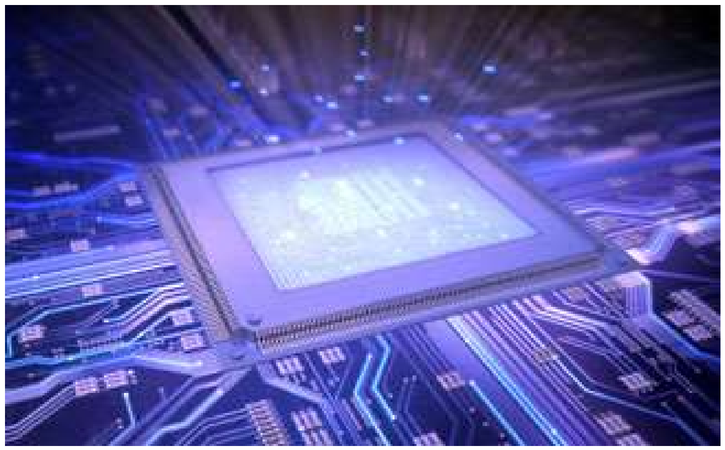
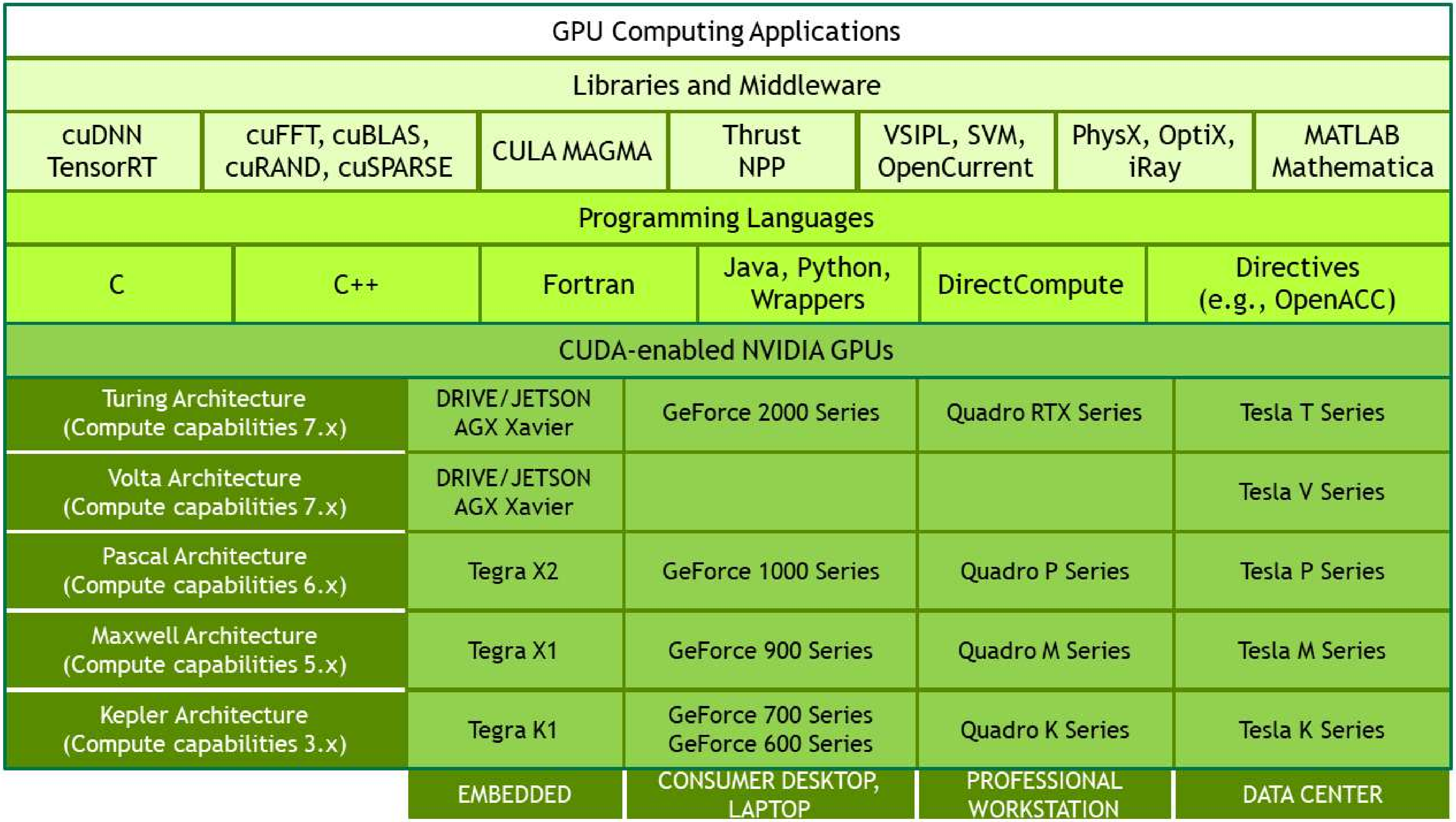
Parallel Computing

CUDA - I

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





GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)

CUDA-enabled NVIDIA GPUs

Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series
	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER

Source: CUDA Toolkit Documentation by NVIDIA

Parallel Computing on a GPU

- GPUs deliver up to 13,800+ GFLOPS (FP32)
  - Available in laptops, clusters, etc.

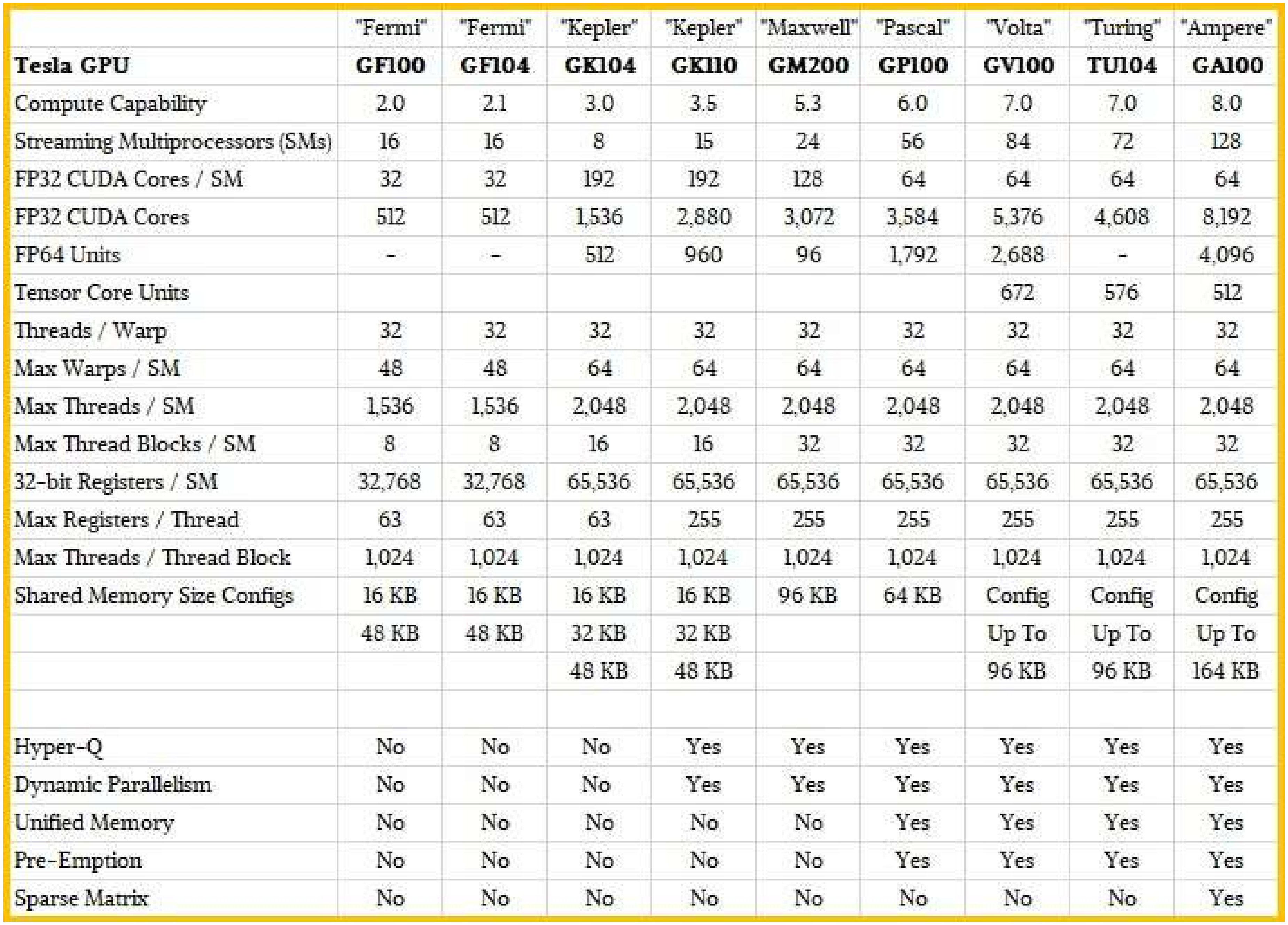
- GPU parallelism is doubling almost every year
- Programming model scales transparently
  - Data parallelism

- Programmable in C/C++ (and other languages) with CUDA tools

- Multithreaded SPMD model uses application data parallelism and thread parallelism.
  [SPMD = Single Program Multiple Data]

GeForce RTX 3080

Compute Capability



	"Fermi"	"Fermi"	"Kepler"	"Kepler"	"Maxwell"	"Pascal"	"Volta"	"Turing"	"Ampere"
Tesla GPU	**GF100**	**GF104**	**GK104**	**GK110**	**GM200**	**GP100**	**GV100**	**TU104**	**GA100**
Compute Capability	2.0	2.1	3.0	3.5	5.3	6.0	7.0	7.0	8.0
Streaming Multiprocessors (SMs)	16	16	8	15	24	56	84	72	128
FP32 CUDA Cores / SM	32	32	192	192	128	64	64	64	64
FP32 CUDA Cores	512	512	1,536	2,880	3,072	3,584	5,376	4,608	8,192
FP64 Units	-	-	512	960	96	1,792	2,688	-	4,096
Tensor Core Units							672	576	512
Threads / Warp	32	32	32	32	32	32	32	32	32
Max Warps / SM	48	48	64	64	64	64	64	64	64
Max Threads / SM	1,536	1,536	2,048	2,048	2,048	2,048	2,048	2,048	2,048
Max Thread Blocks / SM	8	8	16	16	32	32	32	32	32
32-bit Registers / SM	32,768	32,768	65,536	65,536	65,536	65,536	65,536	65,536	65,536
Max Registers / Thread	63	63	63	255	255	255	255	255	255
Max Threads / Thread Block	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024
Shared Memory Size Configs	16 KB	16 KB	16 KB	16 KB	96 KB	64 KB	Config	Config	Config
	48 KB	48 KB	32 KB	32 KB			Up To	Up To	Up To
			48 KB	48 KB			96 KB	96 KB	164 KB
Hyper-Q	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic Parallelism	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Unified Memory	No	No	No	No	No	Yes	Yes	Yes	Yes
Pre-Emption	No	No	No	No	No	Yes	Yes	Yes	Yes
Sparse Matrix	No	No	No	No	No	No	No	No	Yes

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)

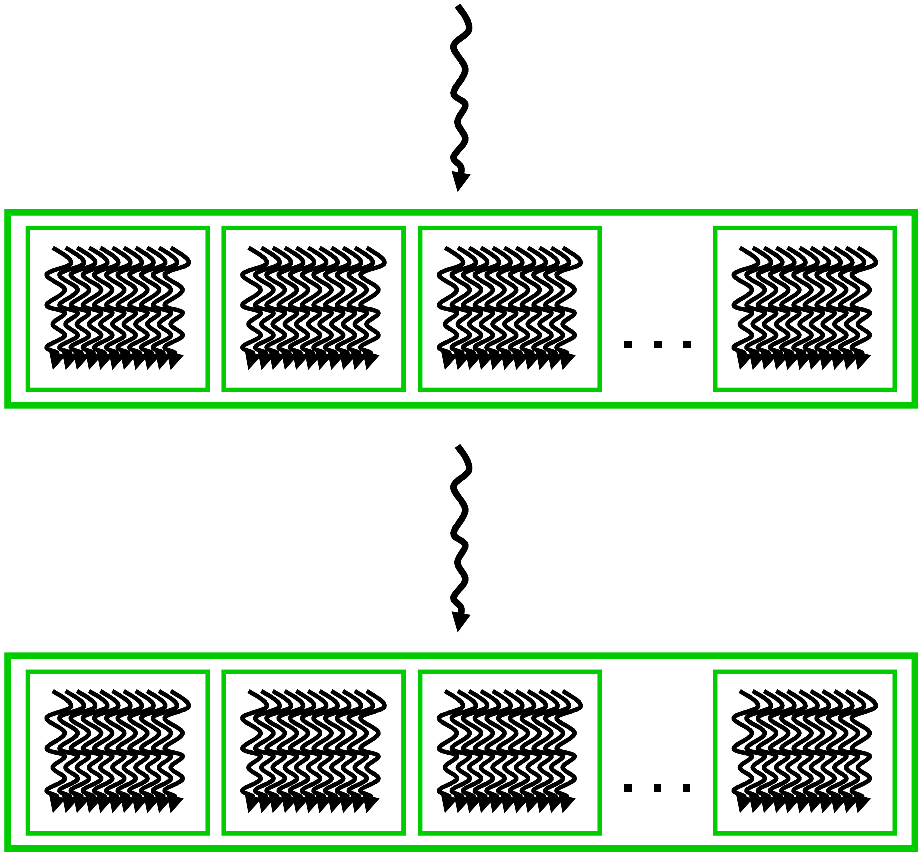
Parallel Kernel (device)

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

Serial Code (host)

Parallel Kernel (device)

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



```
#include <stdio.h>
#include <cuda.h>   /* Header file for CUDA */

/* Device code:  runs on GPU */
__global__ void Hello(void) {

   printf("Hello from thread %d!\n", threadIdx.x);
}  /* Hello */


/* Host code:  Runs on CPU */
int main(int argc, char* argv[]) {
   int thread_count;     /* Number of threads to run on GPU */

   thread_count = strtol(argv[1], NULL, 10);
                         /* Get thread_count from command line */

   Hello <<<1, thread_count>>>();
                         /* Start thread_count threads on GPU, */

   cudaDeviceSynchronize();      /* Wait for GPU to finish */

   return 0;
}  /* main */
```

```c
#include <stdio.h>
#include <cuda.h>   /* Header file for CUDA */

/* Device code:  runs on GPU */
__global__ void Hello(void) {

   printf("Hello from thread %d!\n", threadIdx.x);
}  /* Hello */


/* Host code:  Runs on CPU */
int main(int argc, char* argv[]) {
   int thread_count;      /* Number of threads to run on GPU */

   thread_count = strtol(argv[1], NULL, 10);
                        /* Get thread_count from command line */

   Hello <<<1, thread_count>>>();
                        /* Start thread_count threads on GPU, */

   cudaDeviceSynchronize();       /* Wait for GPU to finish */

   return 0;
}  /* main */
```

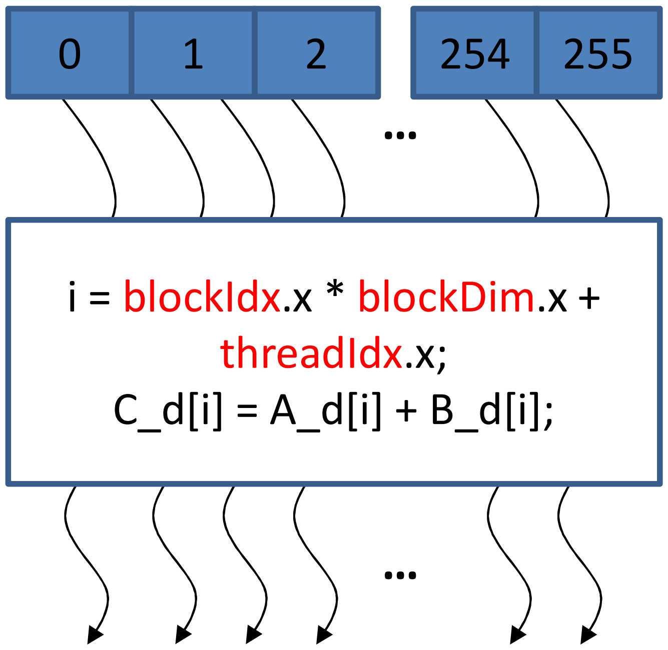
```
$ ./cuda_hello 10
```

and the output of will be

```
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
Hello from thread 3!
Hello from thread 4!
Hello from thread 5!
Hello from thread 6!
Hello from thread 7!
Hello from thread 8!
Hello from thread 9!
```

Parallel Threads

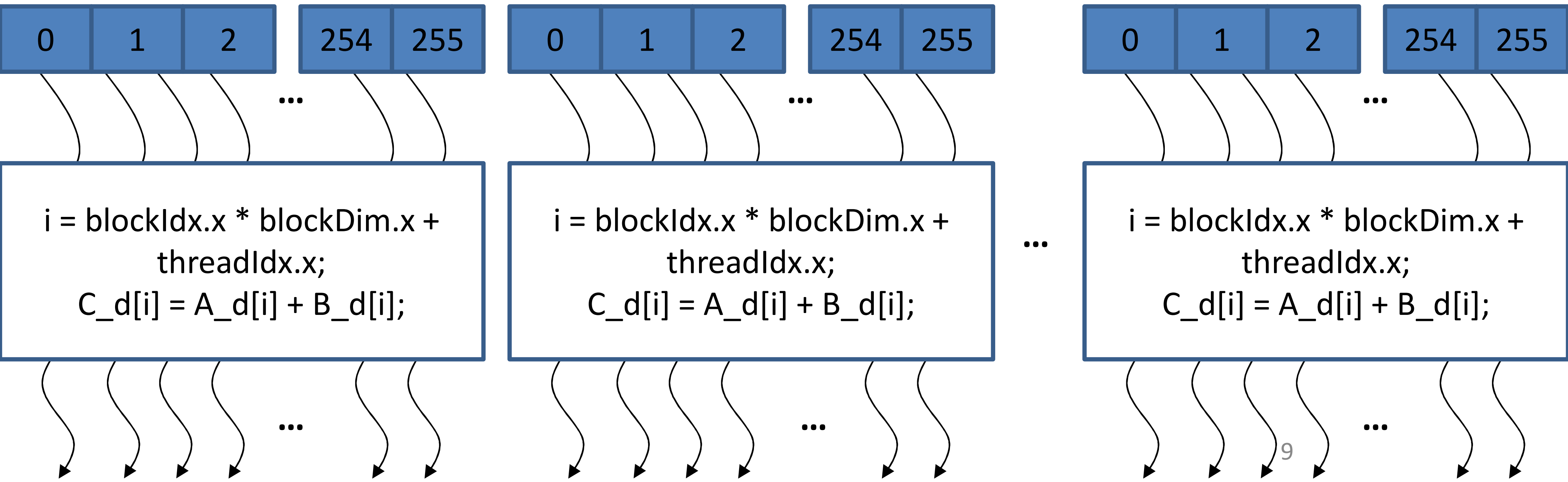
- A CUDA kernel is executed by an array of threads
  - All threads run the same code (the SP in SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



```
i = blockIdx.x * blockDim.x +
threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

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


0
1
2
254
255
…
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];

Kernel

- Launched by the host
- Very similar to a C function
- To be executed on device
- All threads will execute that same code in the kernel.

$\downarrow$

Grid

- 1D, 2D, or 3D organization of a Grid
- gridDim.x, gridDim.y, gridDim.z are the size of the grid in number of blocks

$\downarrow$

Block

- 1D, 2D, or 3D organization of a block
- Block is assigned to an SM
- blockDim.x, blockDim.y, blockDim.z are block dimensions counted as number of threads
- blockIdx.x, blockIdx.y, blockIdx.z are indices of the block within a GRID.

$\downarrow$

Thread

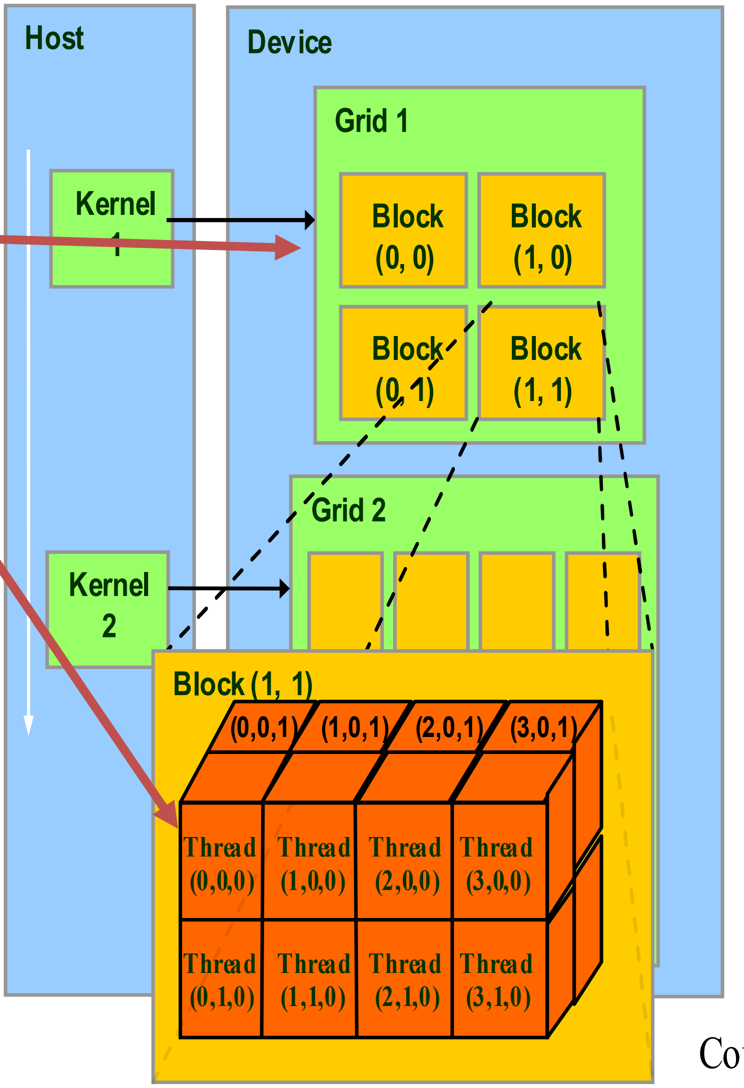
- threadIdx.x, threadIdx.y, threadIdx.z are the index *within* a block

Decisions you have to make as a GPU programmers:

1. Which part(s) of the program will be executed on the GPU?
2. How many total threads will you spawn?
3. How many blocks? That is: how many threads per block?
4. What will be the geometry of the block (1D, 2D, or 3D)?
5. What will be the geometry of the grid (1D, 2D, or 3D)?

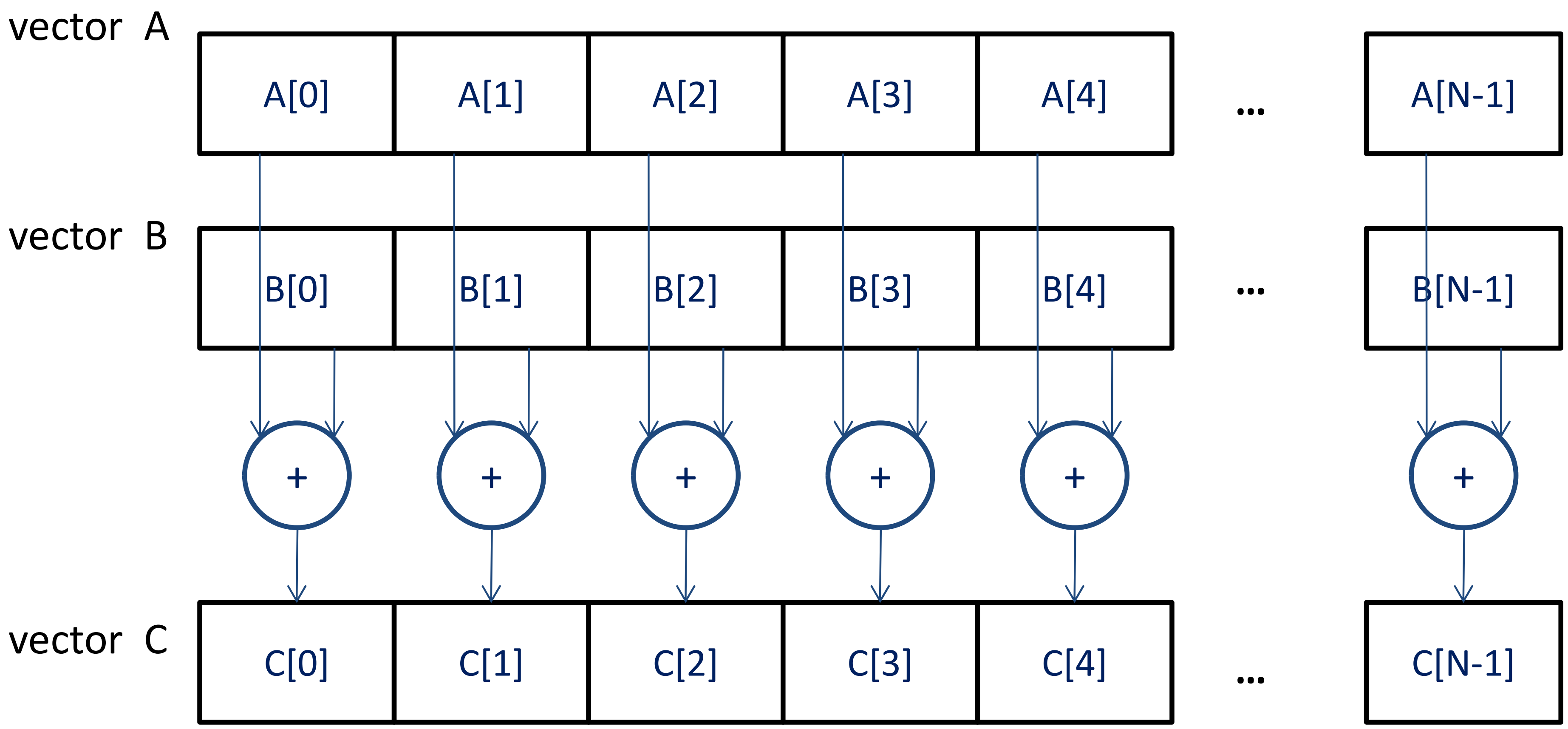
IDs

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



Courtesy: NDVIA

A Simple Example: Vector Addition



A Simple Example: Vector Addition

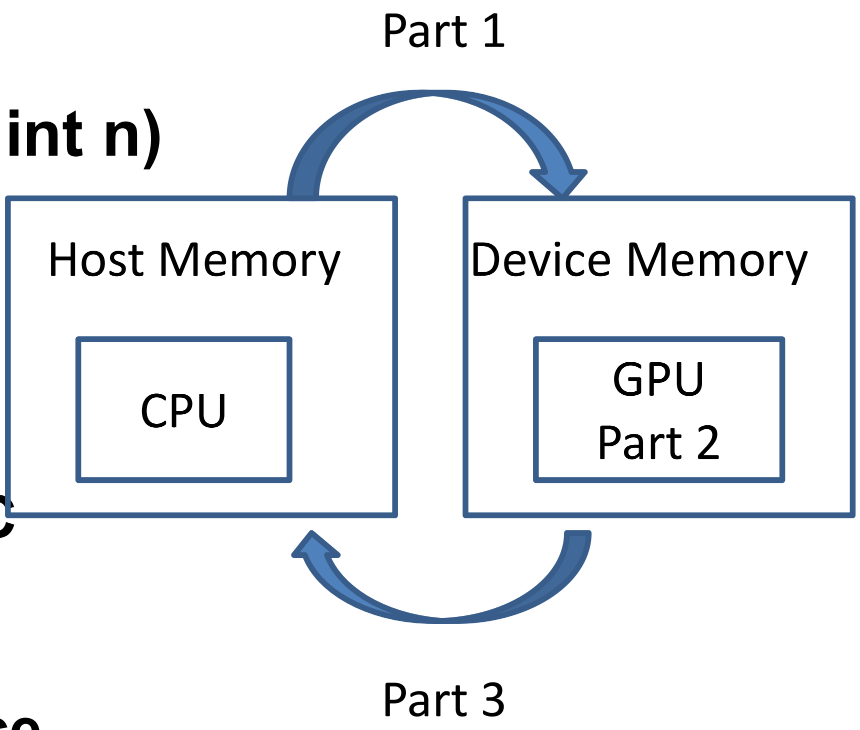
```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
  for (i = 0, i < n, i++)
    C[i] = A[i] + B[i];
}
```

GPU friendly!

```
int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    …
    vecAdd(A_h, B_h, C_h, N);
}
```

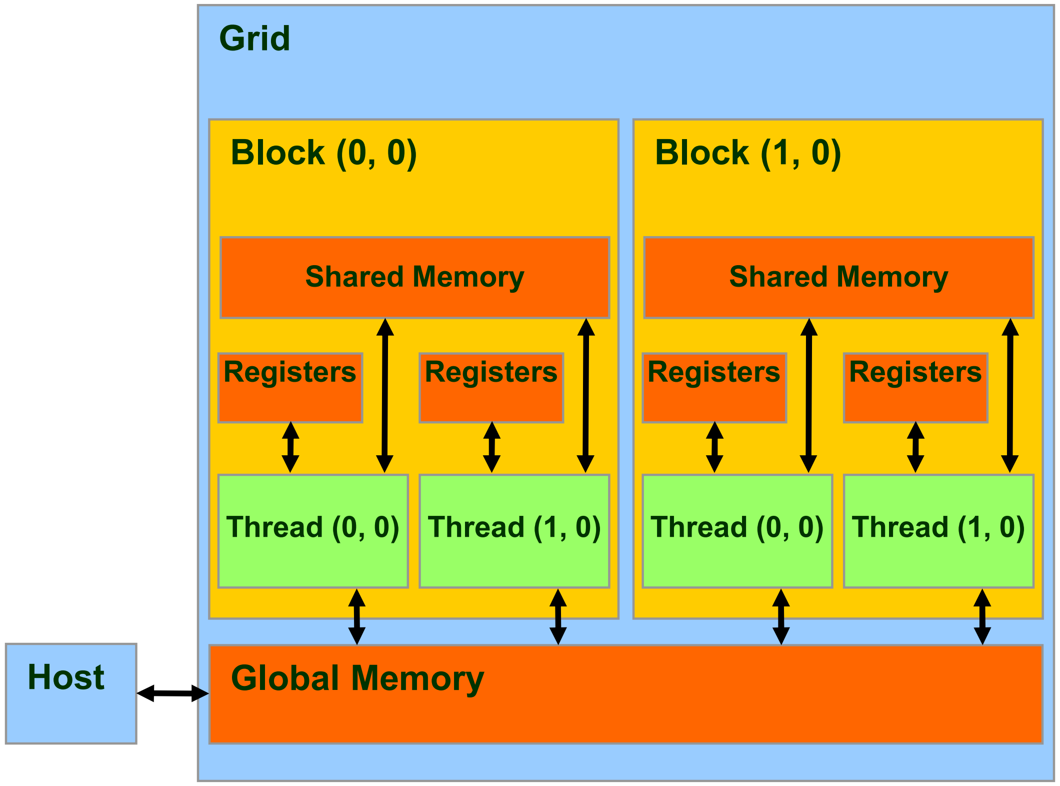
A Simple Example: Vector Addition

```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
  int size = n* sizeof(float);
  float* A_d, B_d, C_d;
  …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```



CUDA Memory Model

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- Shared memory:
  - Per SM
  - Shared by all threads in a block

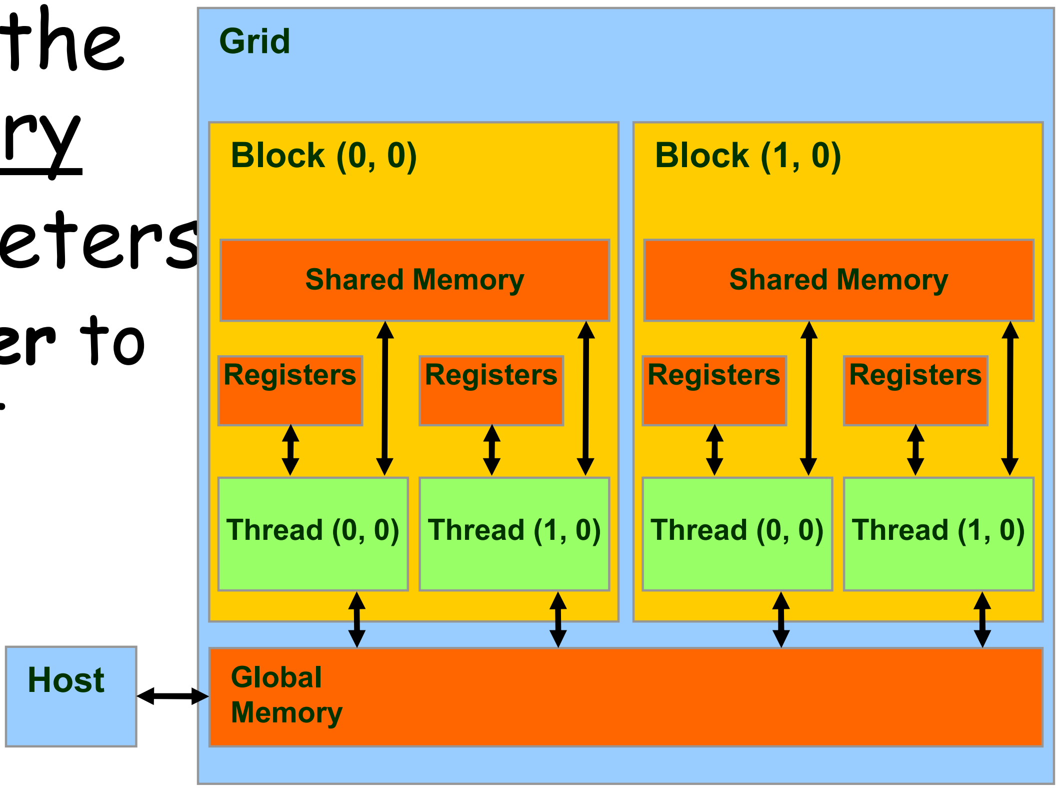


CPU & GPU Memory

- In CUDA, host and devices have separate memory spaces.
  - But in recent GPUs we have Unified Memory Access
- If GPU and CPU are on the same chip, then they share memory space → fusion

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** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object

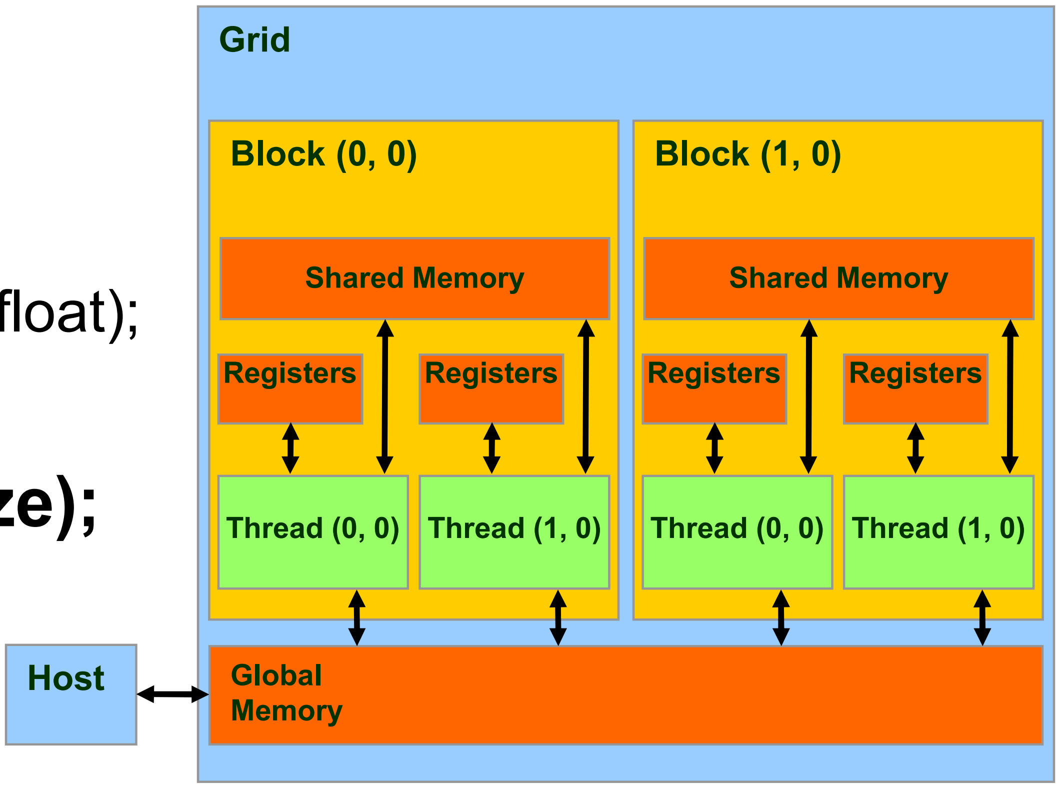


CUDA Device Memory Allocation

Example:

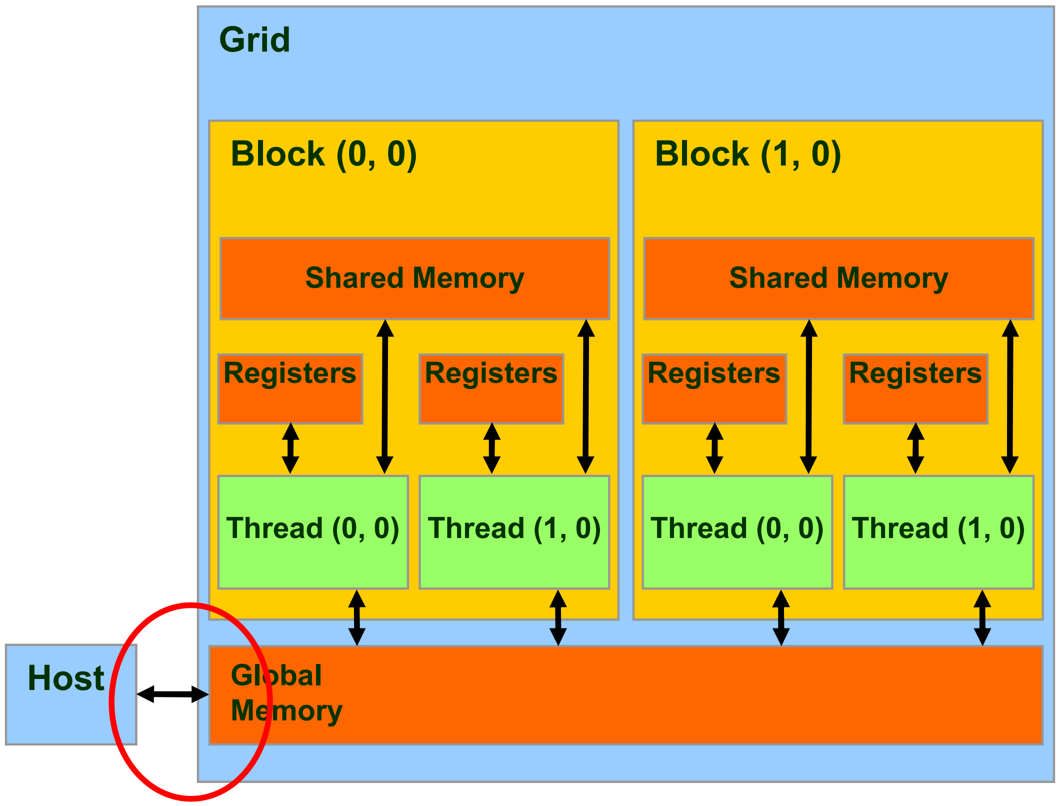
```
WIDTH = 64;
float* Md
int size = WIDTH * WIDTH * sizeof(float);
```

```
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

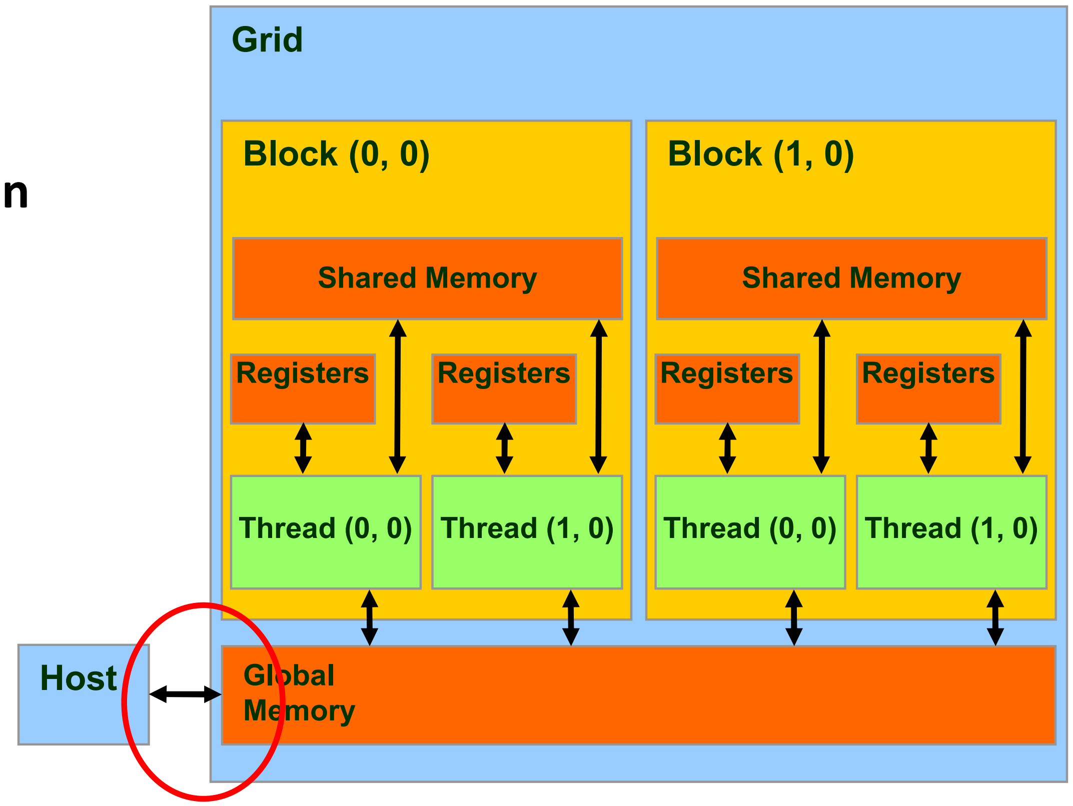


Important!
cudaMemcpy() cannot be used to copy between different GPUs in multi-GPUs system

CUDA Device Memory Allocation

Example:

Destination pointer **Source pointer** **Size in bytes** **Direction**



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

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

Note About Error Handling

- Almost all API calls return success or failure.
- The type of the outcome is: cudaError_t
- Success → cudaSuccess
- Translate the error code to an error message:

char * cudaGetErrorString (cudaError_t error)

A Simple Example: Vector Addition

```
void vecAdd(float* A, float* B, float* C, int n)
{
  int size = n * sizeof(float);
   float* A_d, *  B_d, * C_d;

1. // Transfer A and B to device memory
   cudaMalloc((void **) &A_d, size);
   cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &B_d, size);
   cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    // Allocate device memory for
    cudaMalloc((void **) &C_d, size);

2.  // Kernel invocation code – to be shown later
  …
3.   // Transfer C from device to host
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
      // Free device memory for A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

How to launch a kernel?

```
int vecAdd(float* A, float* B, float* C, int n)
{
 // A_d, B_d, C_d allocations and copies omitted
 // Run ceil(n/256) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256),256>>>(A_d, B_d, C_d, n);
}
```

#blocks **#threads/blks**

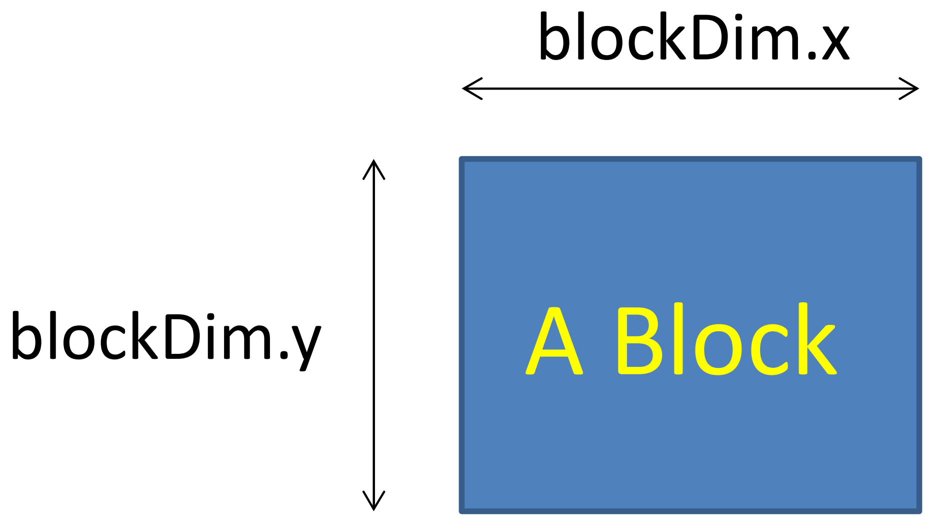
```
// Each thread performs one pair-wise addition
__global__
void vecAddkernel(float* A_d, float* B_d, float* C_d, int n)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

Unique ID
1D grid of 1D blocks

```
blockIdx.x *blockDim.x + threadIdx.x;
```

Unique ID
1D grid of 2D blocks

```
blockIdx.x * blockDim.x * blockDim.y +
threadIdx.y * blockDim.x +
threadIdx.x;
```



Unique ID
Example: 1D grid of 3D blocks

```
blockIdx.x * blockDim.x * blockDim.y *
blockDim.z +
threadIdx.z * blockDim.y * blockDim.x +
threadIdx.y * blockDim.x +
threadIdx.x;
```

Unique ID
Example: 2D grid of 1D blocks

```
int blockId   = blockIdx.y * gridDim.x +
blockIdx.x;

int threadId = blockId * blockDim.x +
threadIdx.x;
```

Unique ID

It is all a question of how to index data using thread and block IDs.

You can generate unique IDs for any combination of block and grid dimensions.

Kernels

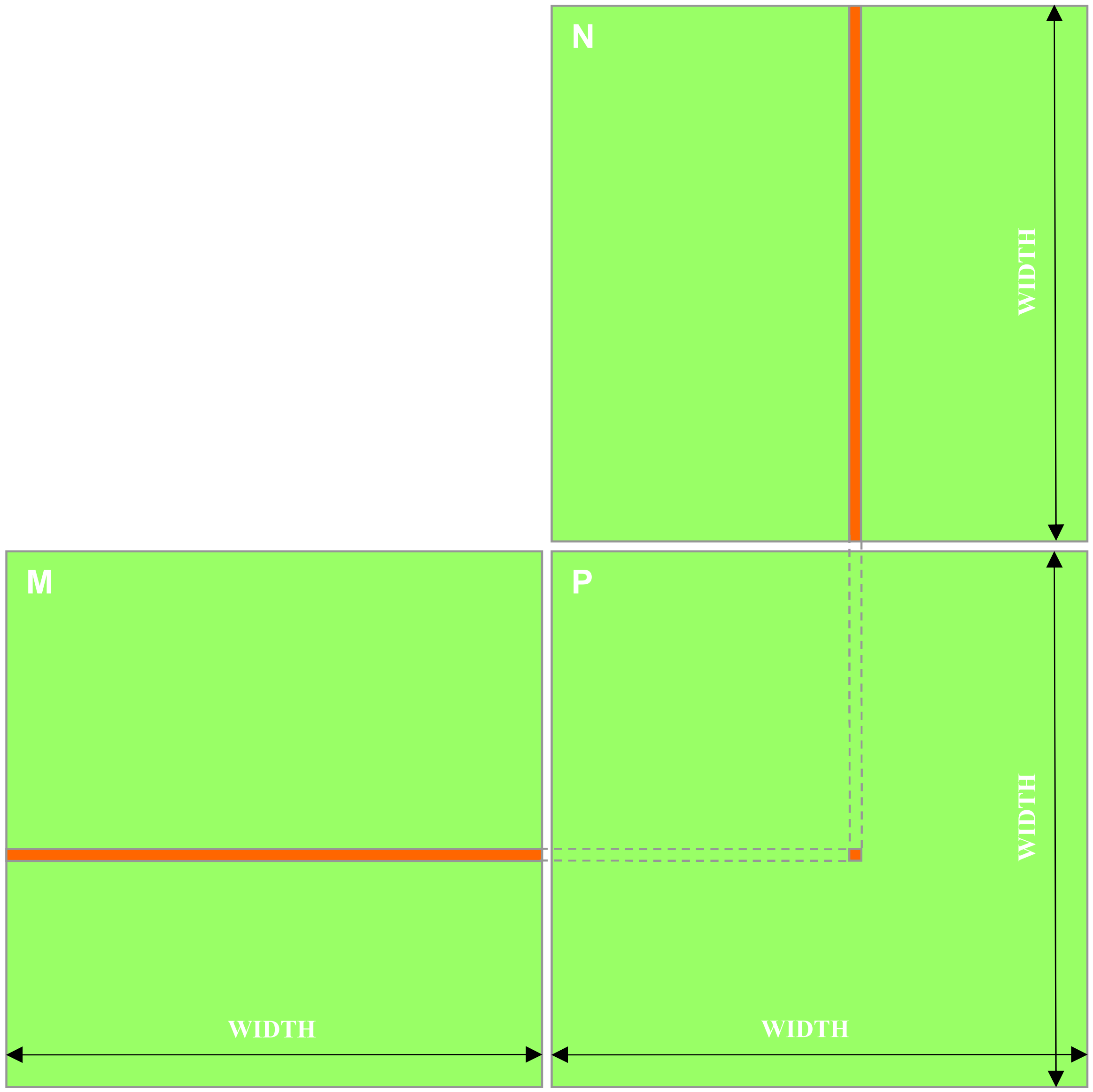
	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

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

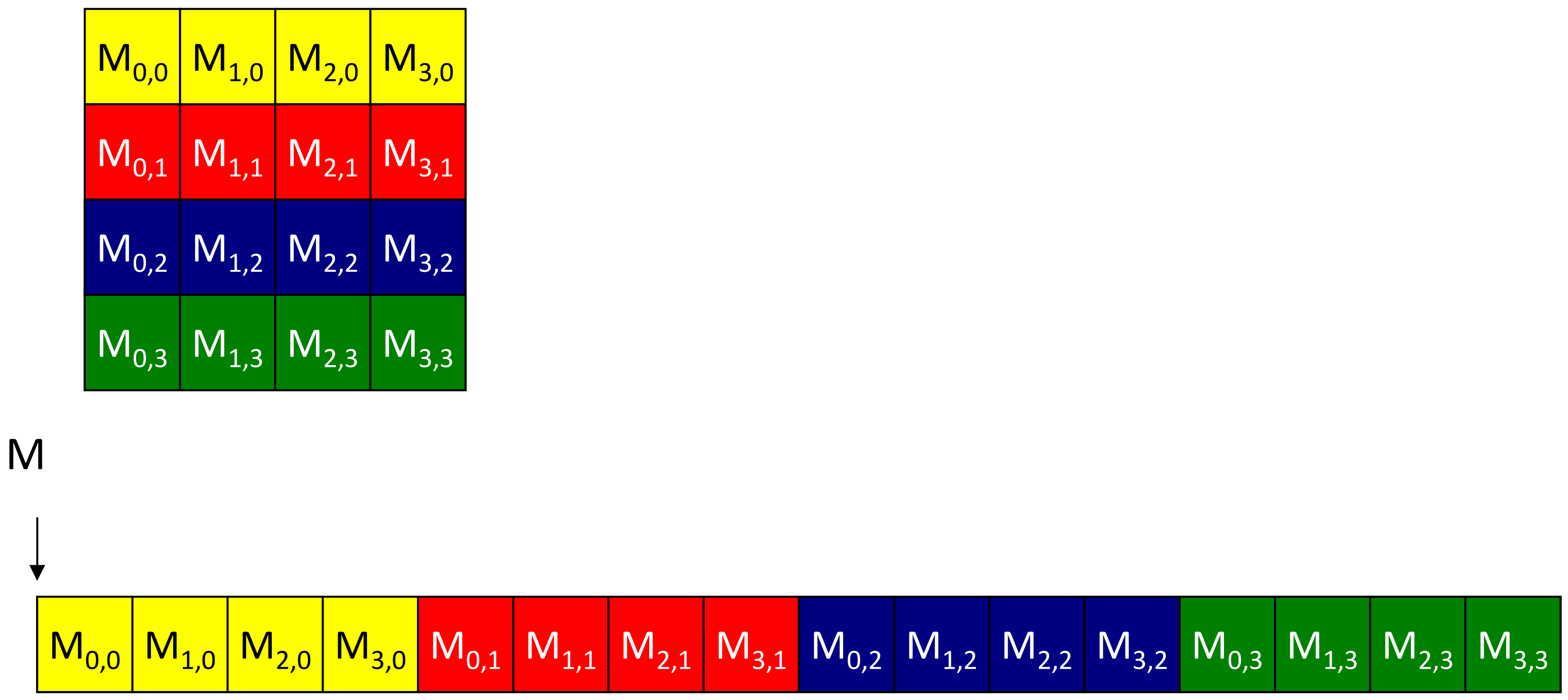
- For functions executed on the device:
  - No static variable declarations inside the function
  - No indirect function calls through pointers

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 row-major placement approach
when storing 2D matrix in linear memory address.

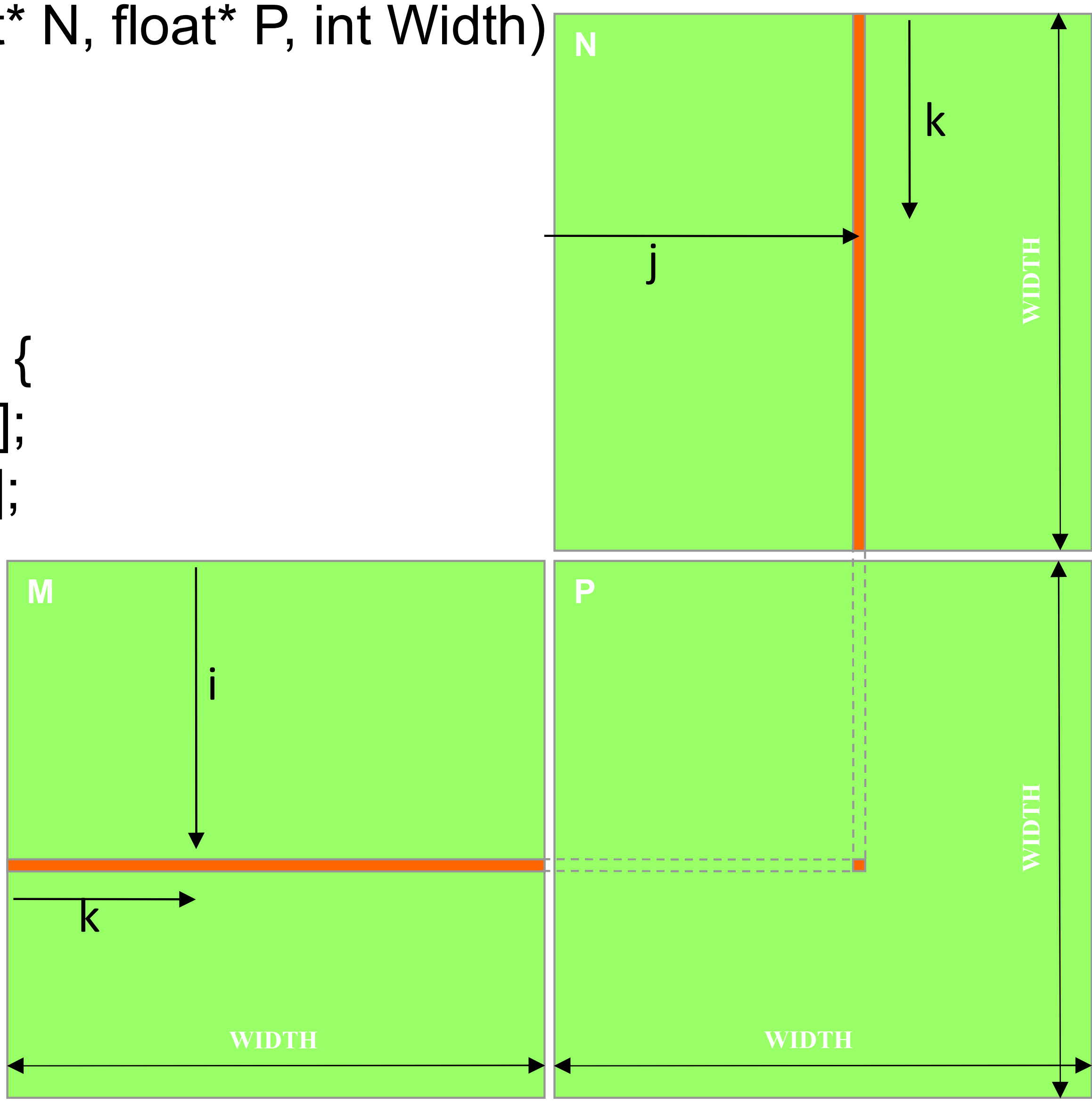
The *Hello World* of Parallel Programming: **Matrix Multiplication**

```
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;
        }
}
```

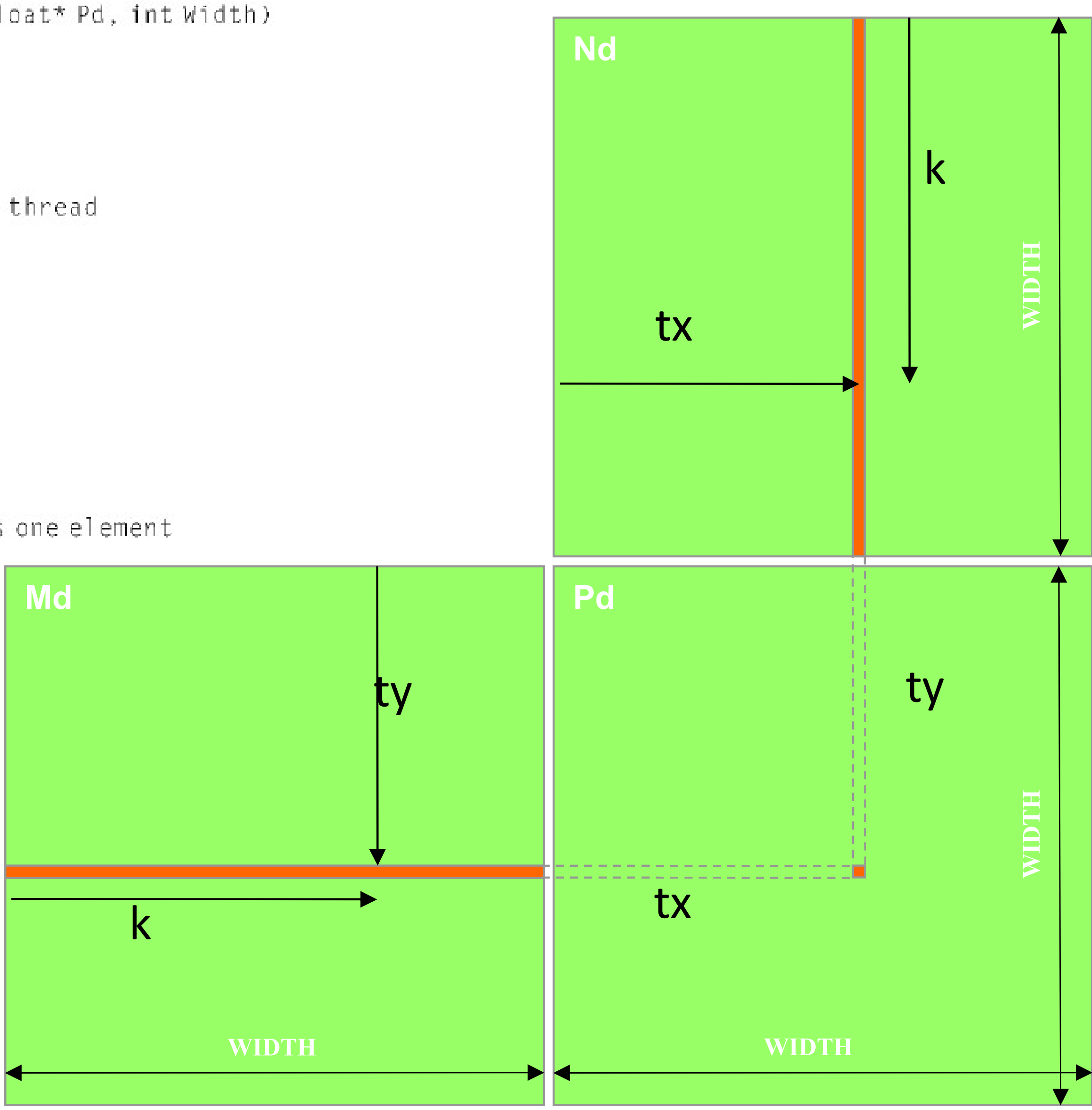


The *Hello World* of Parallel Programming: **Matrix Multiplication**

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

1. // 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

More On Specifying Dimensions

```
// Setup the execution configuration
   dim3 dimGrid(x, y, z);
   dim3 dimBlock(x, y, z);

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

Important:

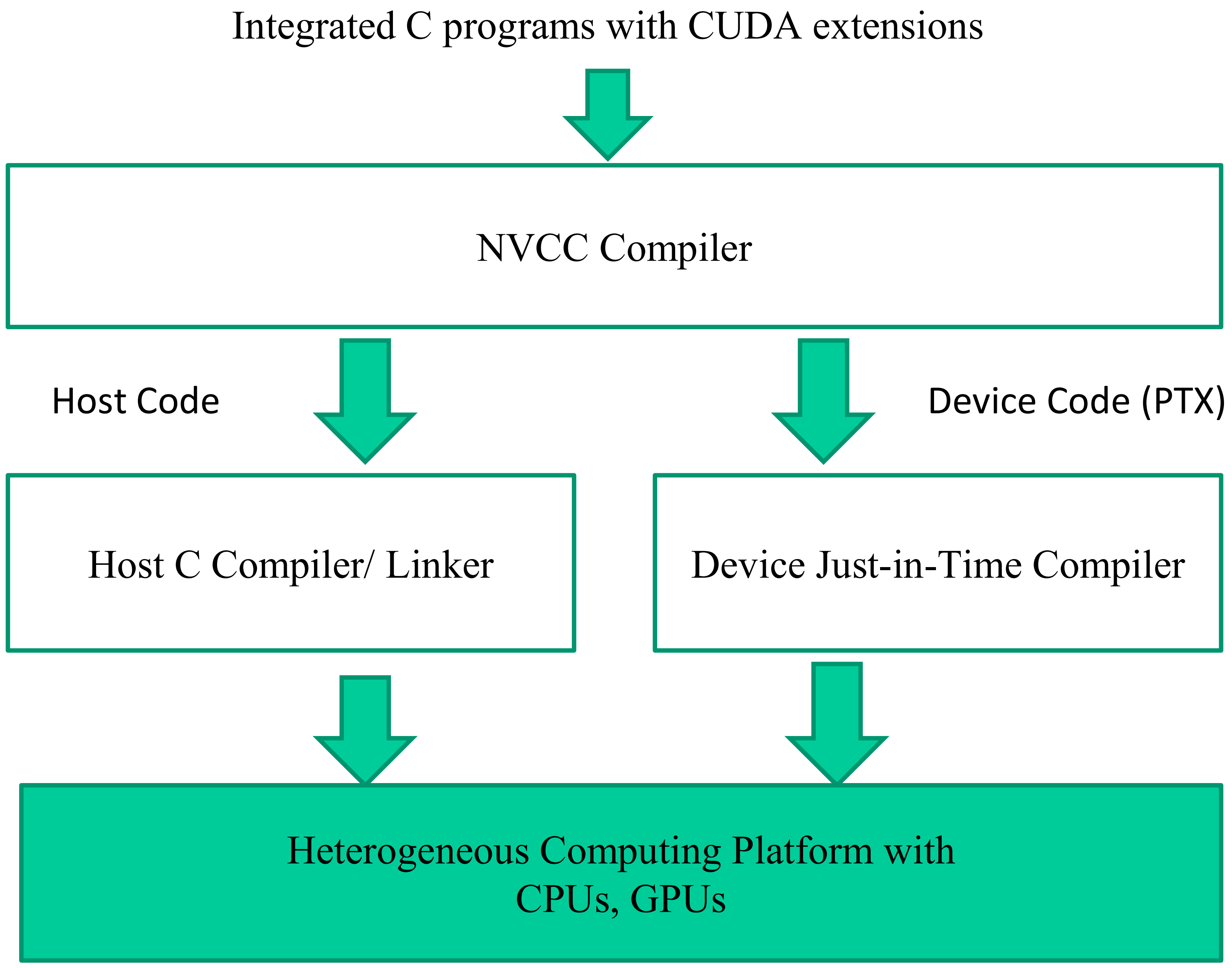
- dimGrid and dimBlock are user defined
- **gridDim** and **blockDim** are built-in predefined variable accessible in kernel functions

Be Sure To Know:

- Maximum dimensions of a block
- Maximum number of threads per block
- Maximum dimensions of a grid
- Maximum number of blocks per grid

Tools: Compiler: NVCC

nvcc -o prog prog.cu


Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C Compiler/ Linker
Device Just-in-Time Compiler
Heterogeneous Computing Platform with CPUs, GPUs

Conclusions

- Data parallelism is the main source of scalability for parallel programs
- Each CUDA source file can have a mixture of both host and device code.
- What we learned today about CUDA:
  - KernelA<<< nBlk, nTid >>>(args)
  - cudaMalloc()
  - cudaFree()
  - cudaMemcpy()
  - gridDim and blockDim
  - threadIdx and blockIdx
  - dim3