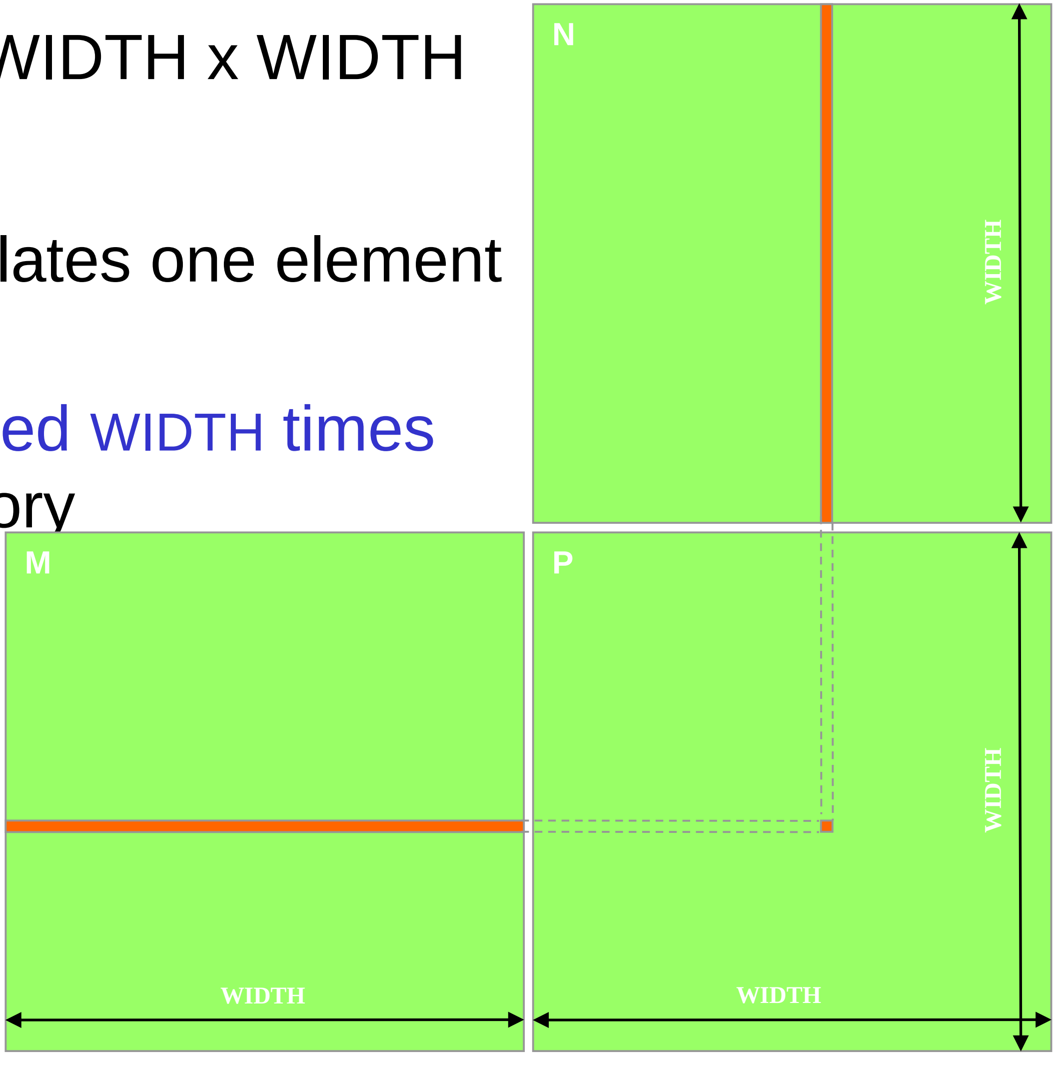
Matrix Multiplication in CUDA

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
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory



Memory Layout of a Matrix in C

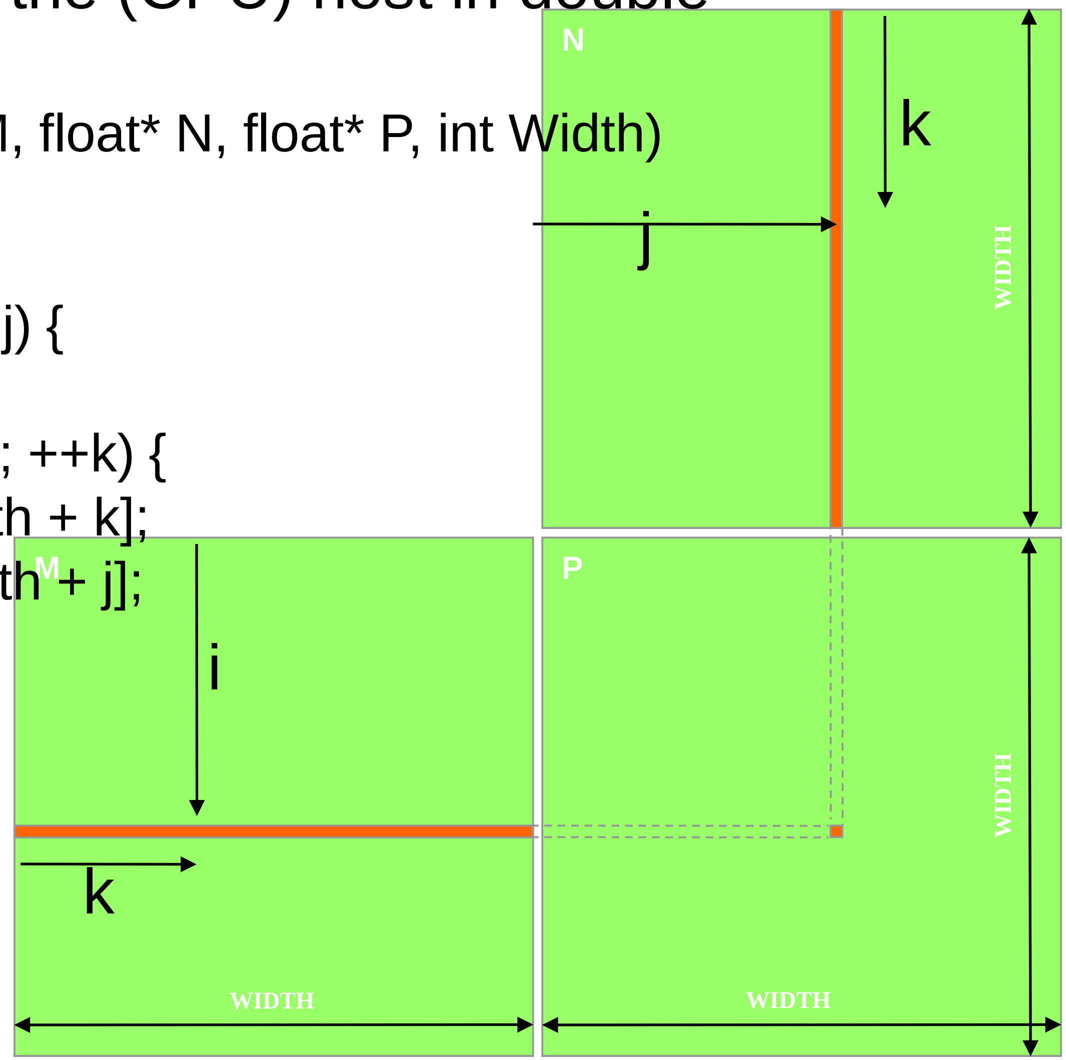
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

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);

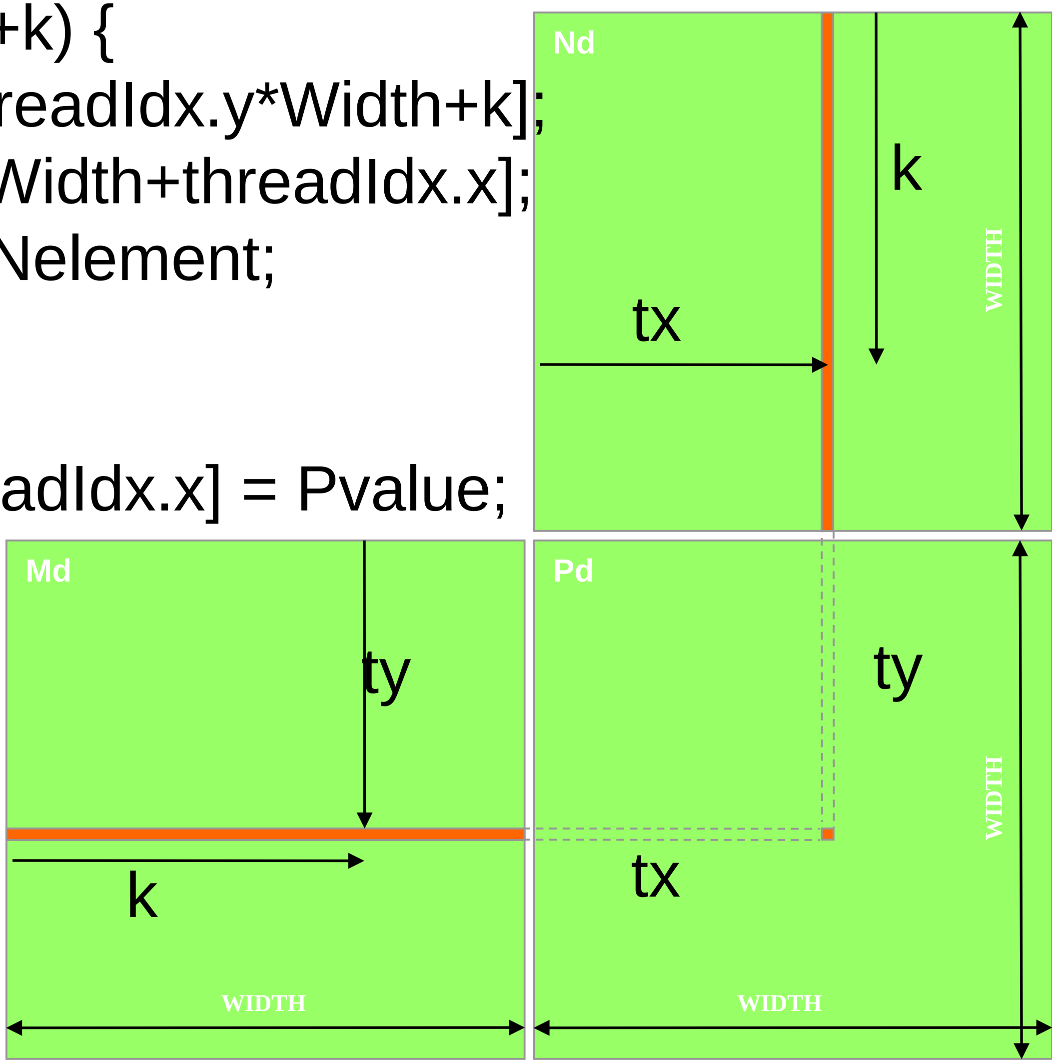
```
 // Free device matrices
cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

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;
}
```

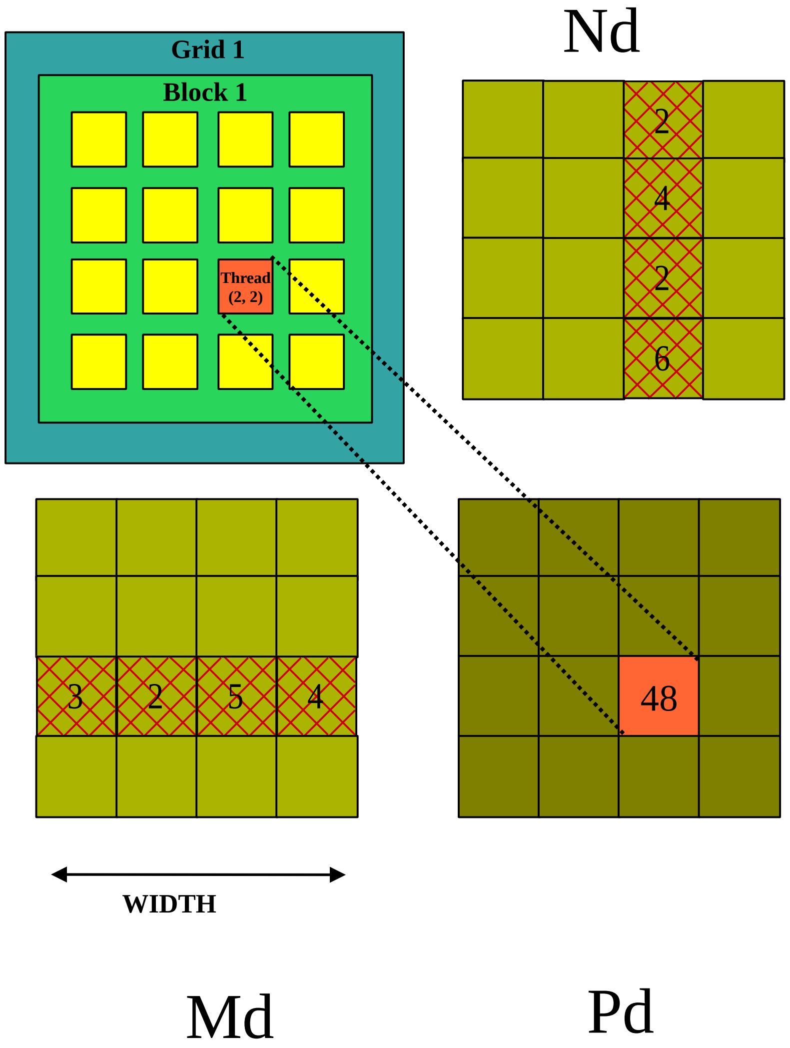


Step 5: Kernel Invocation (Host-side Code)

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

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

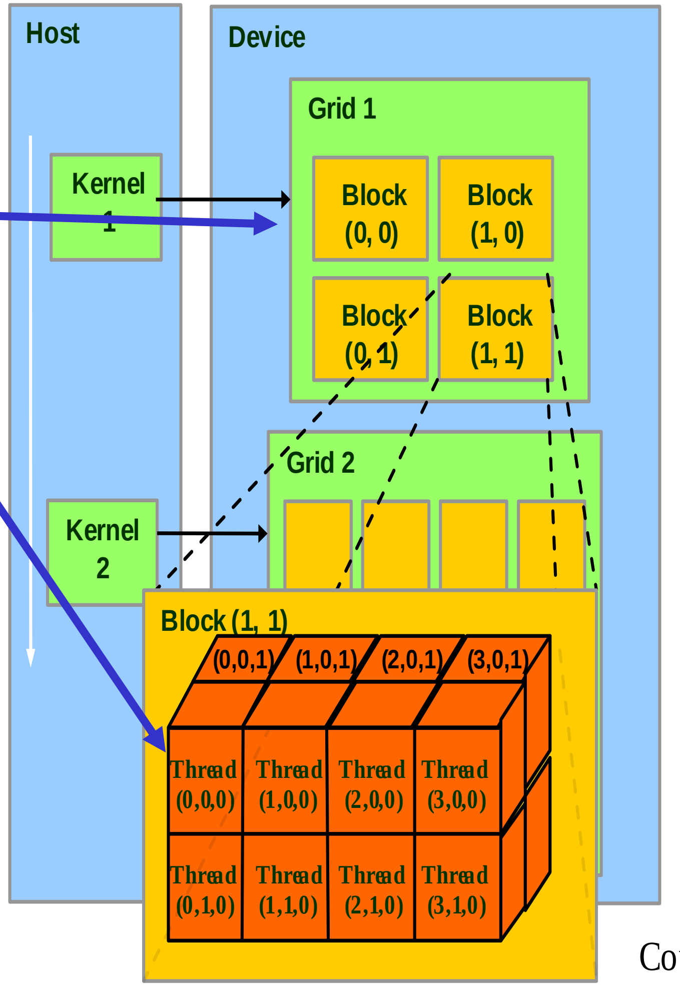
Only One Thread Block Used

- 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
- Size of matrix limited by the number of threads allowed in a thread block



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

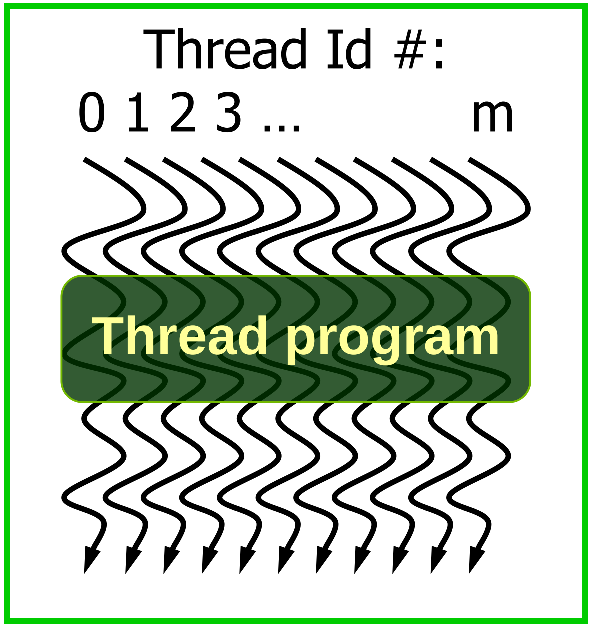


Courtesy: NDVIA

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!

CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

Language Extensions: Built-in Variables

- **`dim3 gridDim;`**
  - Dimensions of the grid in blocks (**`gridDim.z`** unused)
- **`dim3 blockDim;`**
  - Dimensions of the block in threads
- **`dim3 blockIdx;`**
  - Block index within the grid
- **`dim3 threadIdx;`**
  - Thread index within the block