

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

PUMPS Summer School

Programming and tUning Massively Parallel Systems

Lecture 2

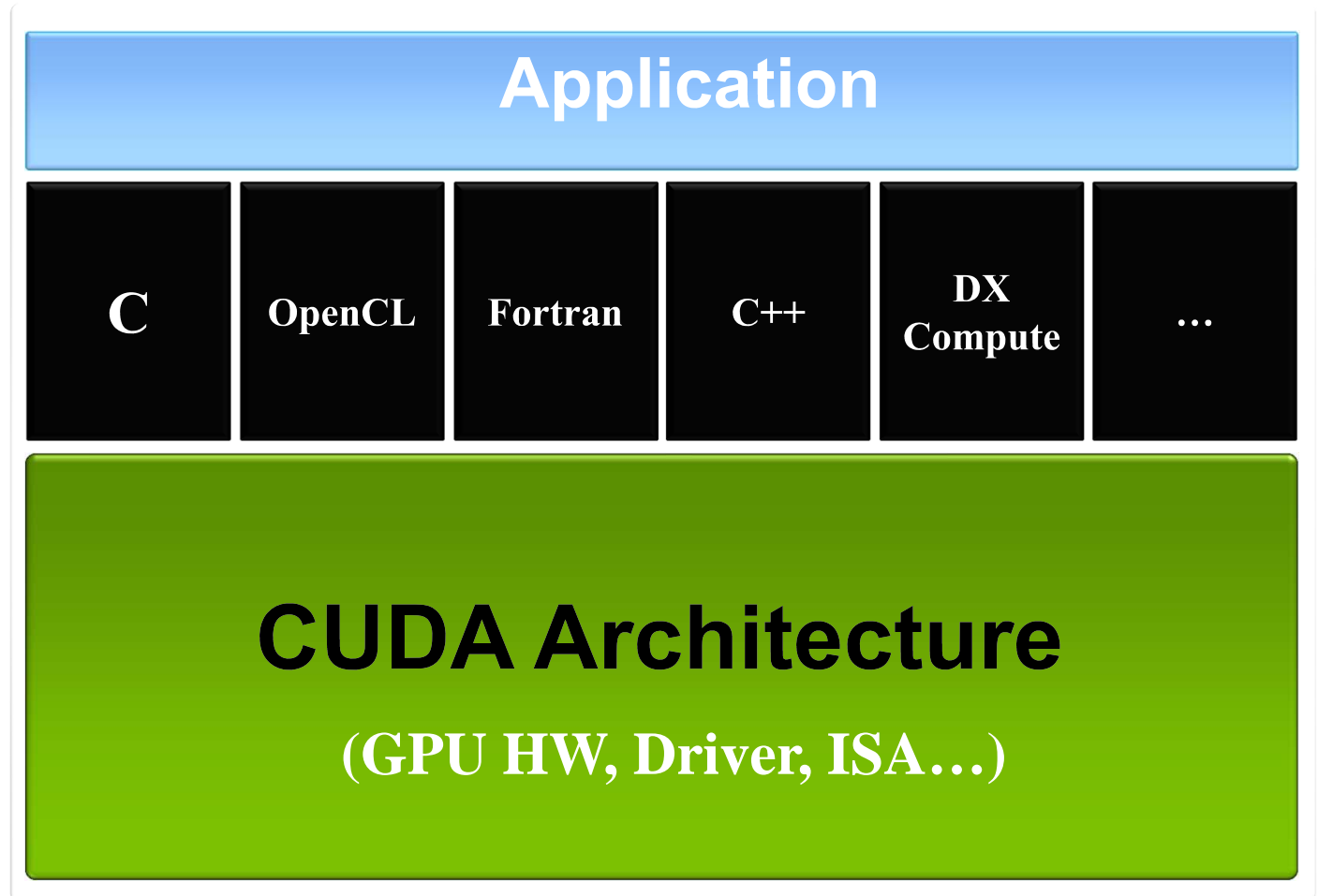
Introduction to CUDA

Overview

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

Many Language/API Choices

C/C++
OpenCL
DirectX Compute
Fortran
Java
Python
.Net
...



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

Serial Code (host)

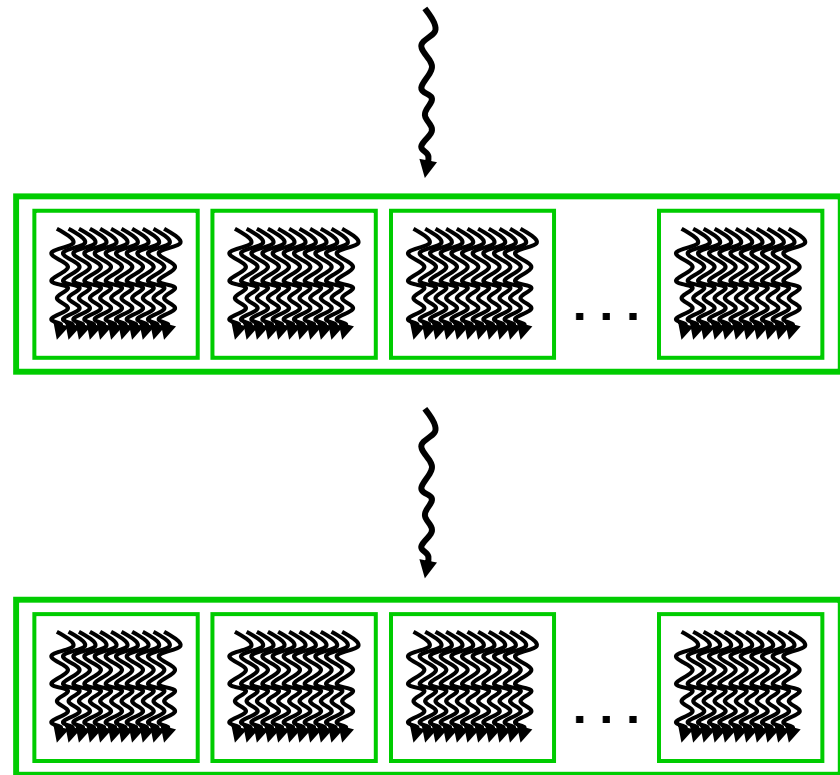
Parallel Kernel (device)

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

Serial Code (host)

Parallel Kernel (device)

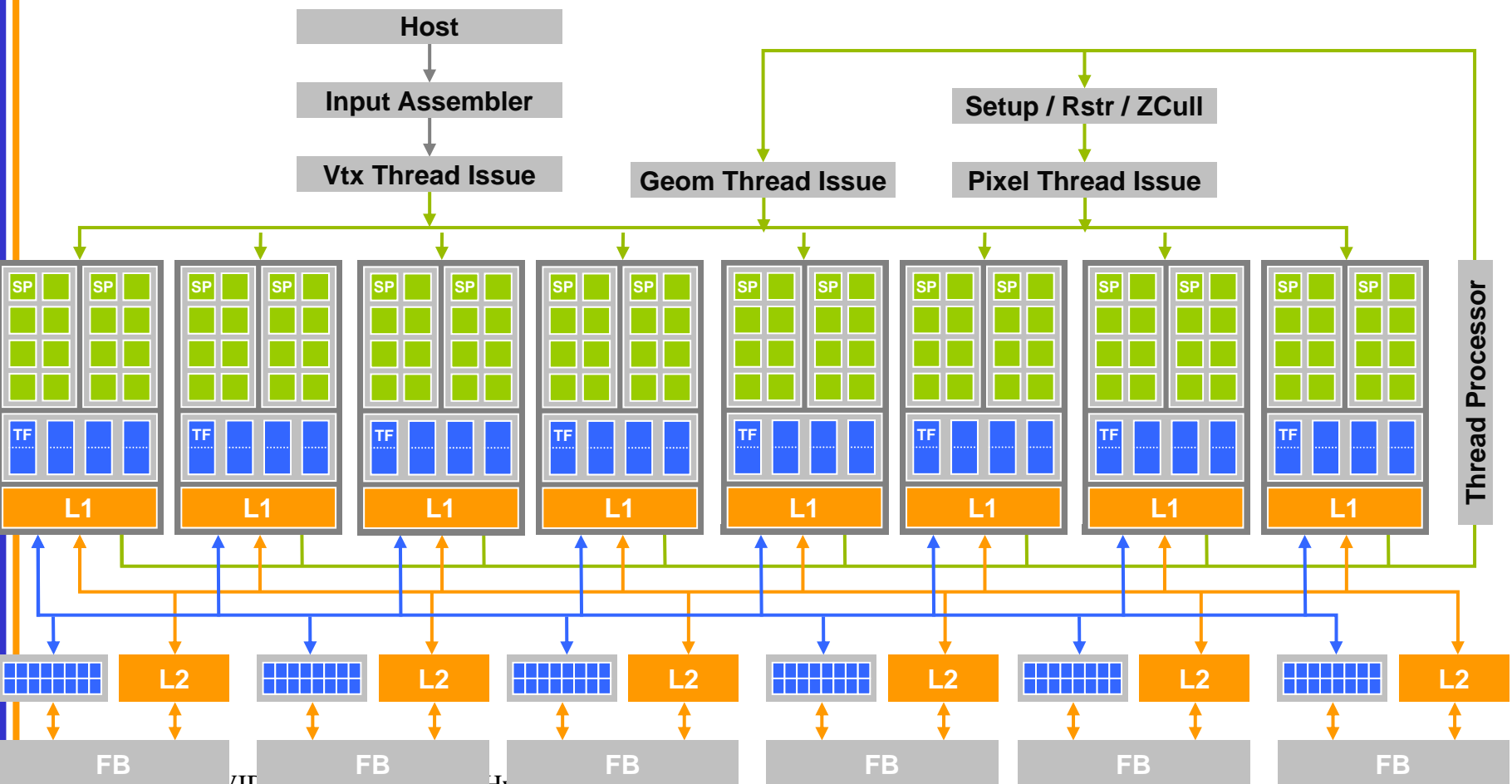
`KernelB<<< nBlk, nTid >>>(args);`



CUDA Devices and Threads

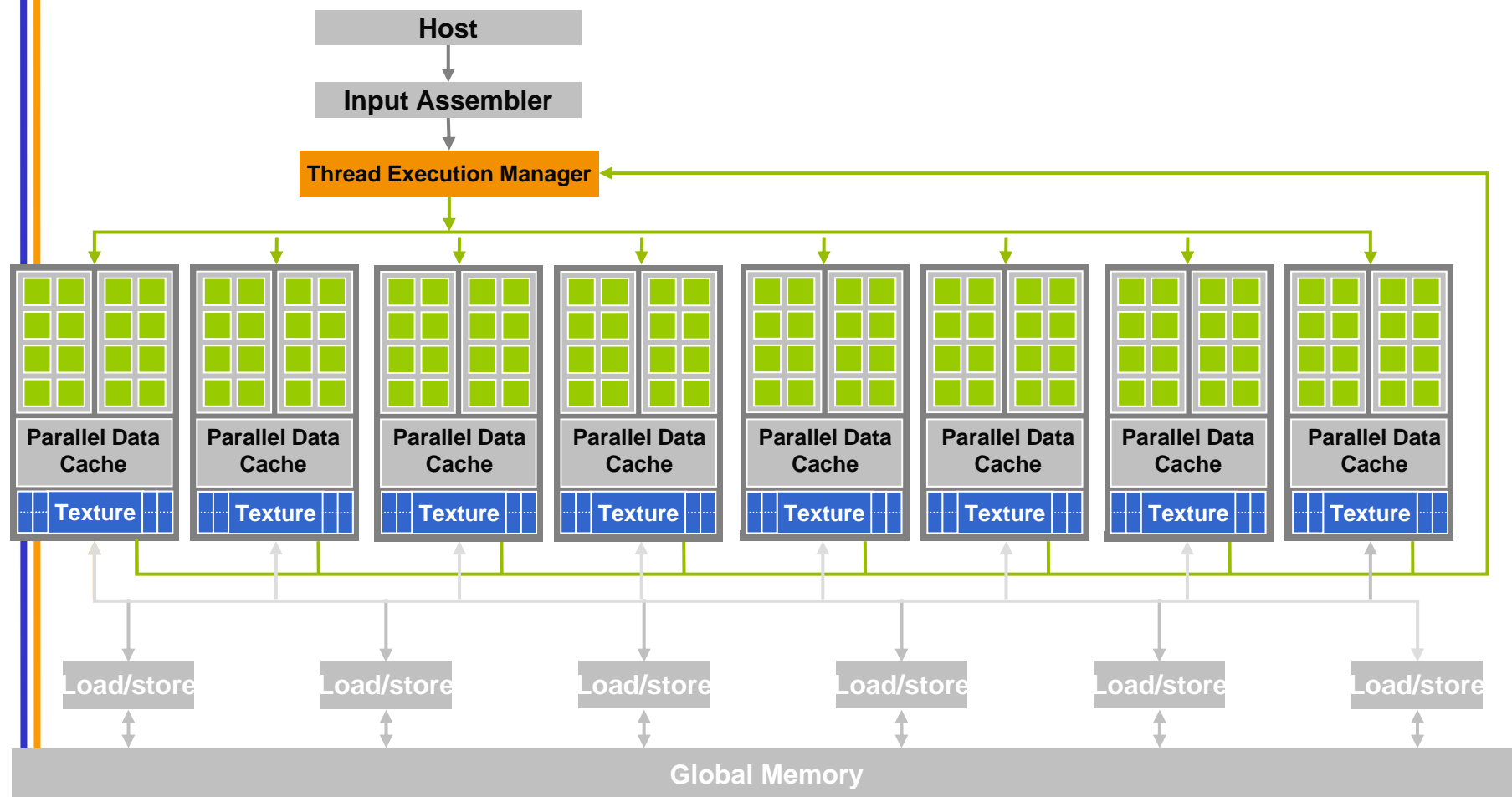
- A compute **device**
 - Is a coprocessor to the CPU or **host**
 - Has its own DRAM (**device memory**)
 - Runs many **threads in parallel**
 - Is typically a **GPU** but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device **kernels** 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

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



CUDA mode – A **Device** Example

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



CUDA C - extensions

- **Declspecs**

- **global, device, shared, constant**

```
__device__ float filter[N];

__global__ void convolve (float *image) {

    __shared__ float region[M];
    ...

    region[threadIdx] = image[i];

    __syncthreads()
    ...

    image[j] = result;
}
```

- **Keywords**

- **threadIdx, blockIdx**

- **Intrinsics**

- **__syncthreads**

- **Runtime API**

- **Memory, symbol, execution management**

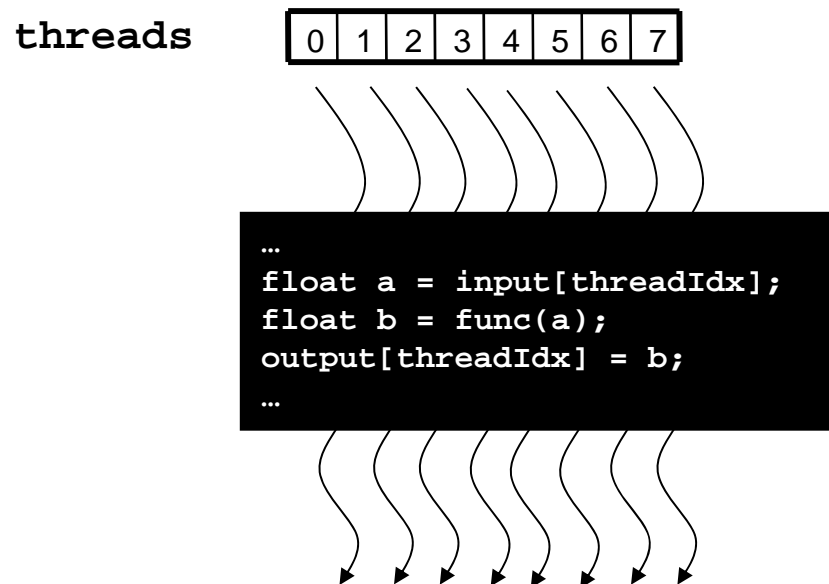
```
// Allocate GPU memory
void *myimage = cudaMalloc(bytes)
```

- **Function launch**

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

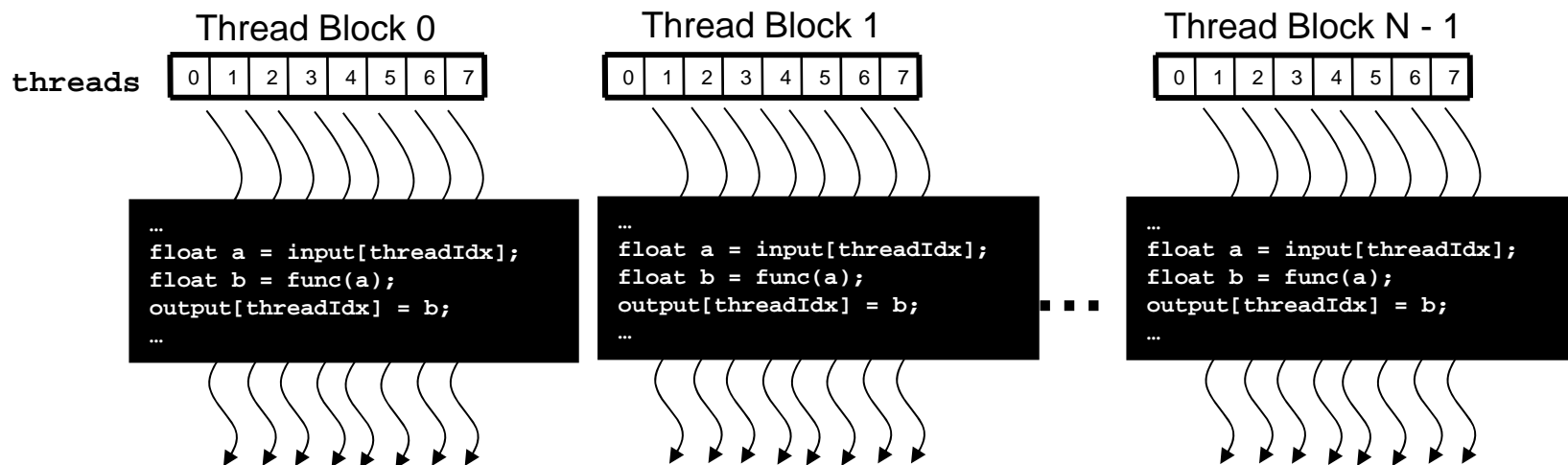

Arrays of Parallel Threads

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



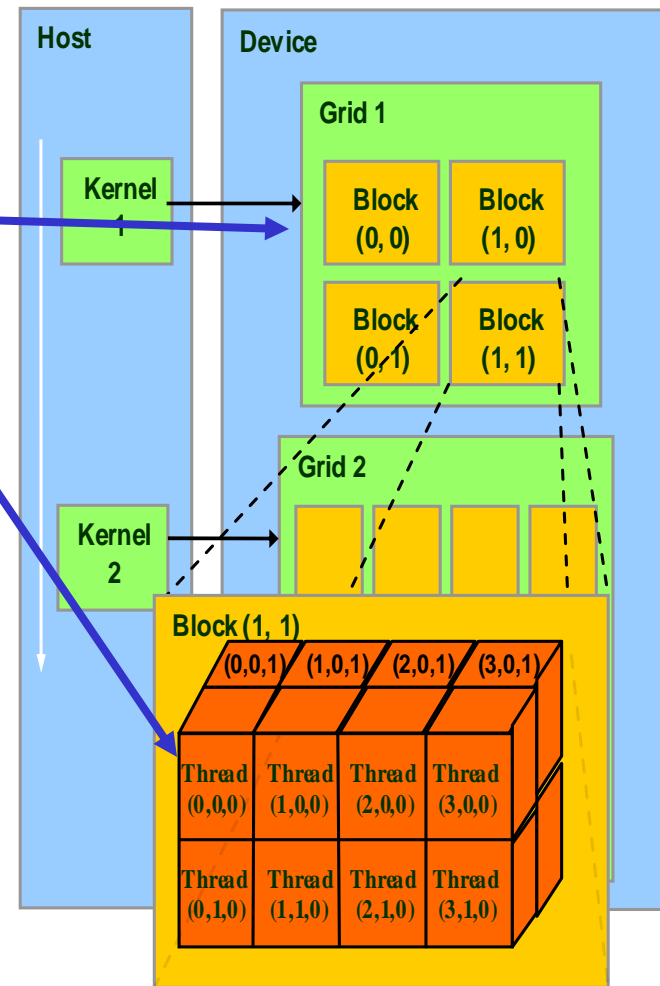
Thread Blocks: Scalable Cooperation

- 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



blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
 - blockIdx: 1D or 2D
 - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
 - Image processing
 - Solving PDEs on volumes
 - ...



Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B  
// Each thread performs one pair-wise addition
```

```
__global__
```

```
void vecAdd(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];
```

```
}
```

```
int main()
```

```
{
```

```
    // Run ceil(N/256) blocks of 256 threads each
```

```
    vecAdd<<<ceil(N/256), 256>>>(d_A, d_B, d_C, N);
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAdd(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];
}
```

Host Code

```
int main()
{
    // Run ceil(N/256) blocks of 256 threads each
    vecAdd<<<ceil(N/256), 256>>>(d_A, d_B, d_C, N);
}
```

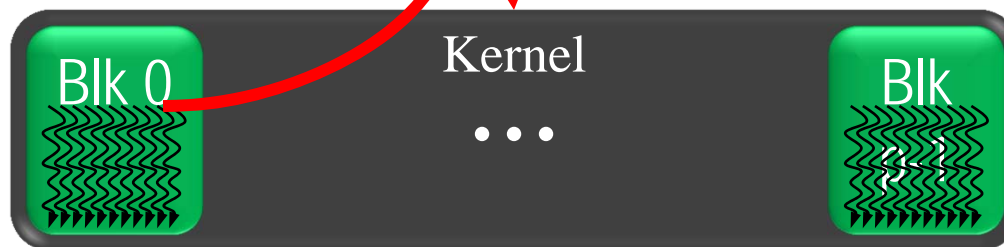
__host__

Kernel execution in a nutshell

__global__

```
vecAdd<<<P,B>>>(n,a,x,y);
```

```
blockIdx.x  blockDim.x  
threadIdx.x
```

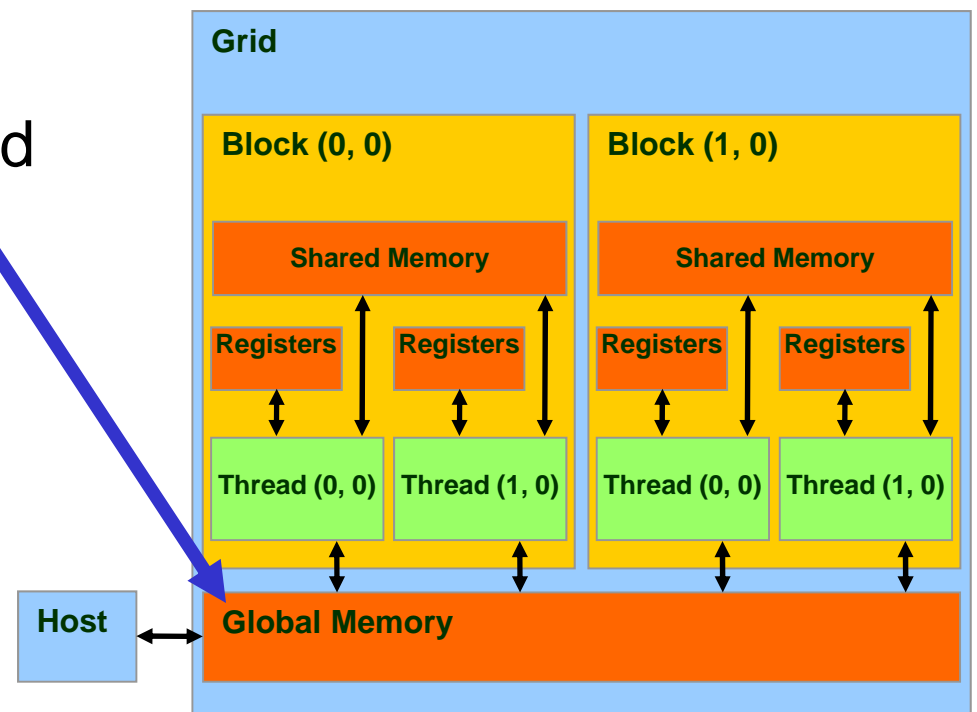


Schedule onto multiprocessors



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

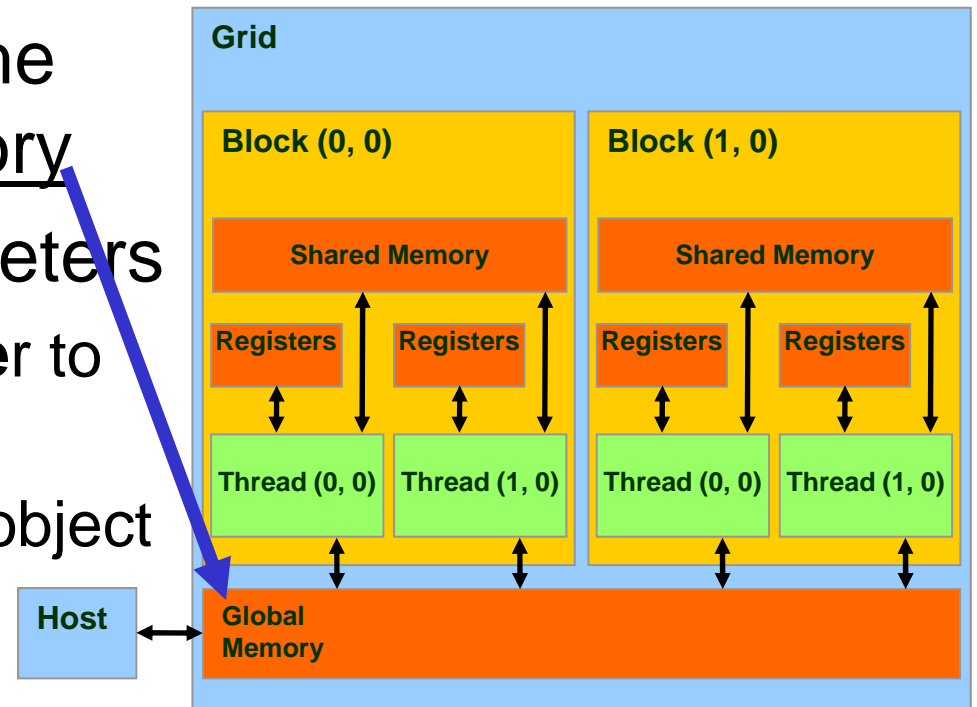


CUDA API Highlights: Easy and Lightweight

- The API is an extension to the ANSI C programming language
 - ➡ Low learning curve
- The hardware is designed to enable lightweight runtime and driver
 - ➡ High performance

CUDA Device Memory Allocation

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



CUDA Device Memory Allocation (cont.)

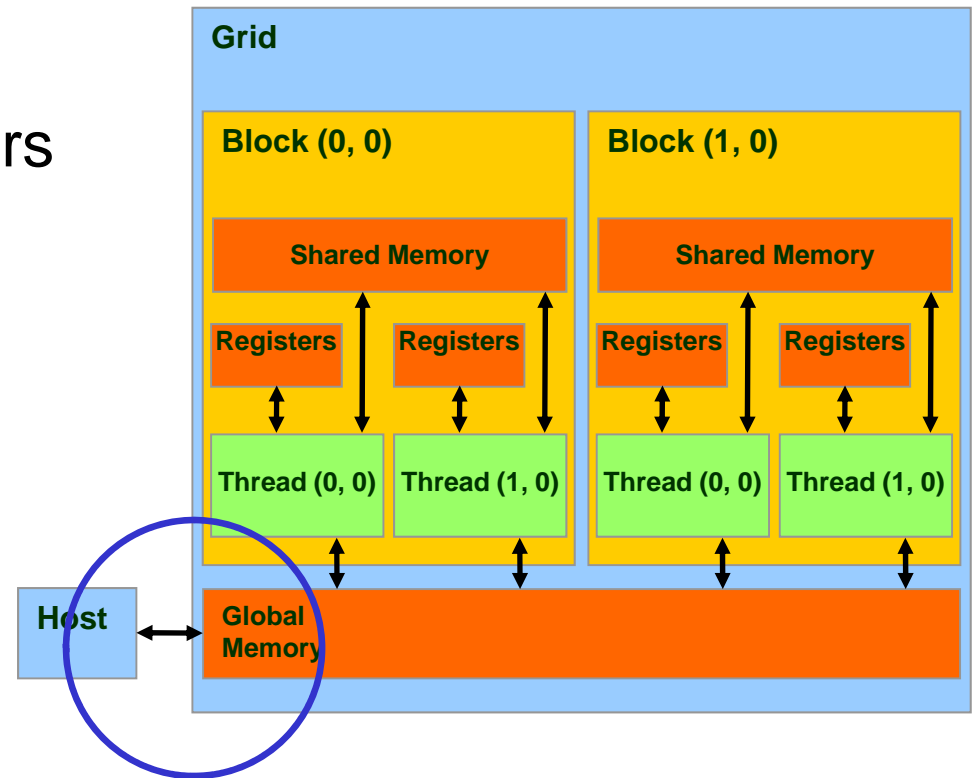
- Code example:
 - Allocate a 64 * 64 single precision float array
 - Attach the allocated storage to Md
 - “d” is often used to indicate a device data structure

```
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 later



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);`

Example: Host code for vecAdd

```
int main()
{
    // allocate and initialize host (CPU) memory
    float *h_A = ..., *h_B = ...;

    // allocate device (GPU) memory
    float *d_A, *d_B, *d_C;
    cudaMalloc( (void**) &d_A, N * sizeof(float));
    cudaMalloc( (void**) &d_B, N * sizeof(float));
    cudaMalloc( (void**) &d_C, N * sizeof(float));

    // copy host memory to device
    cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
    cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

    // Execute the kernel on ceil(N/256) blocks of 256 threads each
    vecAdd<<<ceil(N/256), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost) );

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

CUDA Keywords

CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function
 - Each “__” consists of two underscore characters
 - 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

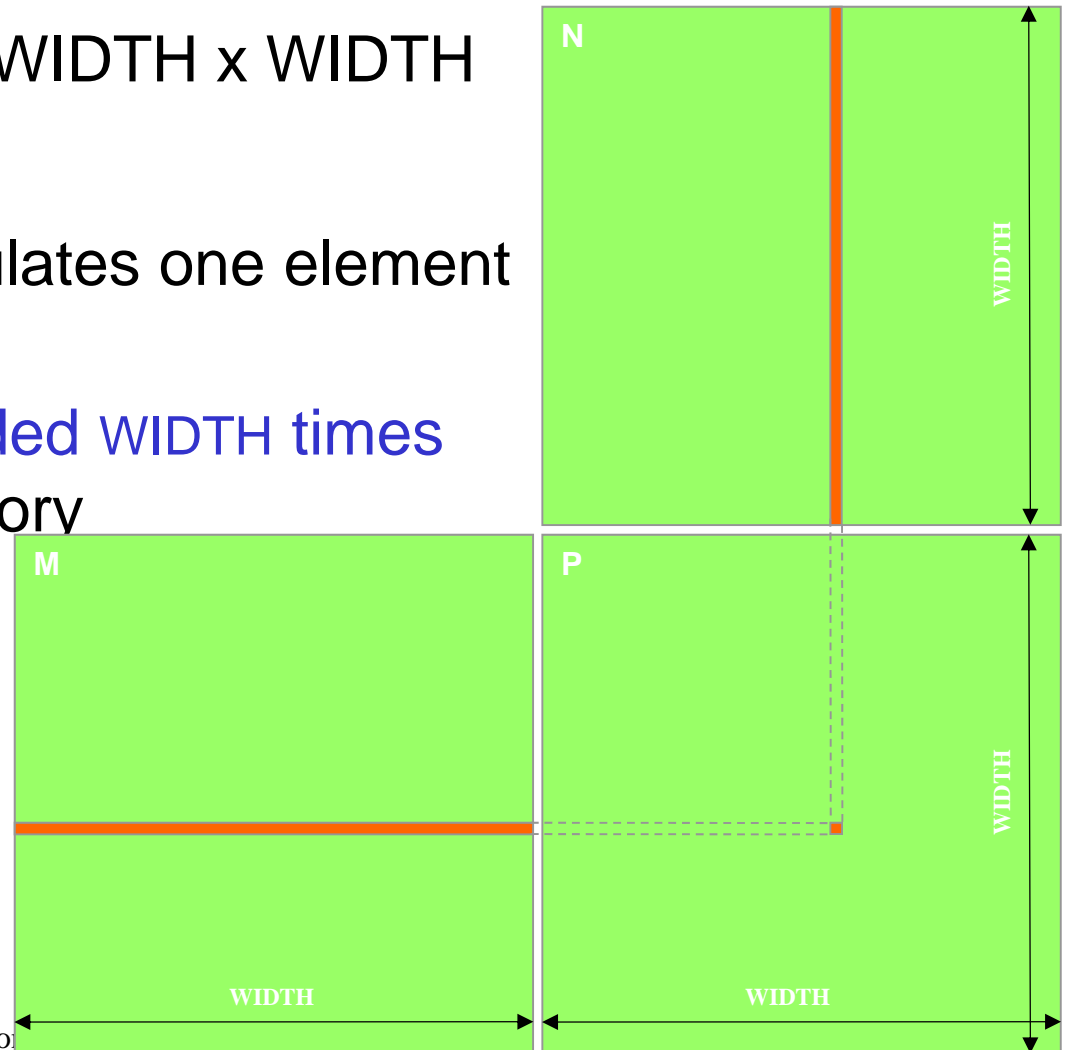
A Simple Running Example

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
 - Register usage
 - Thread index usage
 - Memory data transfer API between host and device
 - Assume square matrix for simplicity

Programming Model: Square Matrix-Matrix Multiplication Example

- $P = M * N$ of size $WIDTH \times WIDTH$
- 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];
```

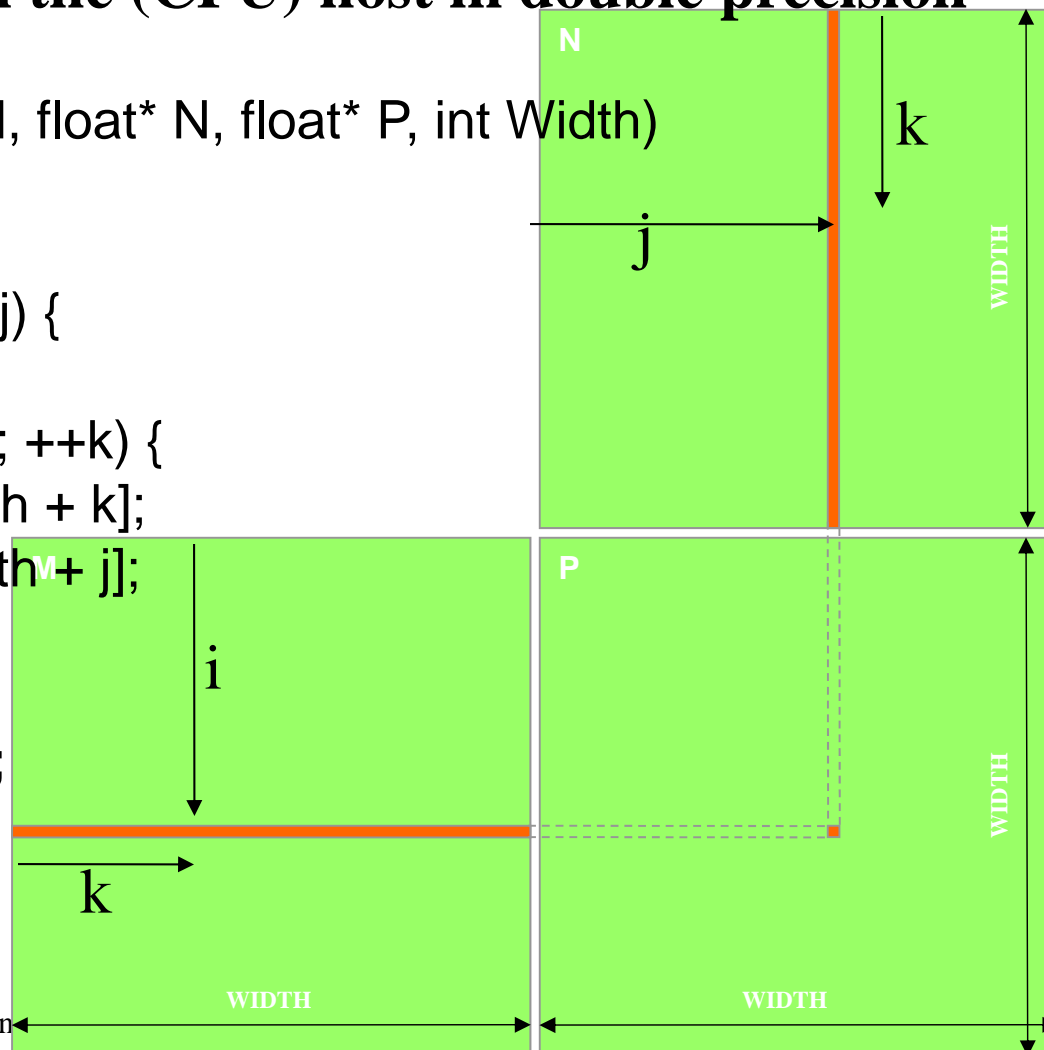
```
                sum += a * b;
```

```
            }
```

```
            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

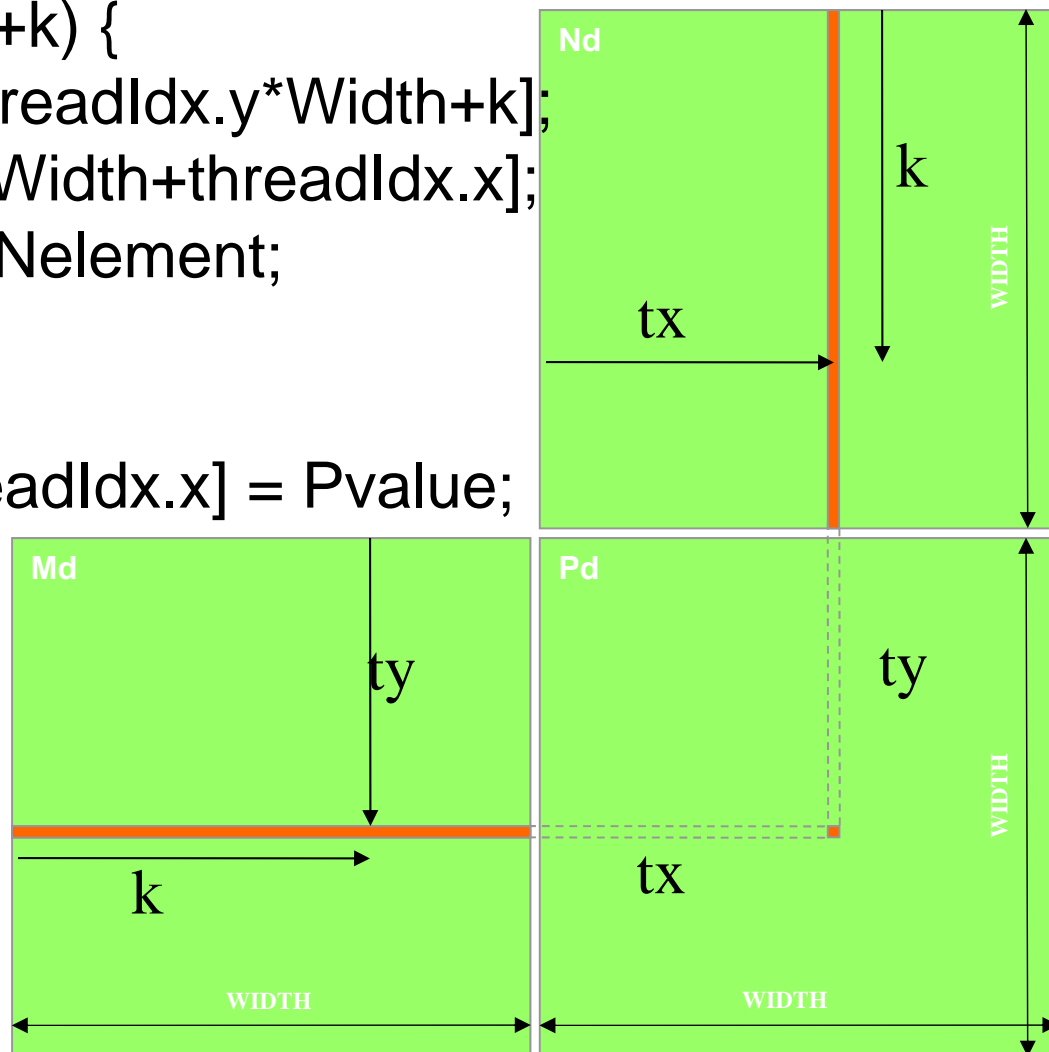
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
```

```
    // 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;  
}
```



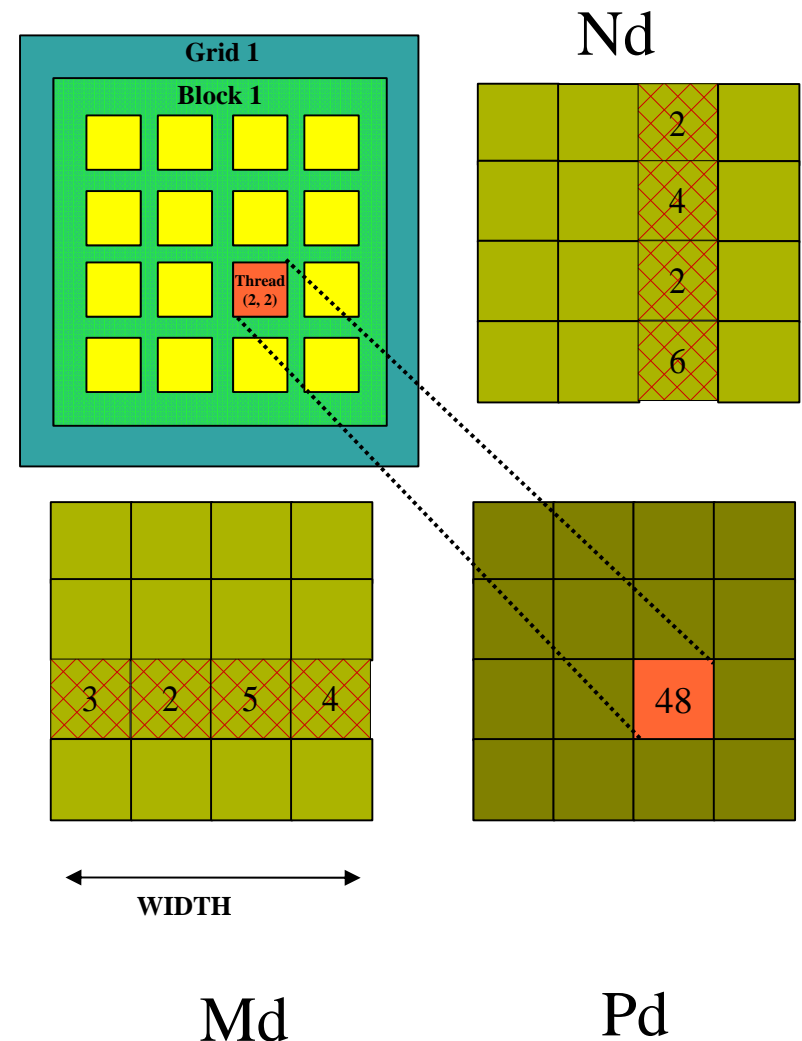
Step 5: Kernel Invocation (Host-side Code)

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

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

Need to Extend to Multiple Block

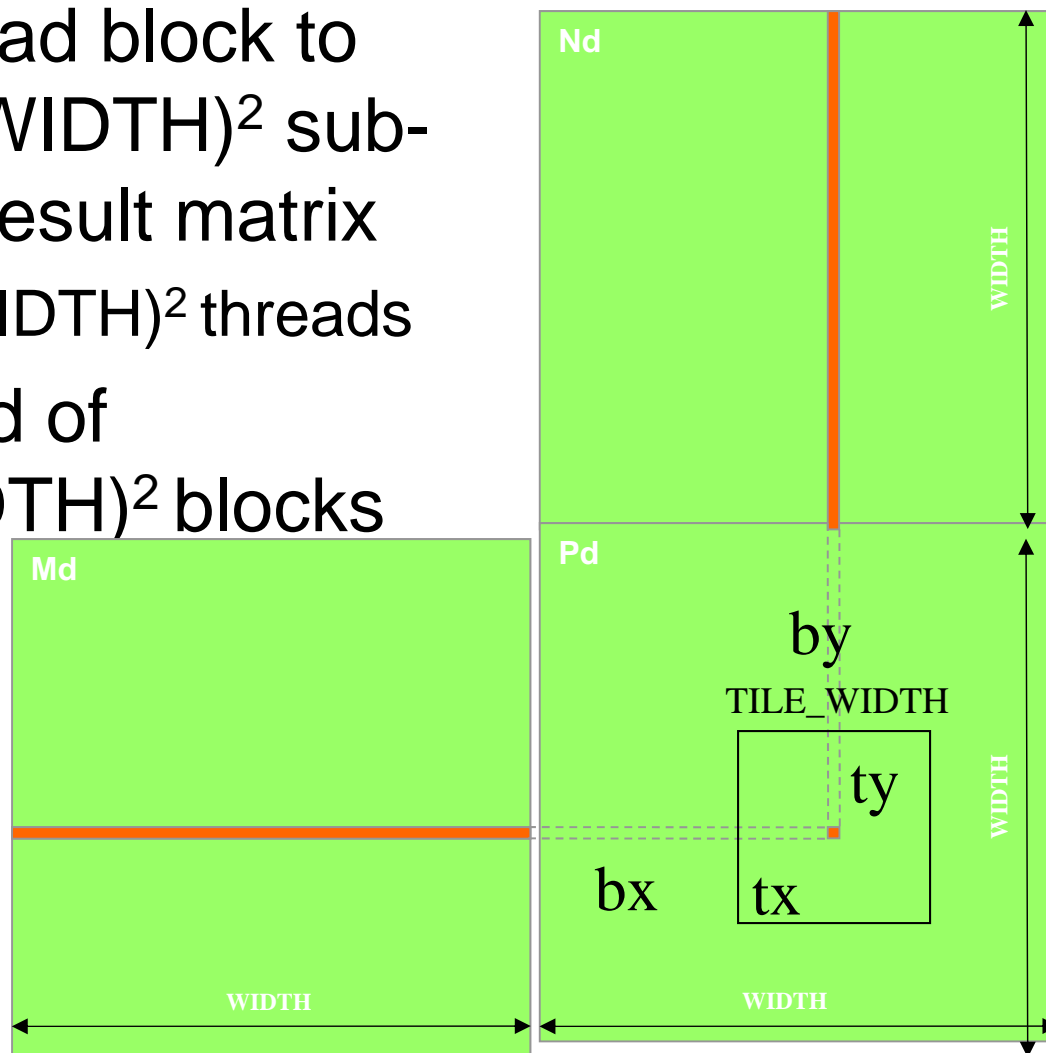
- One Block of threads compute matrix Pd
 - Each thread computes one element of Pd
- Each thread
 - Loads a row of matrix Md
 - Loads a column of matrix Nd
 - Perform one multiply and addition for each pair of Md and Nd elements
 - Compute to off-chip memory access ratio close to 1:1 (not very high)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



Step 7: Handling Arbitrary Sized Square Matrices

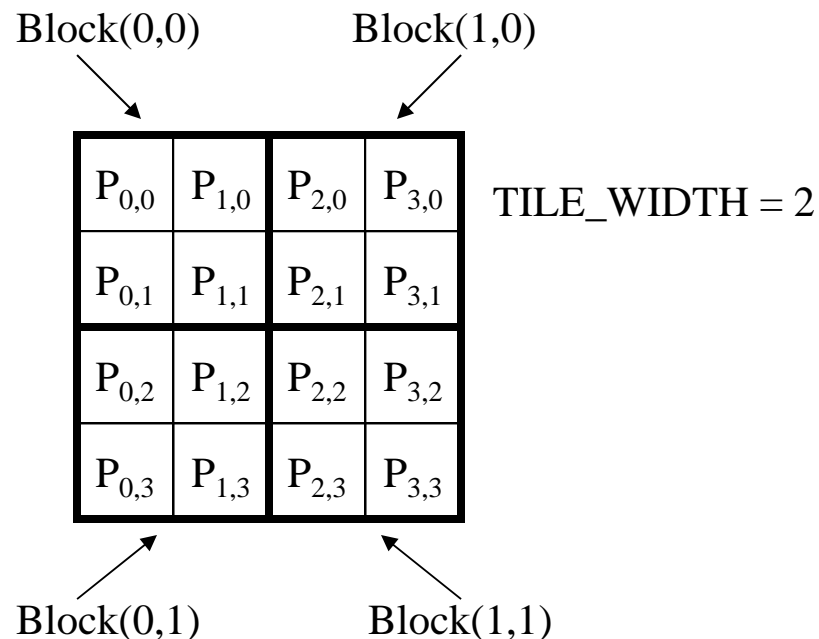
- Have each 2D thread block to compute a $(\text{TILE_WIDTH})^2$ sub-matrix (tile) of the result matrix
 - Each has $(\text{TILE_WIDTH})^2$ threads
- Generate a 2D Grid of $(\text{WIDTH}/\text{TILE_WIDTH})^2$ blocks

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

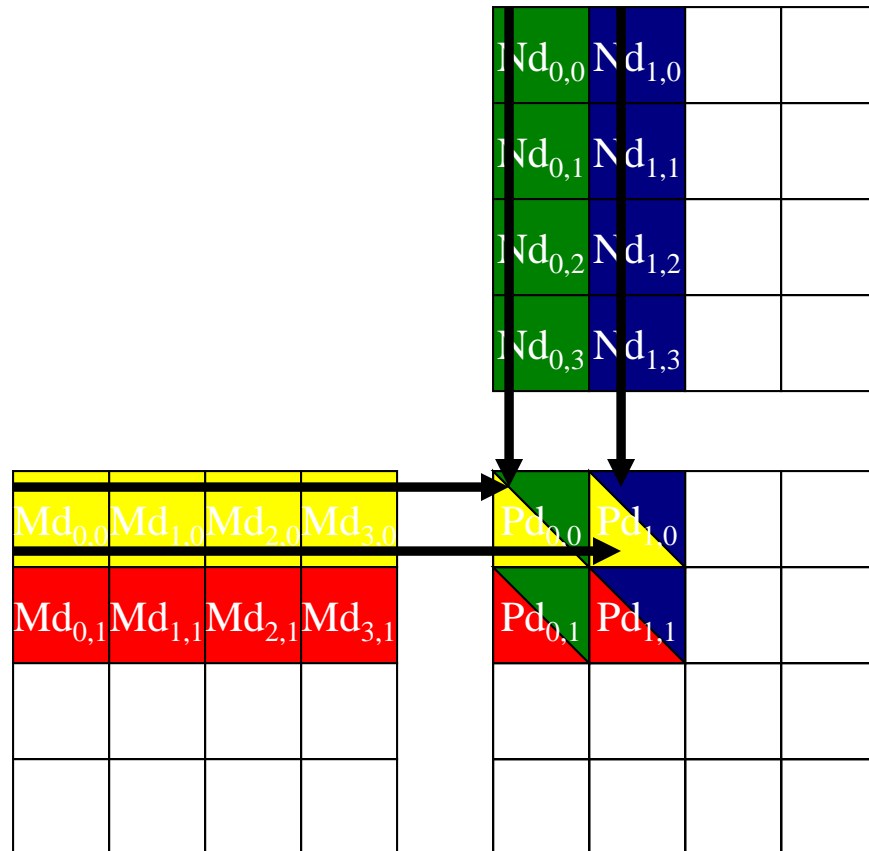


A Small Example

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



A Small Example: Multiplication



Revised Matrix Multiplication Kernel using Multiple Blocks

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column idenx of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

    Pd[Row*Width+Col] = Pvalue;
}
```

Revised Step 5: Kernel Invocation (Host-side Code)

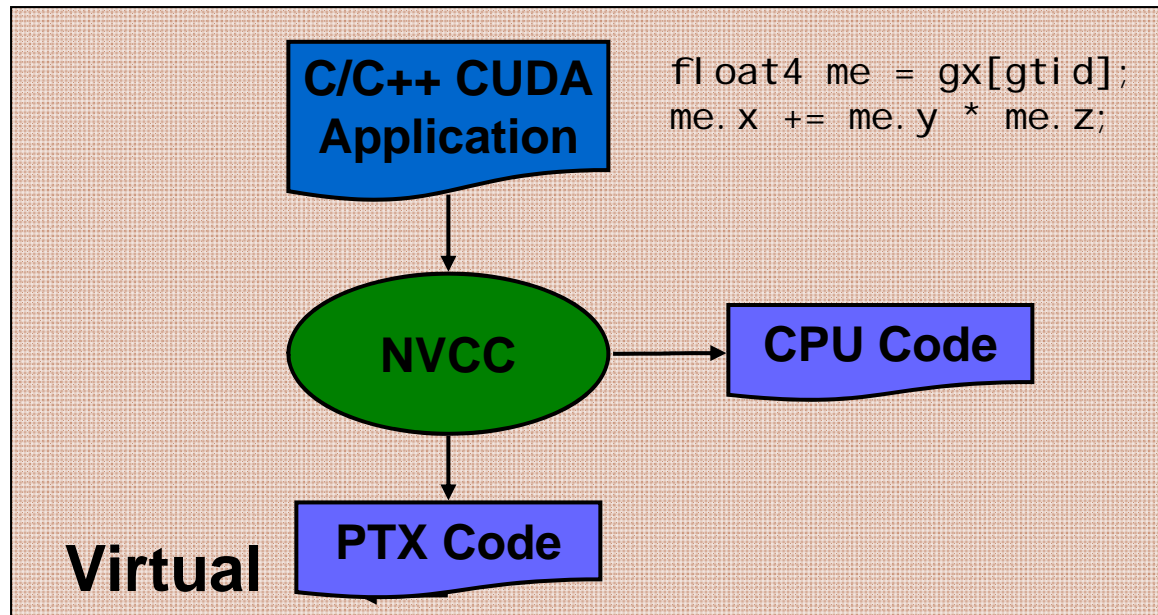
```
// Setup the execution configuration
dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

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

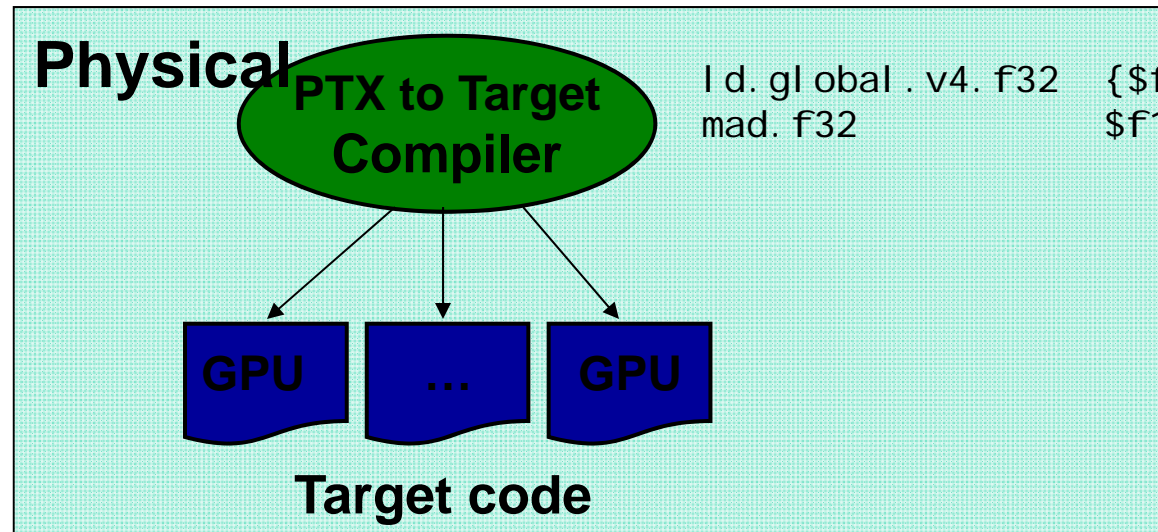

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

Some Useful Information on Tools

Compiling a CUDA Program



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



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 (`cuda`)
 - The CUDA core library (`cuda`)