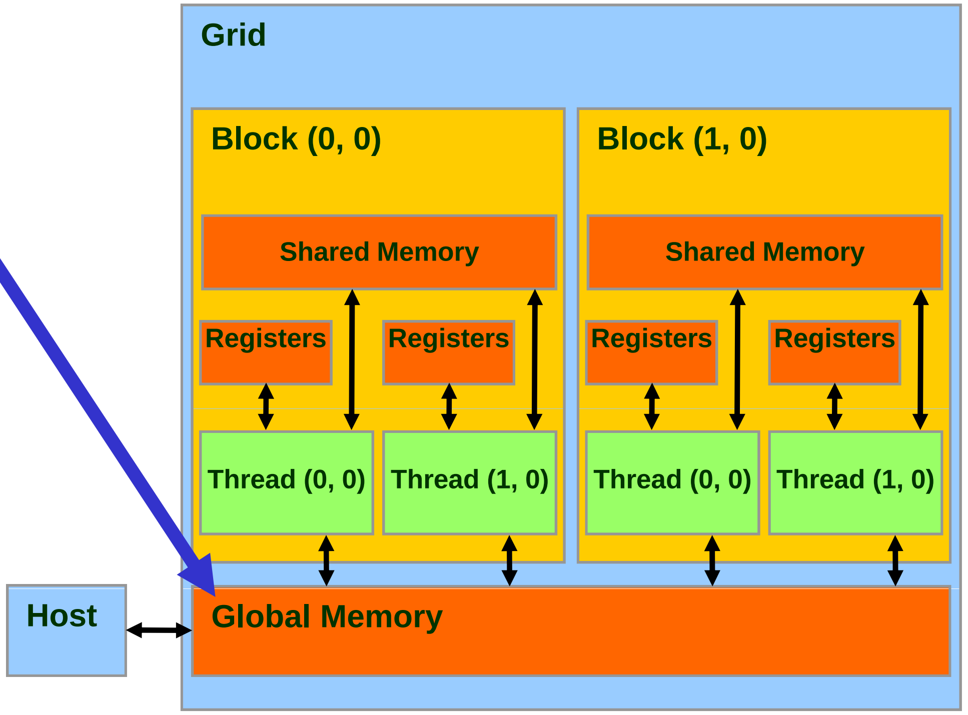
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

CUDA Memory Model Overview

- 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 API Highlights: Easy and Lightweight

- The API is an extension to the ANSI C programming language

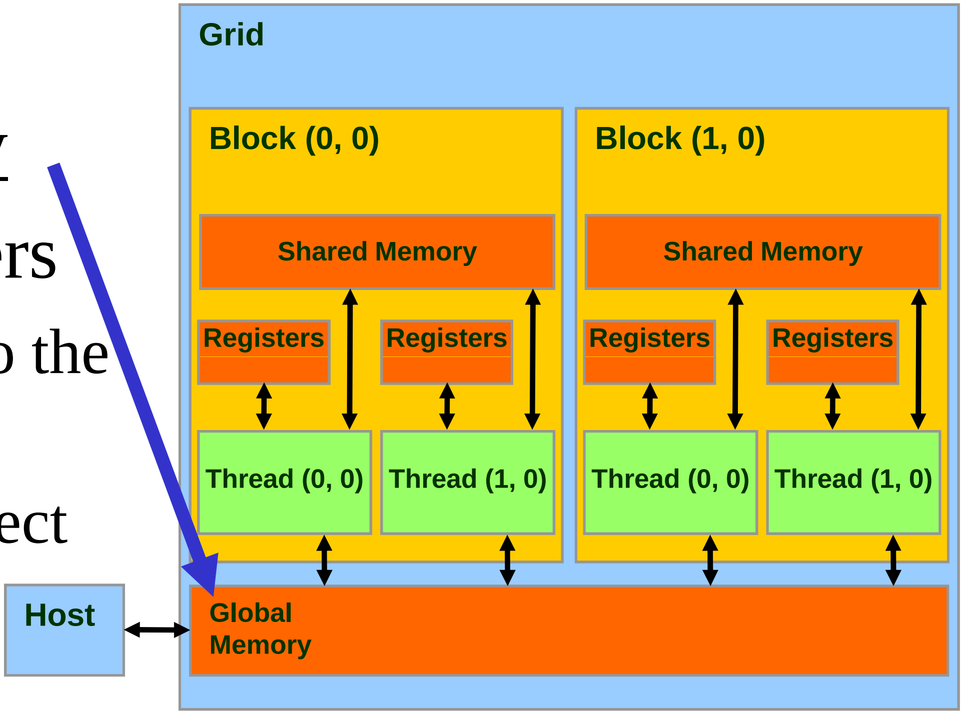
  $\longrightarrow$ Low learning curve

- The hardware is designed to enable lightweight runtime and driver

  $\longrightarrow$ High performance

CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



CUDA Device Memory Allocation (cont.)

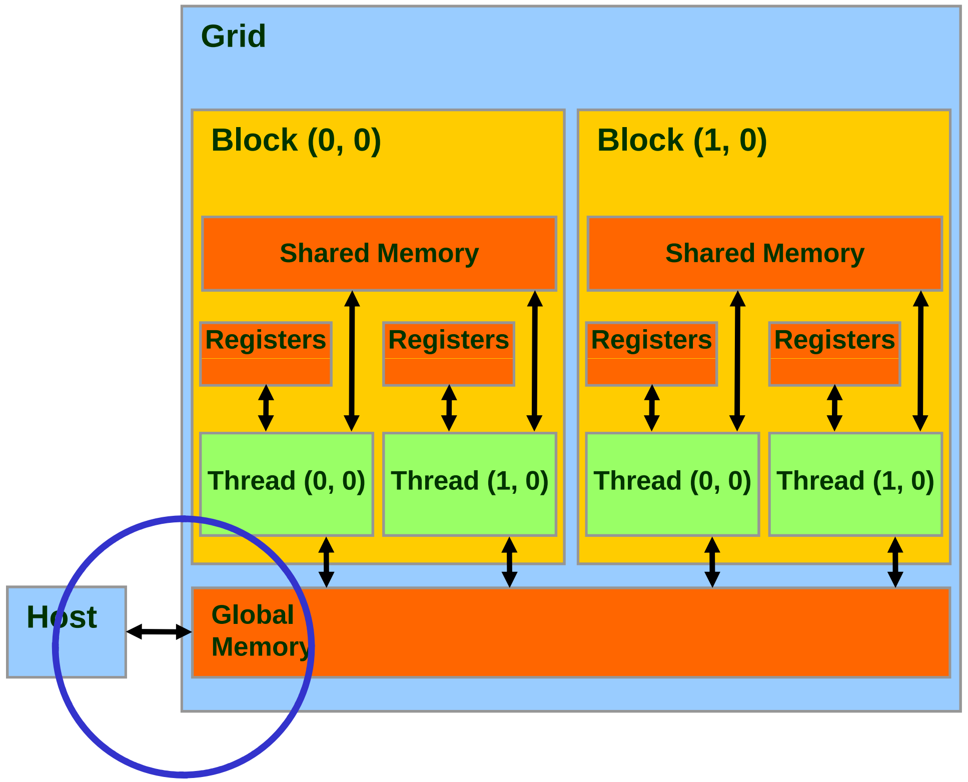
- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md
  - “d” is often used to indicate a device data structure

```
TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);
```

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

CUDA Host-Device Data Transfer

- 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 Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

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

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

CUDA Keywords

CUDA Function Declarations

	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

CUDA Function Declarations (cont.)

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);     // 5000 thread blocks
dim3   DimBlock(4, 8, 8);    // 256 threads per
   block
size_t SharedMemBytes = 64; // 64 bytes of shared
   memory
KernelFunc<<< DimGrid, DimBlock,
   SharedMemBytes >>>(...);
```

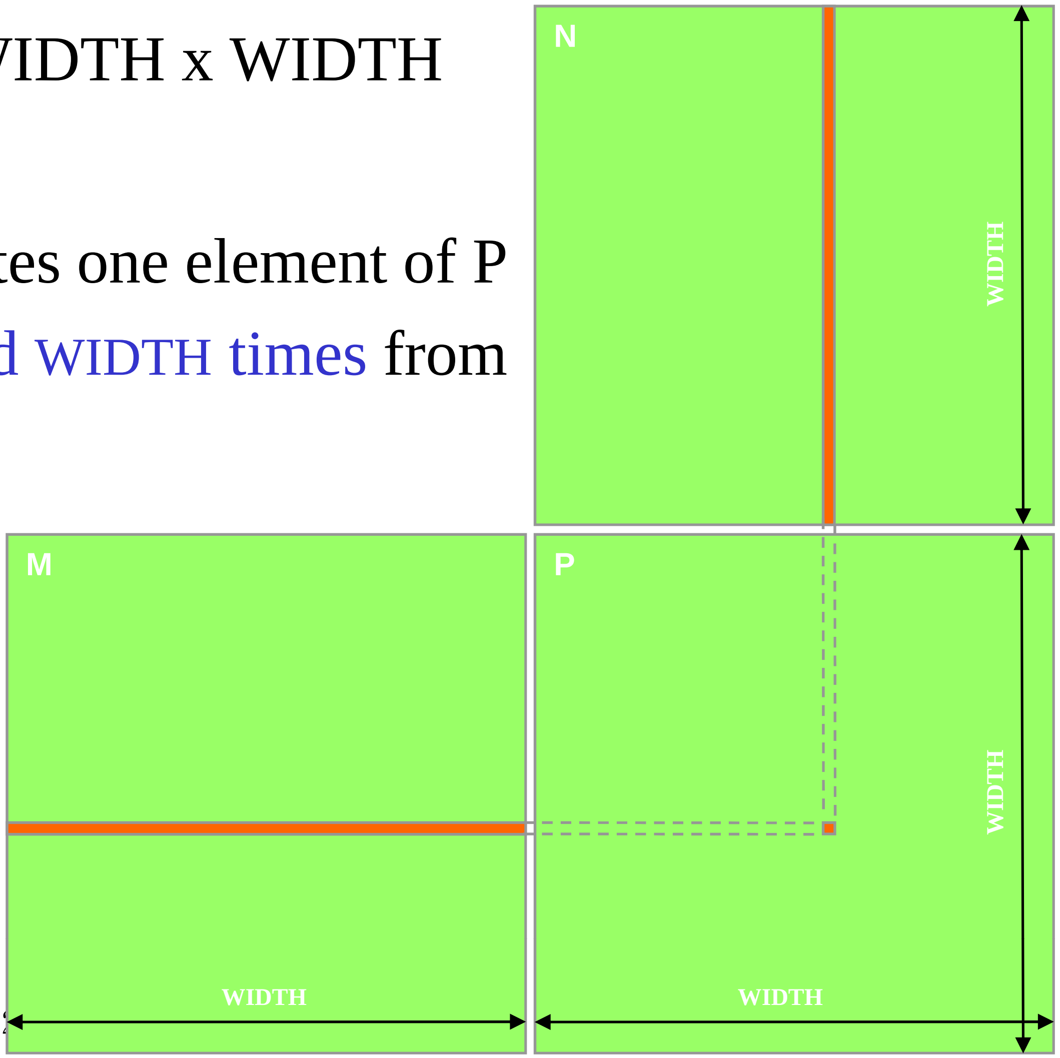
- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

A Simple Running Example Matrix Multiplication

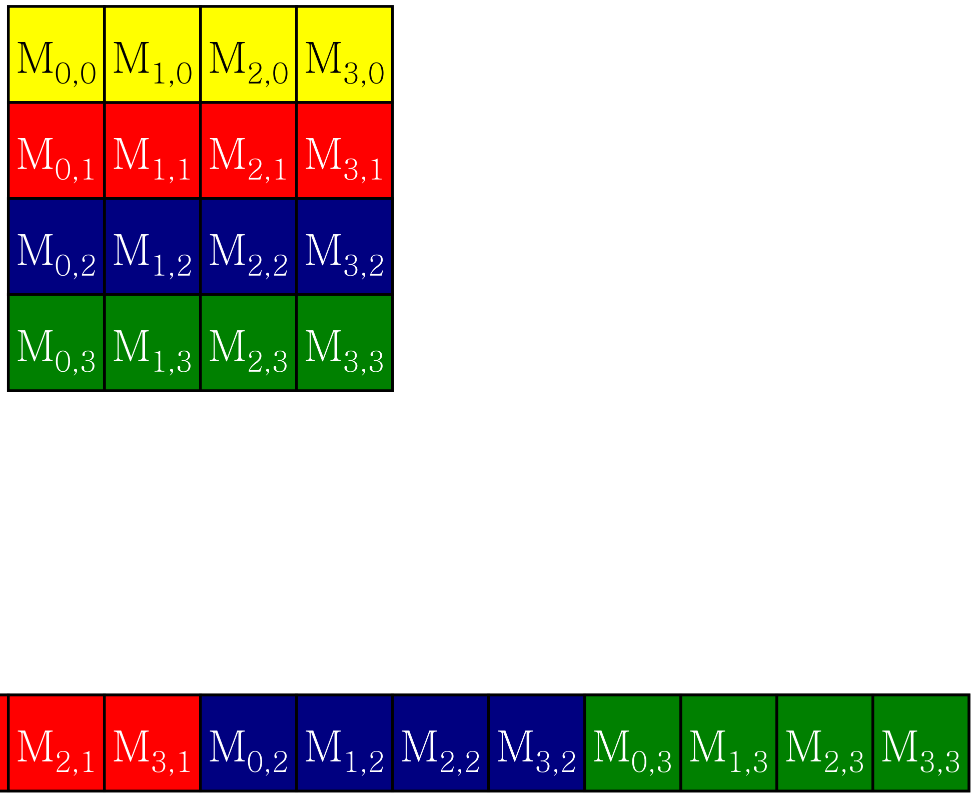
- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity

Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory

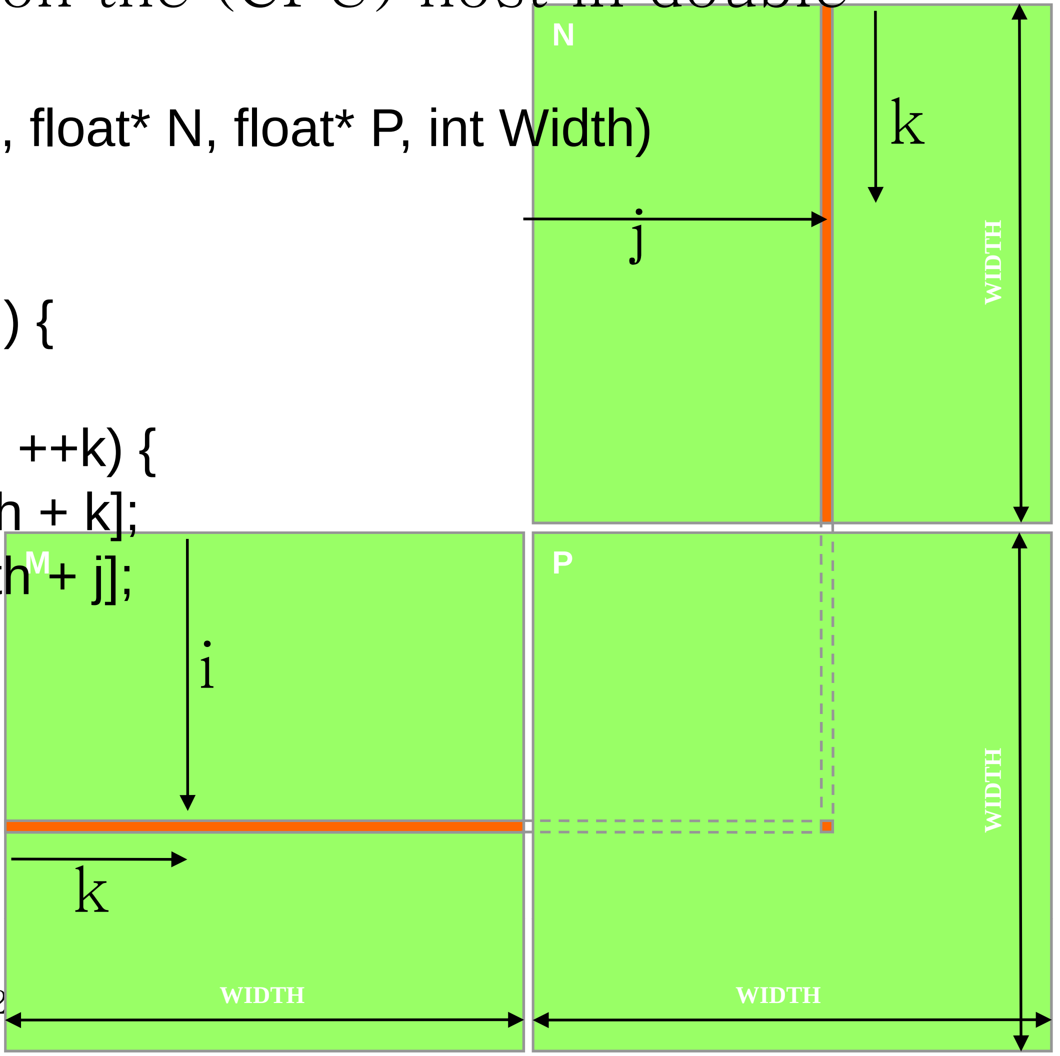


Memory Layout of a Matrix in C


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

Step 1: Matrix Multiplication A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host in double
precision
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;
        }
}
```



Step 2: Input Matrix Data Transfer (Host-side Code)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;
   …
1. // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

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

Step 3: Output Matrix Data Transfer (Host-side Code)

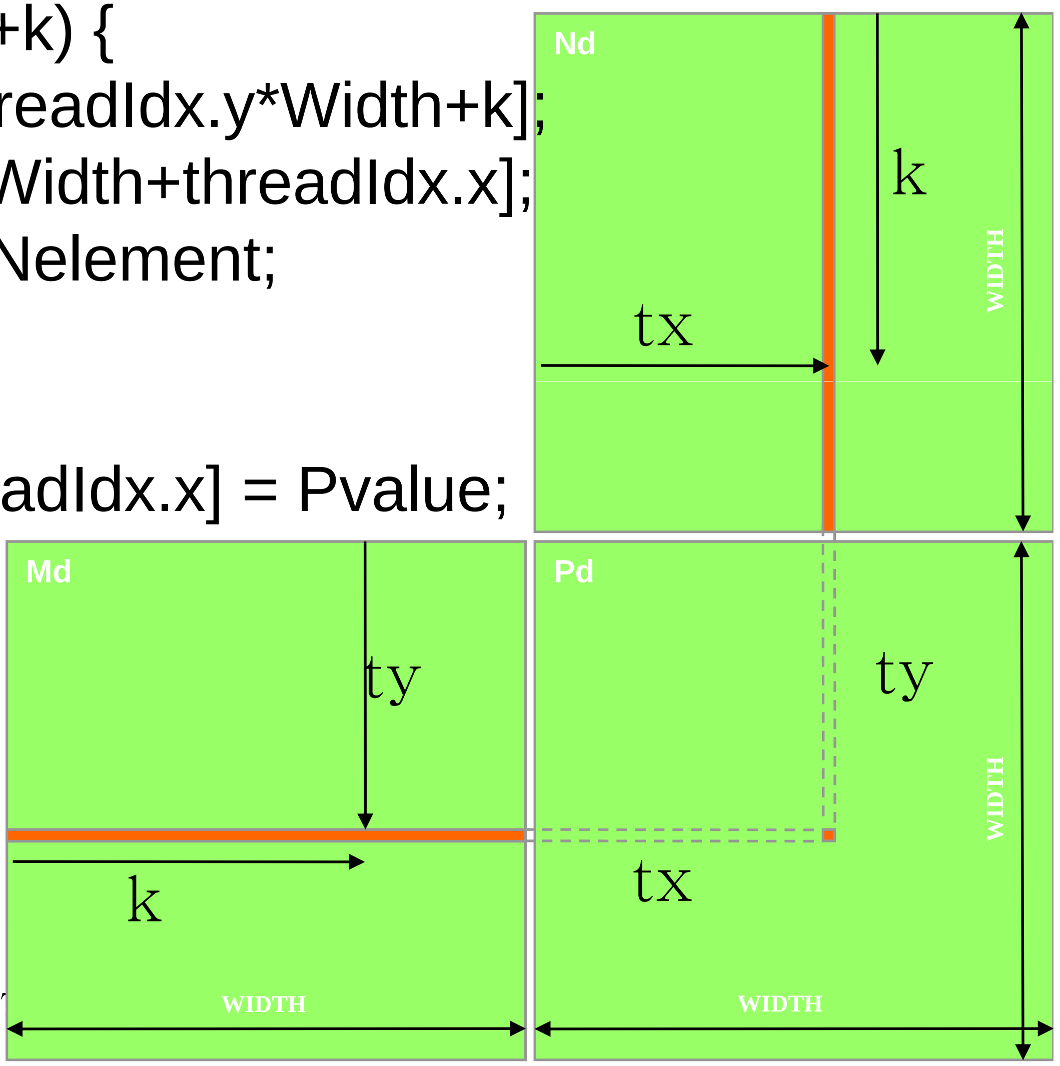
```
2.   // Kernel invocation code – to be shown later
     …

3.    // Read P from the device
      cudaMemcpy(P, Pd, size,
cudaMemcpyDeviceToHost);

       // Free device matrices
      cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
     }
```

Step 4: Kernel Function

```
// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Step 4: Kernel Function (cont.)

```
 for (int k = 0; k < Width; ++k) {
     float Melement = Md[threadIdx.y*Width+k];
     float Nelement = Nd[k*Width+threadIdx.x];
     Pvalue += Melement * Nelement;
 }

Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```

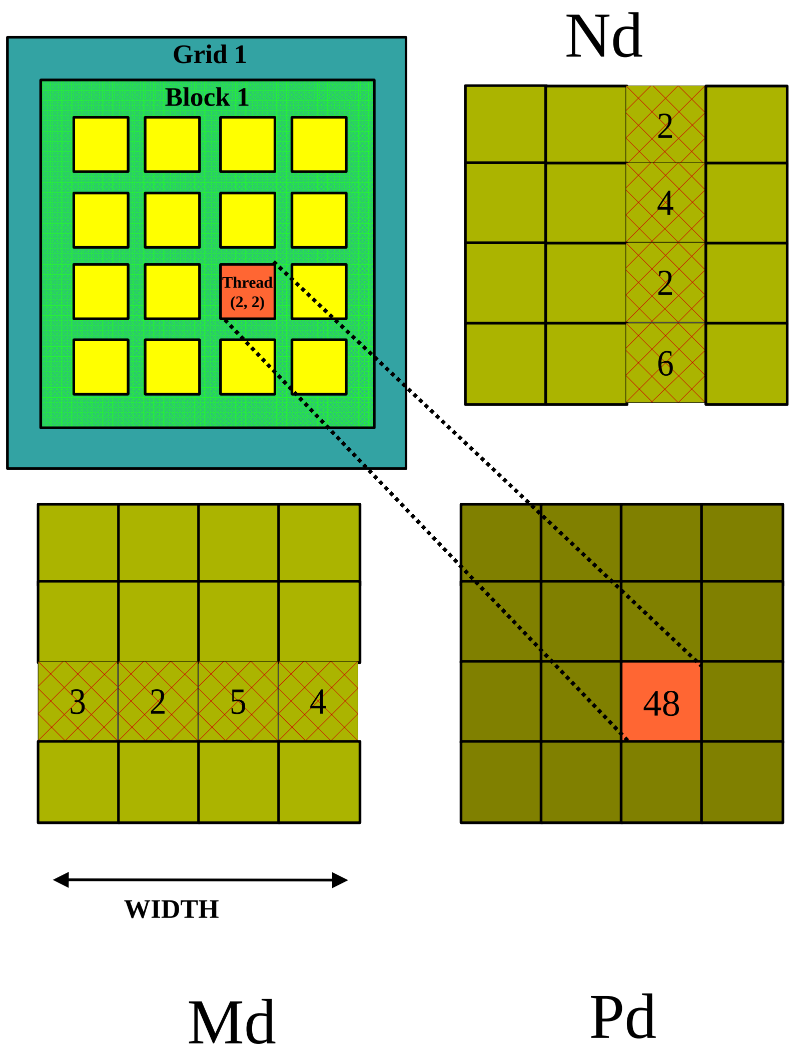


Step 5: Kernel Invocation (Host-side Code)

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


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

Only One Thread Block Used

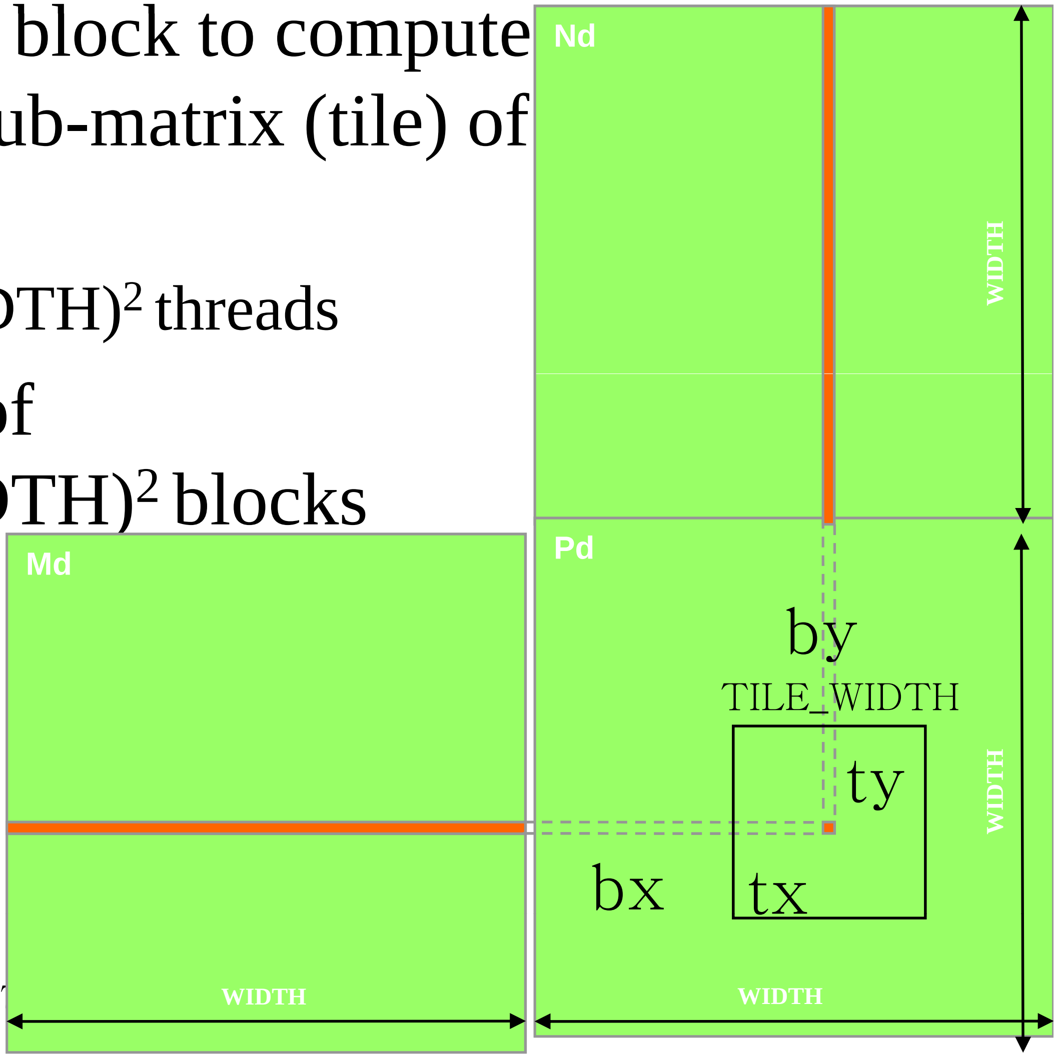
- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



Step 7: Handling Arbitrary Sized Square Matrices

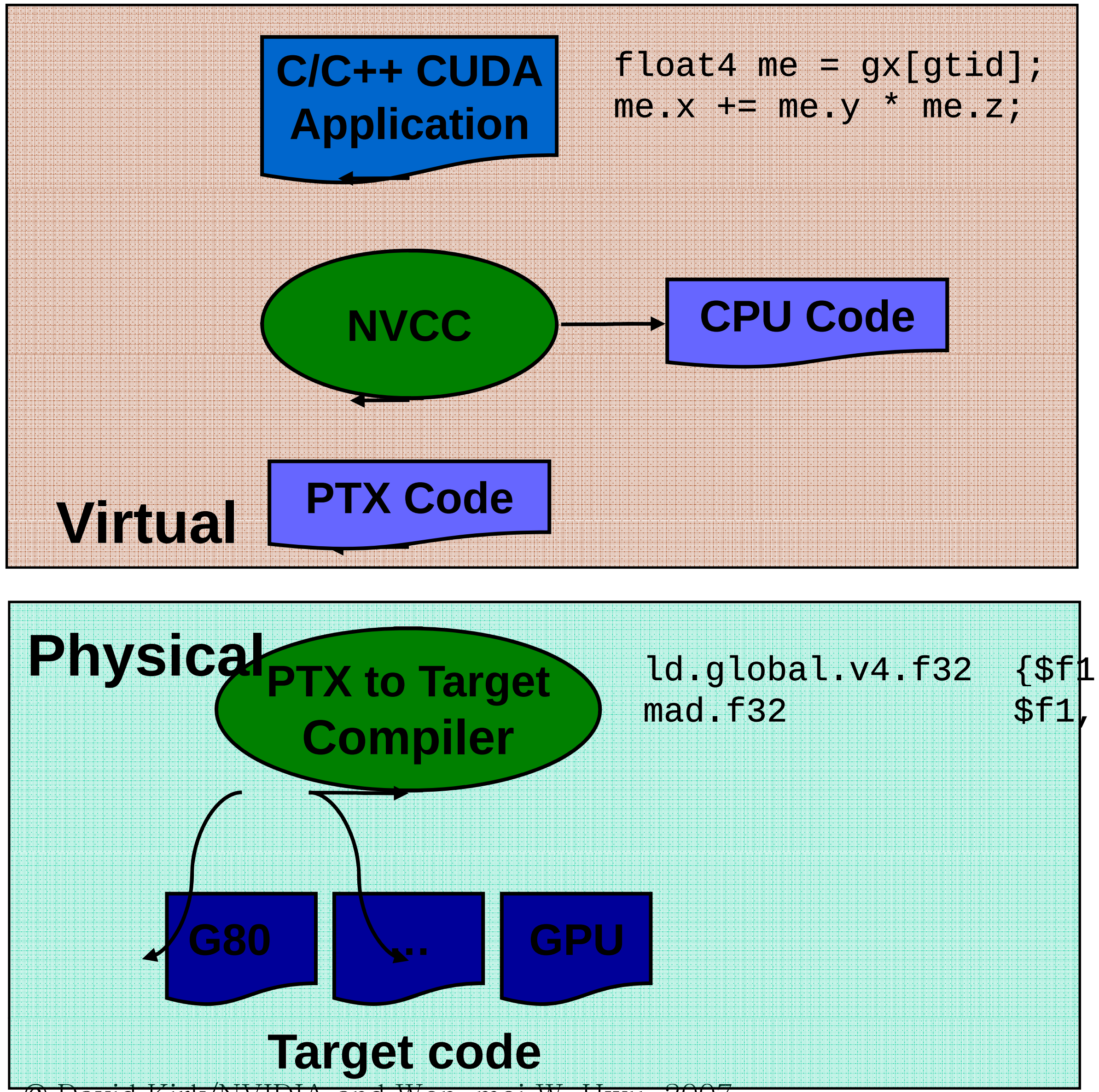
- Have each 2D thread block to compute a (TILE_WIDTH)^2 sub-matrix (tile) of the result matrix
  - Each has (TILE_WIDTH)^2 threads
- Generate a 2D Grid of (WIDTH/TILE_WIDTH)^2 blocks

You still need to put a loop around the kernel call for cases where WIDTH/TILE_WIDTH is greater than max grid size (64K)!



Some Useful Information on Tools

Compiling a CUDA Program



```
float4 me = gx[gtid];
me.x += me.y * me.z;
```

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```

- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

39

Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime

Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (**`cudart`**)
  - The CUDA core library (**`cuda`**)

Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (**`nvcc -deviceemu`**) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

- Running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. **`printf`**) and vice-versa
  - Detect deadlock situations caused by improper usage of **`__syncthreads`**

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode

Floating Point

- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host