CUDA

Parallelism

- Writing a parallel program must always start by identifying the parallelism inherent in the algorithm at hand
- Different variants of parallelism induce different methods of parallelization

Parallelism

- Instruction-level parallelism (ILP) is the parallel or simultaneous execution of a sequence of instructions in a computer program.
- More specifically ILP refers to the average number of instructions run per step of this parallel execution.
- **Instruction-level parallelism** (ILP) is a measure of how many operations in a computer program can be performed "in-parallel" at the same time

Caution:

- Data dependency : Data dependence means that one instruction is dependent on another if there exists a chain of dependencies between them
- Name Dependency: A me dependency occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between them.

Parallelism

- **Data Level parallelism**
  - Many problems in scientific computing involve processing of large quantities of data stored on a computer
  - If this manipulation can be performed in parallel, i.e., by multiple processors working on different parts of the data, we speak of **data parallelism**
  - This is the dominant parallelization concept in scientific computing on MIMD-type computers
  - The same code is executed on all processors, with independent instruction pointers

Parallelism

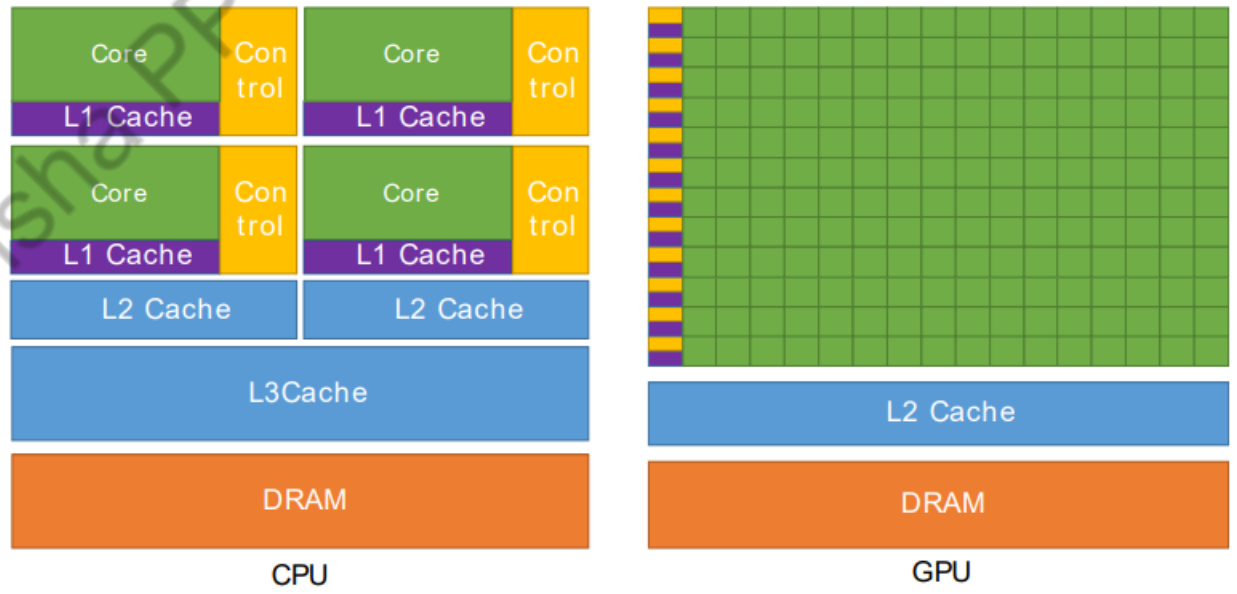
- **Functional/Task level parallelism**
  - Sometimes the solution of a “big” numerical problem can be split into separate subtasks
  - Which work together by **data exchange** and **synchronization**
  - In this case, the subtasks execute completely different code on different data items, which is why functional parallelism is also called MPMD
  - Functional parallelism bears pros and cons:
    - When different parts of the problem have different performance properties and hardware requirements**, bottlenecks and load imbalance** can easily arise
    - On the other hand, overlapping tasks that would otherwise be executed sequentially could accelerate execution considerably

Introduction

- The **Graphics Processing Unit (GPU)** provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope
- Many applications leverage these higher capabilities to run faster on the GPU than on the CPU
- This difference in capabilities between the **GPU** and the **CPU** exists because they are designed with different goals in mind
  - The CPU is designed to excel at executing a **sequence of operations**, called a **thread**, as fast as possible and can execute a few tens of these threads in parallel
  - The GPU is designed to excel at executing **thousands of them in parallel**
  - The GPU is specialized for highly parallel computations
  - Designed such that more transistors are devoted to data processing rather than data caching and flow control

Introduction

- Devoting more transistors to data processing is beneficial for high parallel computing
  - e.g: floating-point computations
- The GPU can hide memory access latencies with computation, instead of relying on large data caches and complex flow control
- Avoids long memory access latencies, both of which are expensive in terms of transistors



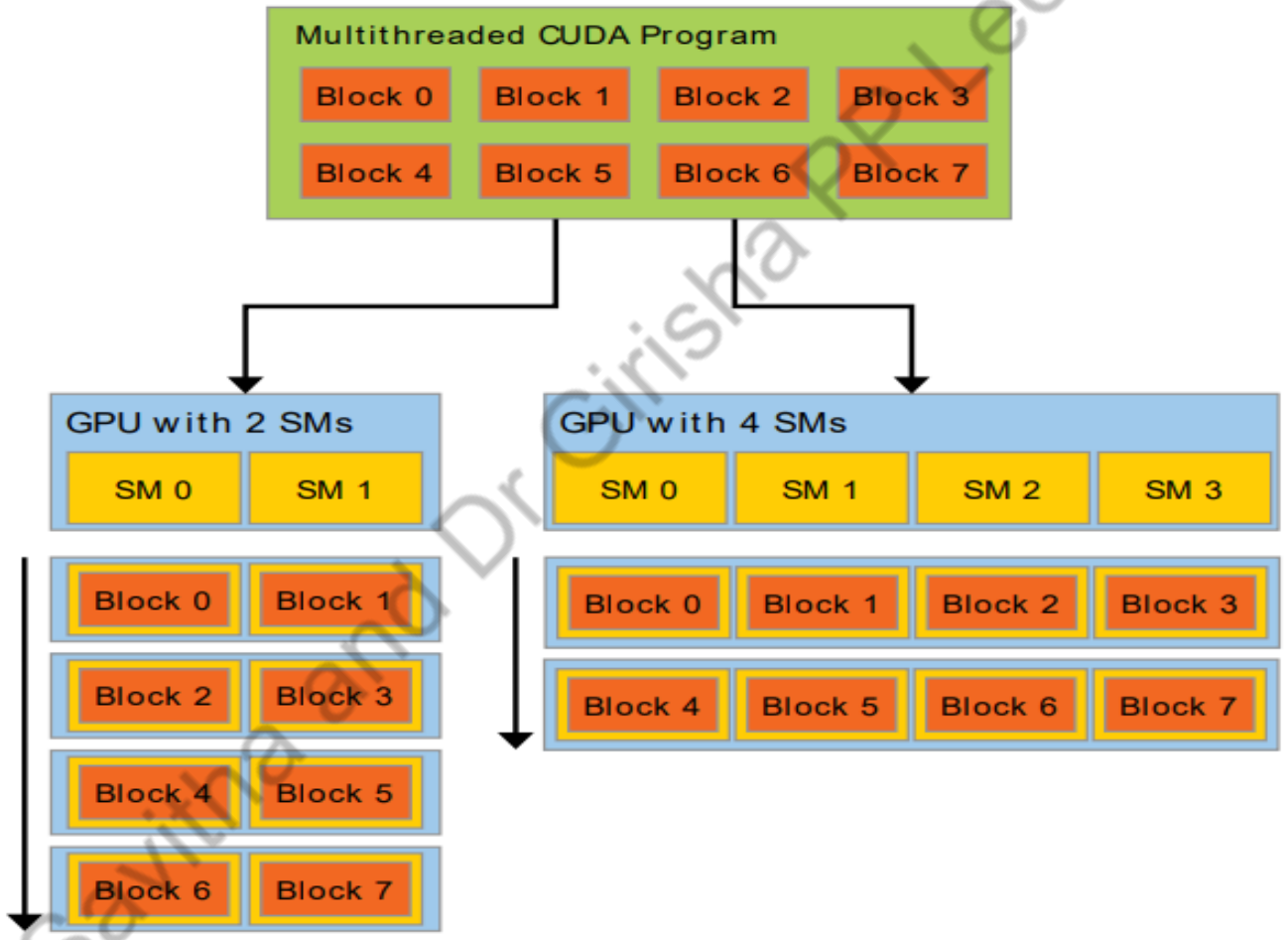
A Scalable Programming Model

- Mainstream processor chips are now parallel systems
  - Multicore CPUs and manycore GPUs
- The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores
- CUDA parallel programing at its core are three key abstractions:
  - A hierarchy of thread groups
  - Shared memories
  - Barrier synchronization
- That are simply exposed to the programmer as a minimal set of language extensions

A Scalable Programming Model

- They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads
- Each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block
- Each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors
- Only the runtime system needs to know the physical multiprocessor count

A Scalable Programming Model

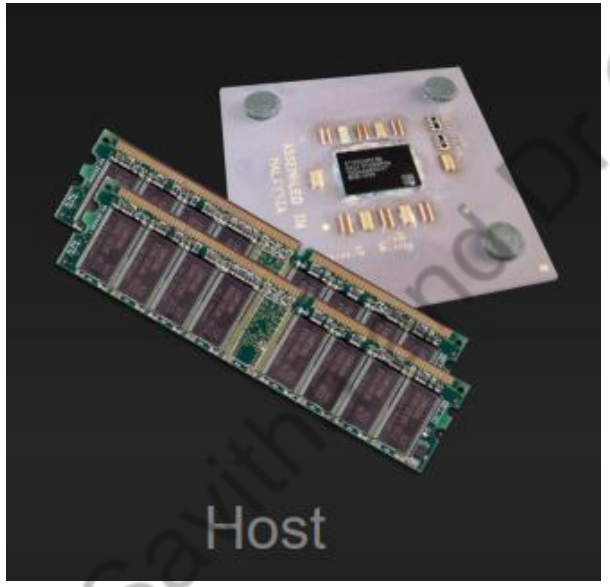


Development environment

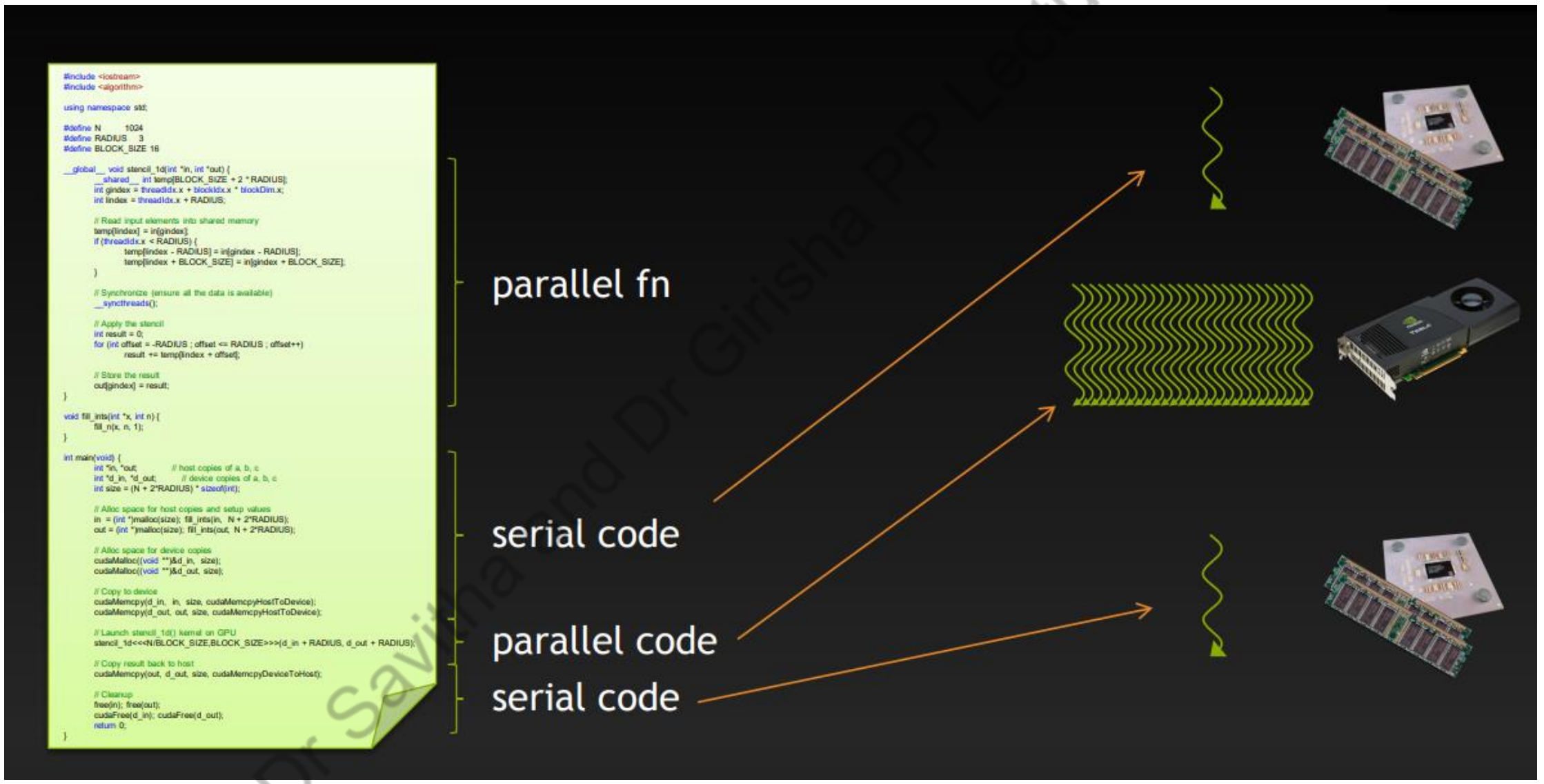
- Every NVIDIA GPU since the 2006 release of the GeForce 8800 GTX has been CUDA-enabled
  - Has been built on the CUDA Architecture
- NVIDIA DEVICE DRIVER
  - NVIDIA provides system software that allows your programs to communicate with the CUDA-enabled hardware
- CUDA Development Toolkit
  - CUDA C applications are going to be computing on two different processors and we need two compilers
  - One compiler will compile code for your GPU, and one will compile code for your CPU
  - NVIDIA provides the compiler for your GPU code (CUDA Toolkit)

Heterogeneous Computing

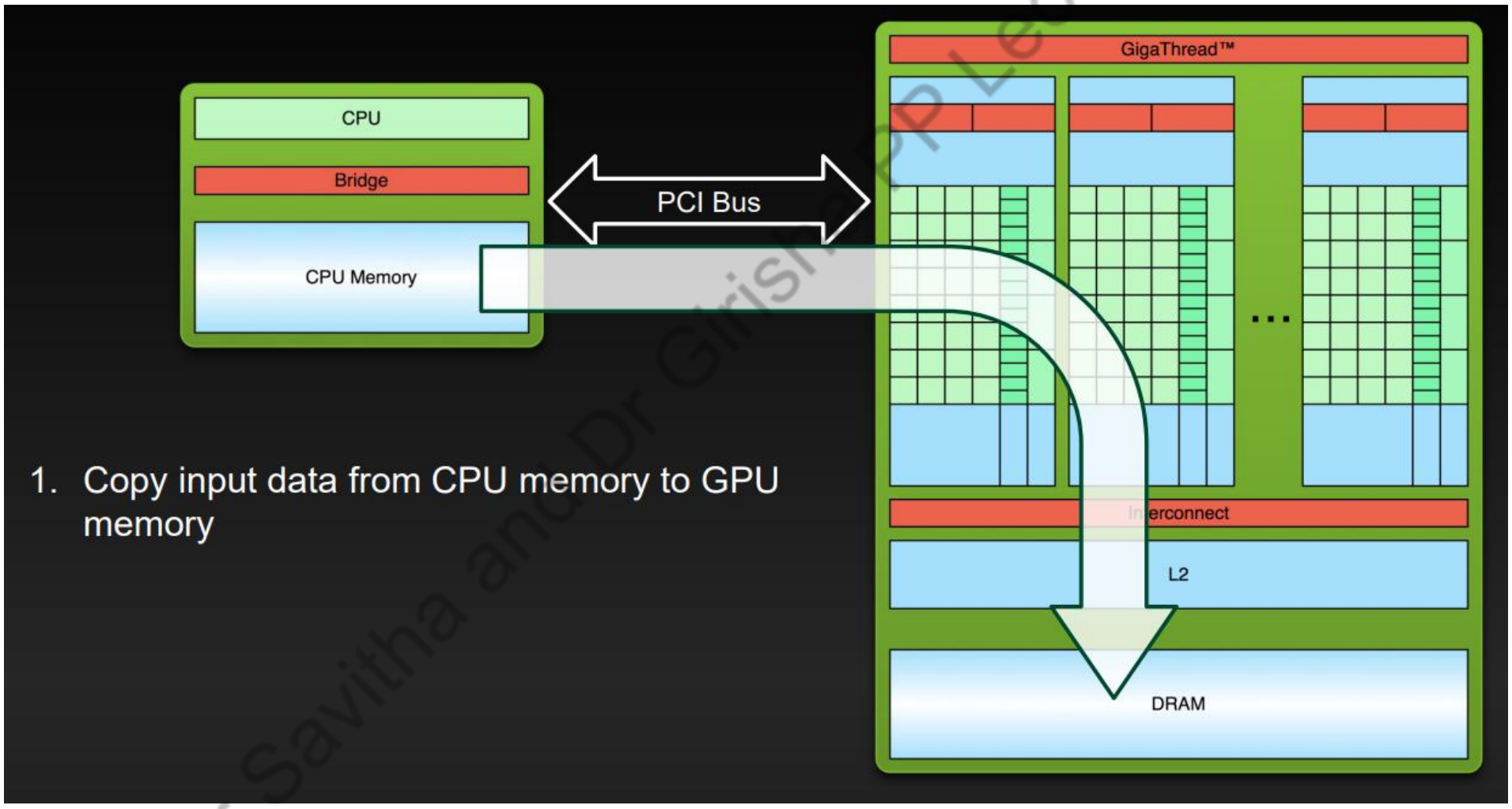
- Terminology:
  - Host: The CPU and its memory (host memory)
  - Device: The GPU and its memory (device memory)



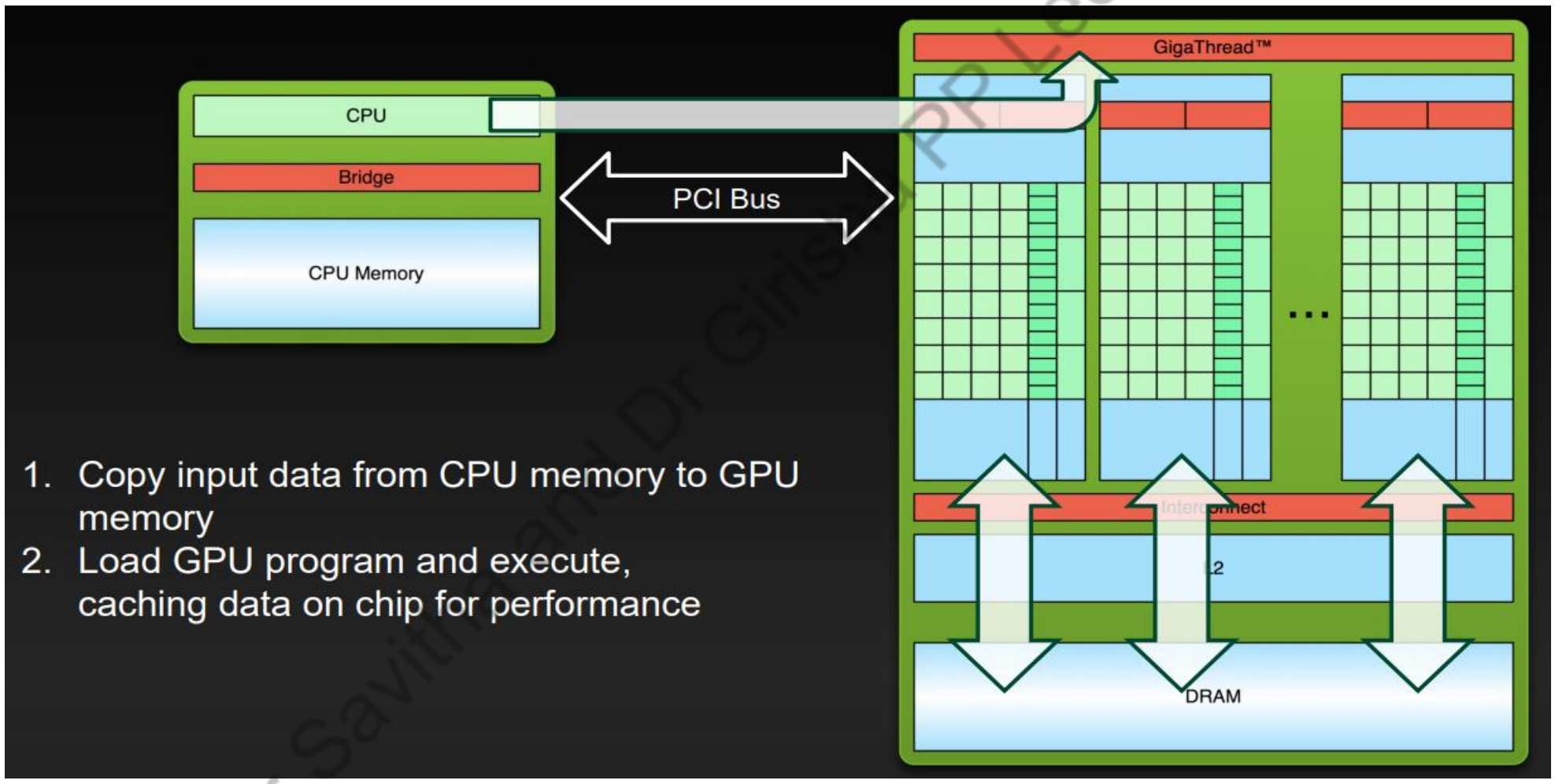
Heterogeneous Computing



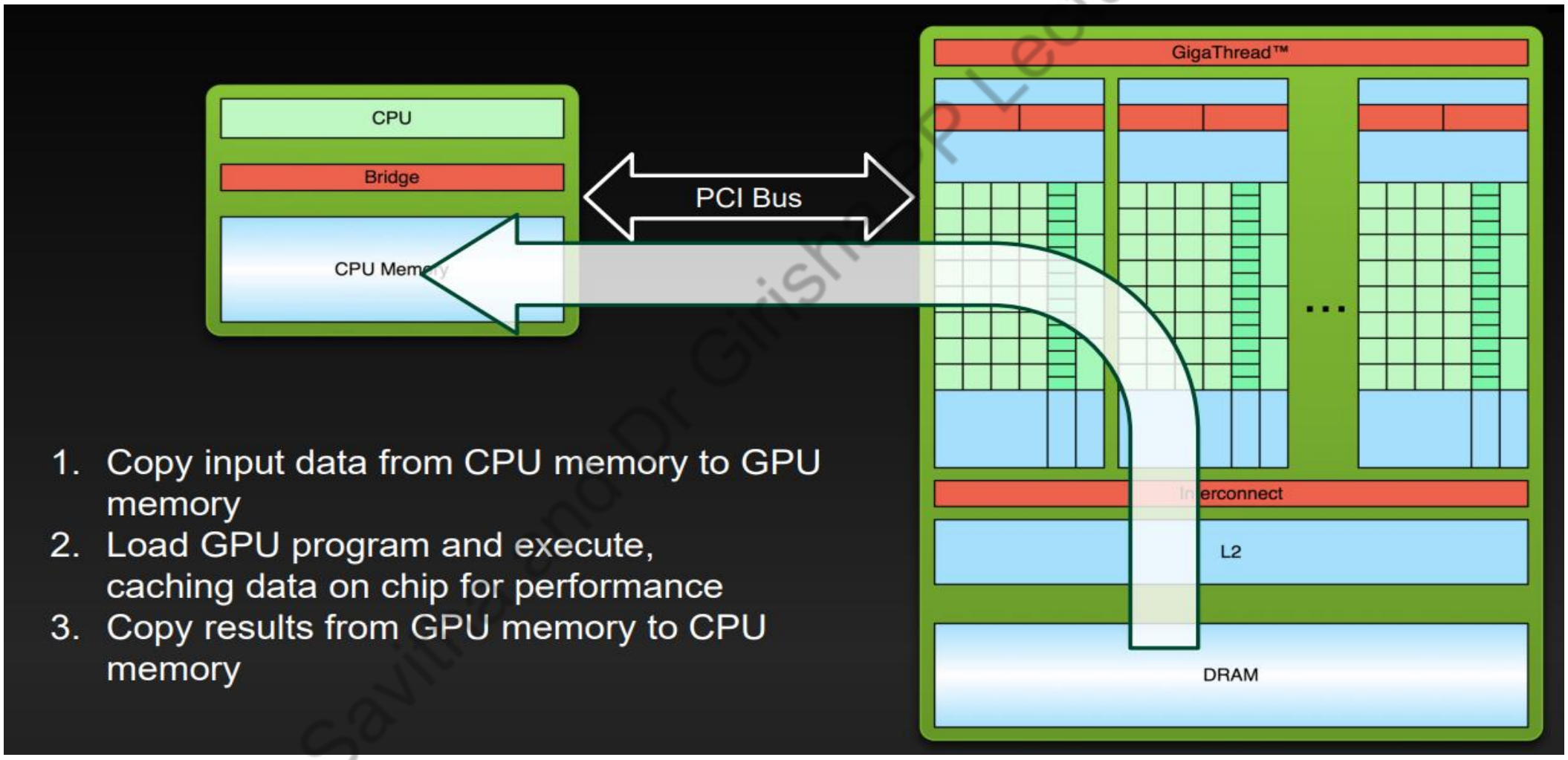
Process flow



Process flow



Process flow



Hello world example

- The __global__ specifier indicates a function that runs on device (GPU)
- Such function can be called through host code, e.g. the main() function in the example, and is also known as "kernels“
- When a kernel is called, its execution configuration is provided through <<<...>>>. This is called kernel launch

- This is a simple CUDA C program with two distinctions:
  - An empty function named `kernel()` qualified with `__global__`
  - A call to the empty function, embellished with `<<<1,1>>>`

```
#include <iostream>

__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- Code is compiled by our system’s standard C compiler by default
- **_ _global_ _** qualifier:This mechanism alerts the compiler that a function should be compiled to run on a device instead of the host
- In this simple example, **nvcc** gives the function **kernel()** to the compiler that handles device code
- And it feeds **main()** to the host compiler

Kernels

- CUDA C/C++ extends C/C++ by allowing the programmer to define C/C++ functions, called **kernels**
- When called, they are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C++ functions
- A kernel is defined using the **__global__** declaration specifier
- The number of CUDA threads that execute that kernel for a given kernel call is specified using a new **<<< >>>** execution configuration syntax
- Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through built-in variables

Kernels

- The following sample code, using the built-in variable **threadIdx**, adds two vectors A and B of size N and stores the result into vector C
- Each of the N threads that execute **VecAdd()** performs one pair-wise addition

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Thread Hierarchy

- **threadIdx** is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index
- Forming a one dimensional, two-dimensional, or three-dimensional block of threads, called a **thread block**

- The index of a thread and its thread ID relate to each other in a straightforward way

Thread Hierarchy

- **Dx** defines the dimension of the block
- For a one-dimensional block, they are the same;
- For a two-dimensional block of size **(Dx , Dy),**the thread ID of a thread of index **(x, y)** is **(x + y Dx)**
- For a three-dimensional block of size (**Dx , Dy , Dz**), the thread ID of a thread of index **(x, y, z) is (x + y Dx + z Dx Dy)**

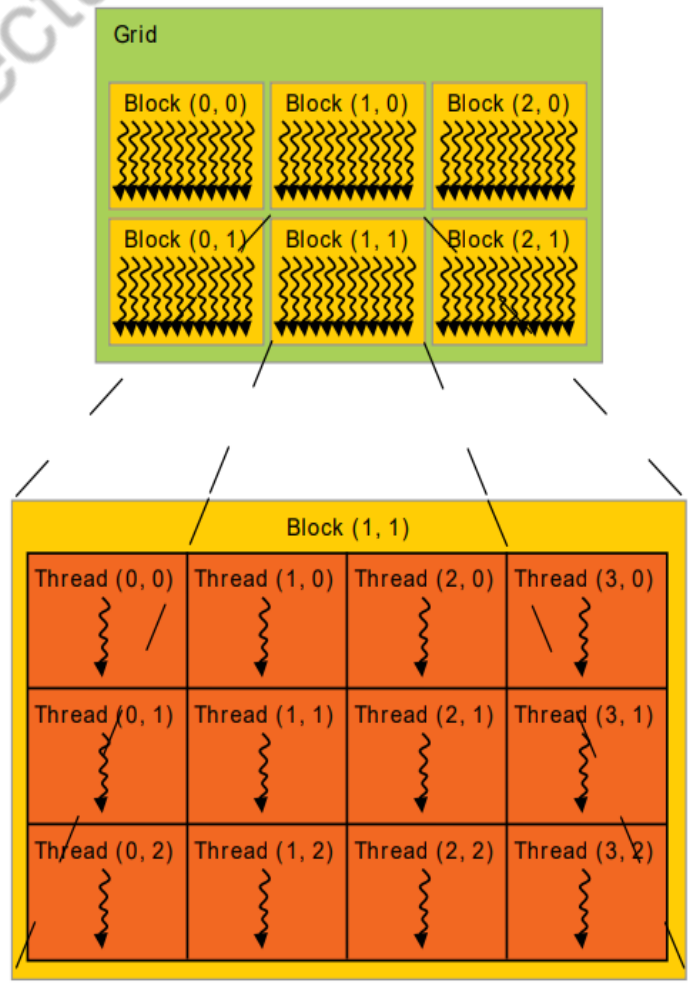
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread Hierarchy

- There is a limit to the number of threads per block
- Since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core
- On current GPUs, a thread block may contain up to 1024 threads
- However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks
- Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks

Thread Hierarchy

- The number of threads per block and the number of blocks per grid is specified in the **<<<>>>** syntax
- Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional unique index accessible within the kernel through the built-in **blockIdx** variable
- The dimension of the thread block is accessible within the kernel through the built-in **blockDim** variable



Thread Hierarchy

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread Hierarchy

- A thread block size of 16x16 (256 threads)
- The grid is created with enough blocks to have one thread per matrix element as before
- The number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension
- Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series
- This independence requirement allows thread blocks to be scheduled in any order across any number of cores
- Enabling programmers to write code that scales with the number of cores

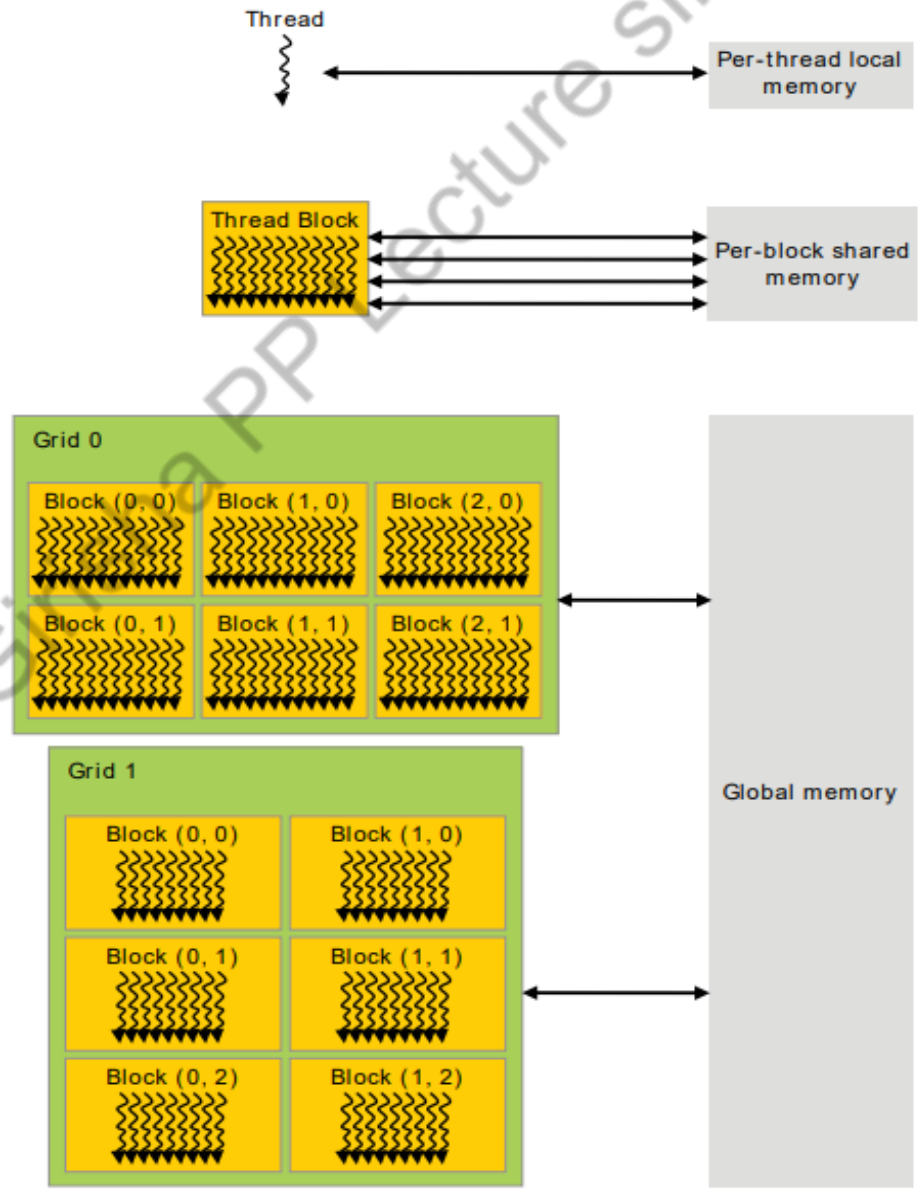
Thread Hierarchy

- Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses

Memory Hierarchy

- CUDA threads may access data from multiple memory spaces during their execution
- Each thread has private local memory
- Each thread block has **shared memory** visible to all threads of the block and with the same lifetime as the block
- All threads have access to the same **global memory**
- There are also two additional read-only memory spaces accessible by all threads: the **constant** and **texture memory** spaces
- The global, constant, and texture memory spaces are optimized for different memory usages
- The global, constant, and texture memory spaces are persistent across kernel launches by the same application

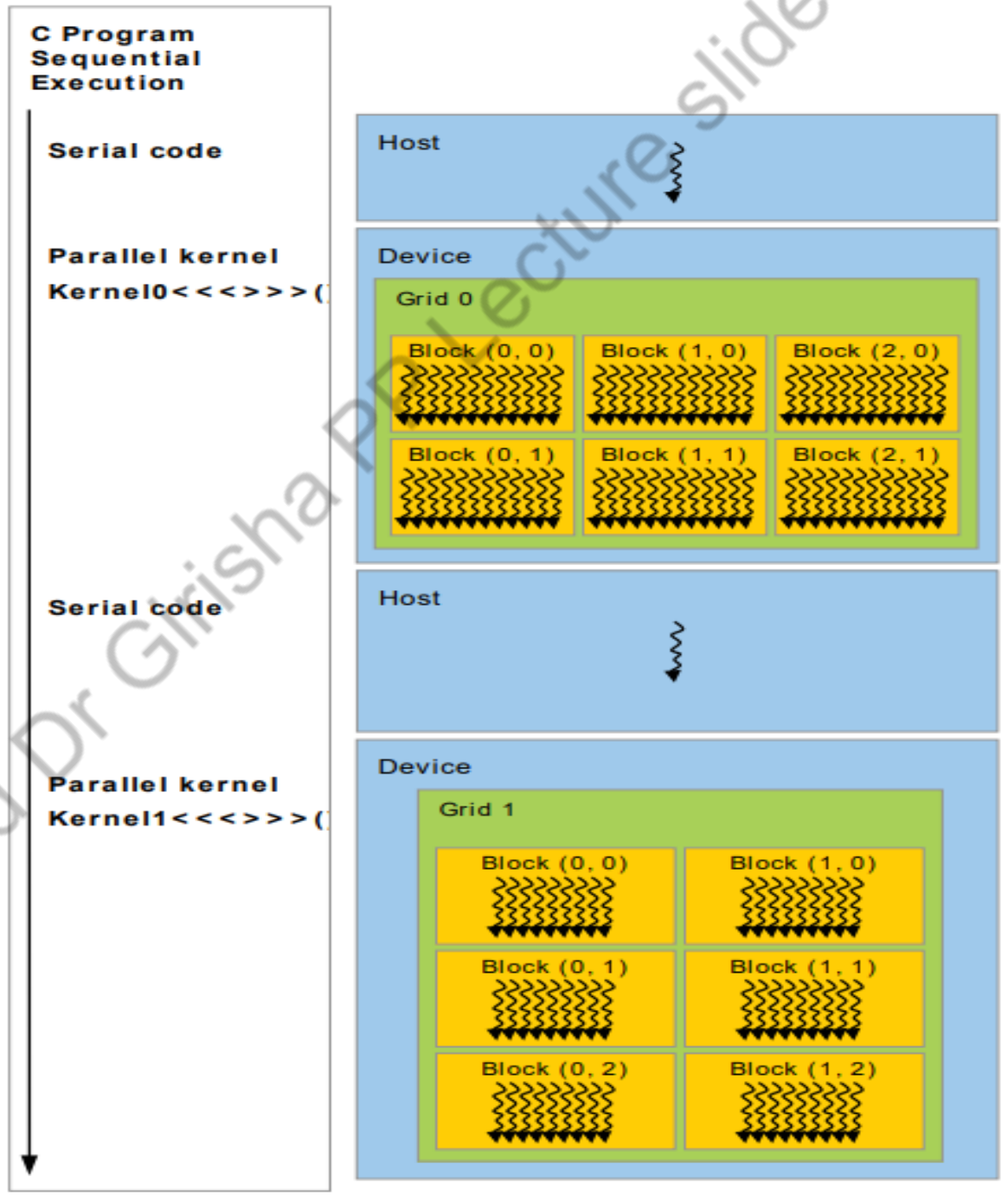
Memory Hierarchy



Heterogeneous Programming

- The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C++ program
- The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory
- Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime
- This includes device memory allocation and deallocation as well as data transfer between host and device memory

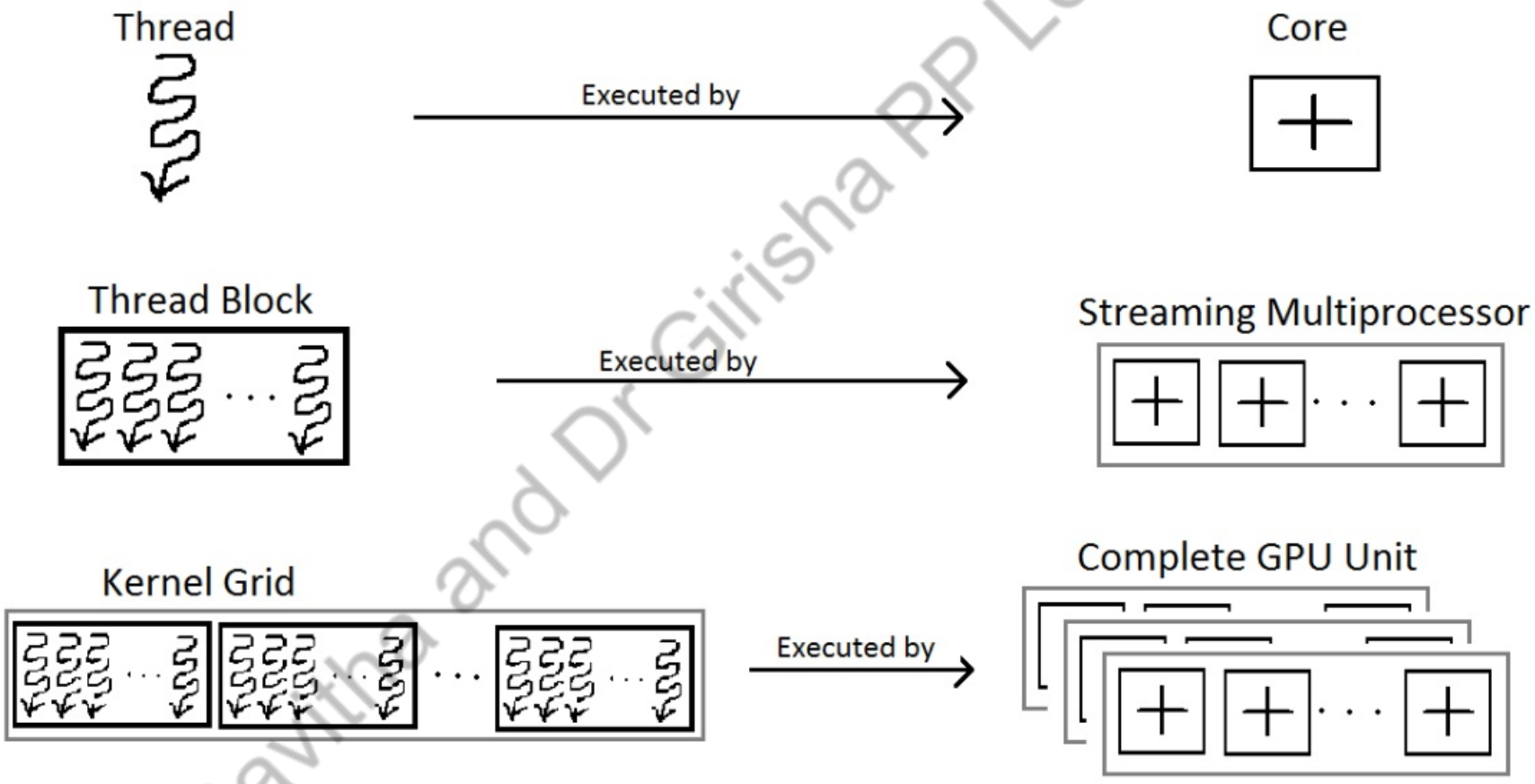
Heterogeneous Programming



Streaming Multiprocessors

- Hardware groups several threads that execute the same instructions into **wraps**
- Several wraps constitute a thread block
- Several thread blocks are assigned to a Streaming Multiprocessors (SM)
- Several SM constitute a GPU
- SM: These are general purpose processors with low clock rate and small cache
- The primary task of an SM is that it must execute several thread blocks in parallel
- As soon as one of its thread block has completed execution, it takes up the serially next thread block

Streaming Multiprocessors



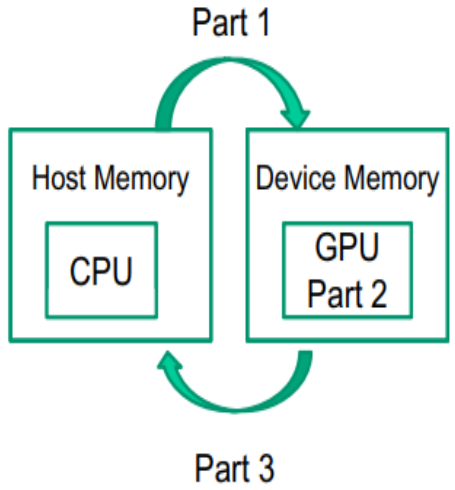
Streaming Multiprocessors

1. **Execution cores:** single precision floating-point units, double precision floating-point units, special function units (SFUs)
2. **Caches**
    1. L1 cache. (for reducing memory access latency).
    2. Shared memory (for shared data between threads).
    3. Constant cache (for broadcasting of reads from a read-only memory).
    4. Texture cache (for aggregating bandwidth from texture memory).
3. **Schedulers for warps**: these are for issuing instructions to warps based on a particular scheduling policies
4. **A substantial number of registers**: an SM may be running a large number of active threads at a time, so it is a must to have registers in thousands

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- Host and devices have separate memory spaces
- To execute a kernel on a device, the programmer needs to allocate global memory (device memory) on the device and transfer pertinent data from the host memory to the allocated device memory
- After device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed.

```
#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
}
```



DEVICE GLOBAL MEMORY AND DATA TRANSFER

- The CUDA runtime system provides API functions for managing data in the device memory

  - cudaMalloc()
    - Allocates object in the device global memory
    - Two parameters
      - **Address of a pointer** to the allocated object
      - **Size** of allocated object in terms of bytes
  - cudaFree()
    - Frees object from device global memoryv
      - **Pointer** to freed object

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- Function **cudaMalloc()** can be called from the host code to allocate a piece of device global memory for an object
- The first parameter to the **cudaMalloc()** function is the **address of a pointer variable** that will be set to point to the allocated object
- The address of the pointer variable should be cast to **(void)** because the function expects a generic pointer
- This parameter allows the cudaMalloc() function to write the address of the allocated memory into the pointer variable
- The host code passes this pointer value to the kernels that need to access the allocated memory
- The second parameter to the cudaMalloc() function gives the size of the data to be allocated, in terms of bytes

DEVICE GLOBAL MEMORY AND DATA TRANSFER

```
float *d_A
int size = n * sizeof(float);
cudaMalloc((void**)&d_A, size);
...
cudaFree(d_A);
```

After the computation, cudaFree() is called with pointer **d_A** as input to free the storage space

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- Once the host code has allocated device memory for the data objects, it can request that data be transferred from host to device
- This is accomplished by calling one of the CUDA API functions **cudaMemcpy()**
- The **cudaMemcpy()** function takes four parameters
- A pointer to the **destination location** for the data object to be copied
- The second parameter points to the **source location**
- The third parameter specifies the **number of bytes to be copied**
- The fourth parameter indicates the **types of memory involved** in the copy:
  - from host memory to host memory
  - from host memory to device memory
  - from device memory to host memory
  - from device memory to device memory

DEVICE GLOBAL MEMORY AND DATA TRANSFER

cudaMemcpy()

- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer

DEVICE GLOBAL MEMORY AND DATA TRANSFER

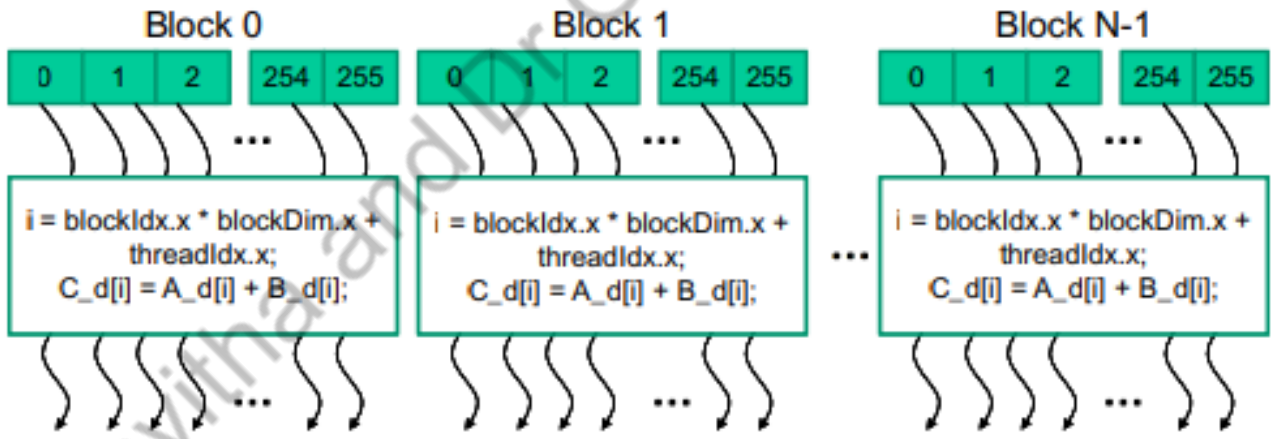
```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

KERNEL FUNCTIONS AND THREADING

- In CUDA, a **kernel function** specifies the code to be executed by all threads during a parallel phase by the device
- Since all these threads execute the same code, CUDA programming is an instance of the well-known **SPMD**
- When a host code launches a kernel, the CUDA runtime system generates a grid of threads that are organized in a **two-level hierarchy**
- Grid is organized into an array of thread blocks
- All blocks of a grid are of the same size; each block can contain up to 1,024 threads

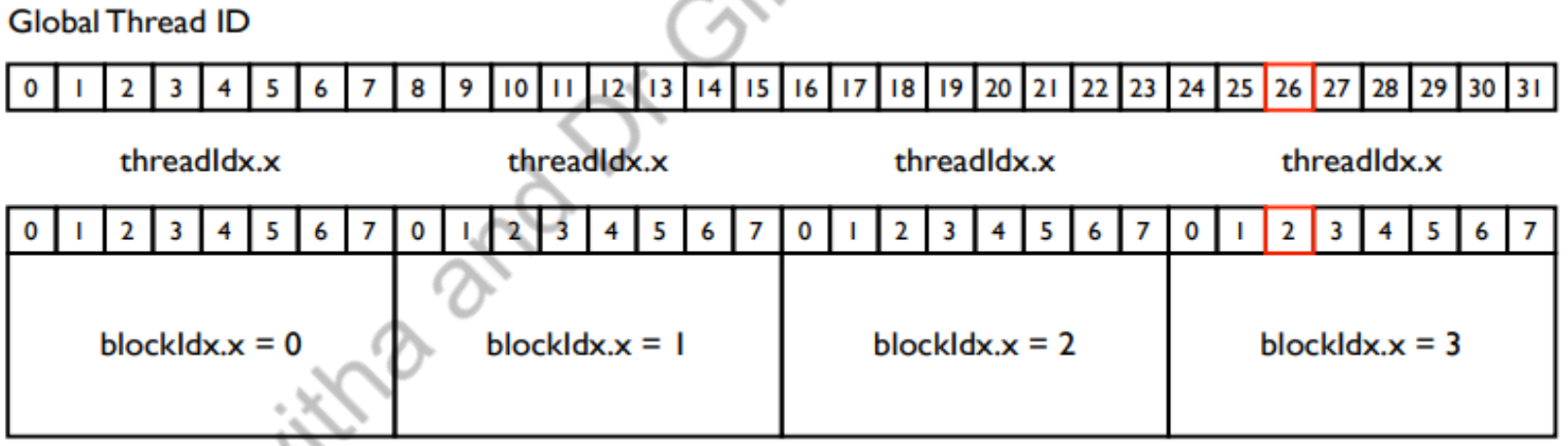
KERNEL FUNCTIONS AND THREADING

- The number of threads in each thread block is specified by the host code when a kernel is launched
- The same kernel can be launched with different numbers of threads at different parts of the host code
- For a given grid of threads, the number of threads in a block is available in the **blockDim** variable



KERNEL FUNCTIONS AND THREADING

- Each thread in a block has a unique **threadIdx** value
- This allows each thread to combine its **threadIdx** and **blockIdx** values to create a unique **global index** for itself with the entire grid



KERNEL FUNCTIONS AND THREADING

- By launching the kernel with a larger number of blocks, one can process larger vectors
- By launching a kernel with n or more threads, one can process vectors of length n

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

KERNEL FUNCTIONS AND THREADING

- **__global__**
  - keyword indicates that the function being declared is a CUDA kernel function
  - Function is to be executed on the device and can only be called from the host code
- **__device__**
  - keyword indicates that the function being declared is a CUDA device function
  - A device function executes on a CUDA device and can only be called from a kernel function or another device function
- **__host__**
  - Host function is simply a traditional C function that executes on the host and can only be called from another host function

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

KERNEL FUNCTIONS AND THREADING

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    vecAddKernel<<<ceil(n/2560), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
      // Free device memory for A, B, C
     cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

KERNEL FUNCTIONS AND THREADING

- The code is hardwired to use thread blocks of 256 threads each
- The number of thread blocks used, however, depends on the length of the vectors (n)
- If n is 750, three thread blocks will be used; if n is 4,000, 16 thread blocks will be used; if n is 2,000,000, 7,813 blocks will be use
- A small GPU with a small amount of execution resources may execute one or two of these thread blocks in parallel
- A larger GPU may execute 64 or 128 blocks in parallel

CUDA THREAD ORGANIZATION

- Threads are organized into a two-level hierarchy
  - **Grid**
  - **Blocks**
- All threads in a block share the same block index
- CUDA provides built-in, preinitialized variables that can be accessed within kernel functions
  - **blockIdx**
  - **threadIdx**
  - **gridDim**
  - **blockDim**

CUDA THREAD ORGANIZATION

- A grid is a 3D array of blocks and each block is a 3D array of thread
- The programmer can choose to use fewer dimensions by setting the unused dimensions to 1
- The exact organization of a grid is determined by the execution configuration parameters (within <<< and >>>) of the kernel launch statement
- The first execution configuration parameter specifies the dimensions of the grid in number of blocks. The second specifies the dimensions of each block in number of thread

CUDA THREAD ORGANIZATION

```
dim3 dimGrid(ceil(n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(...);
```

- This allows the number of blocks to vary with the size of the vectors so that the grid will have enough threads to cover all vector elements
- The value of variable n at kernel launch time will determine the dimension of the grid
- If n is equal to 1,000, the grid will consist of four blocks. If n is equal to 4,000, the grid will have 16 blocks

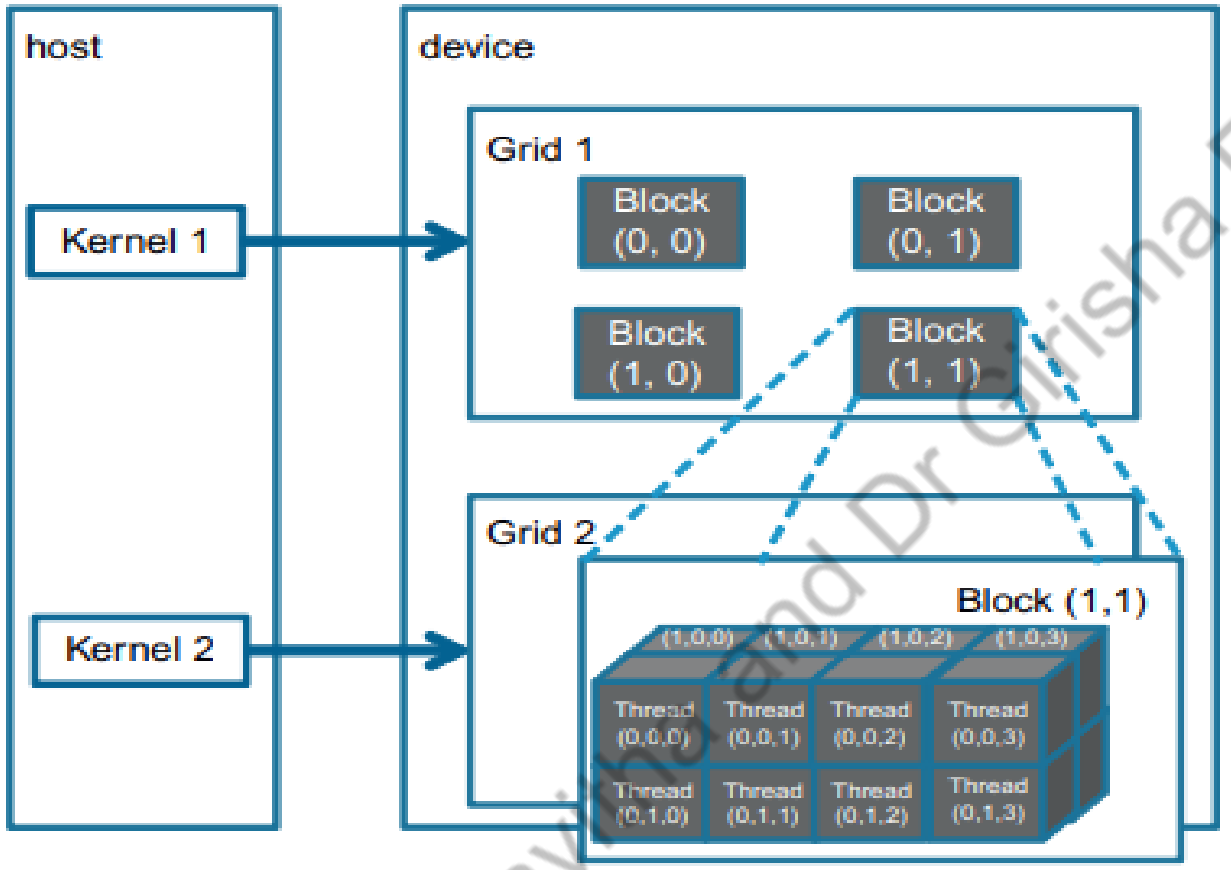
CUDA THREAD ORGANIZATION

- In CUDA C, the allowed values of gridDim.x, gridDim.y, and gridDim.z range from 1 to 65,536
- All threads in a block share the same blockIdx.x, blockIdx.y, and blockIdx.z values
- Among all blocks, the blockIdx.x value ranges between 0 and gridDim.x-1, the blockIdx.y value between 0 and gridDim.y-1, and the blockIdx.z value between 0 and gridDim.z-1.

CUDA THREAD ORGANIZATION

- Blocks are organized into 3D arrays of threads
- Two-dimensional blocks can be created by setting the z dimension to 1. One-dimensional blocks can be created by setting both the y and z dimensions to 1
- The number of threads in each dimension of a block is specified by the second execution configuration parameter at the kernel launch
- The total size of a block is limited to 1,024 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1,024
  - For example, (512, 1, 1), (8, 16, 4), and (32, 16, 2) are all allowable blockDim values, but (32, 32, 2) is not allowable since the total number of threads would exceed 1,024

CUDA THREAD ORGANIZATION




```
dim3 dimBlock(2, 2, 1);
dim3 dimGrid(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

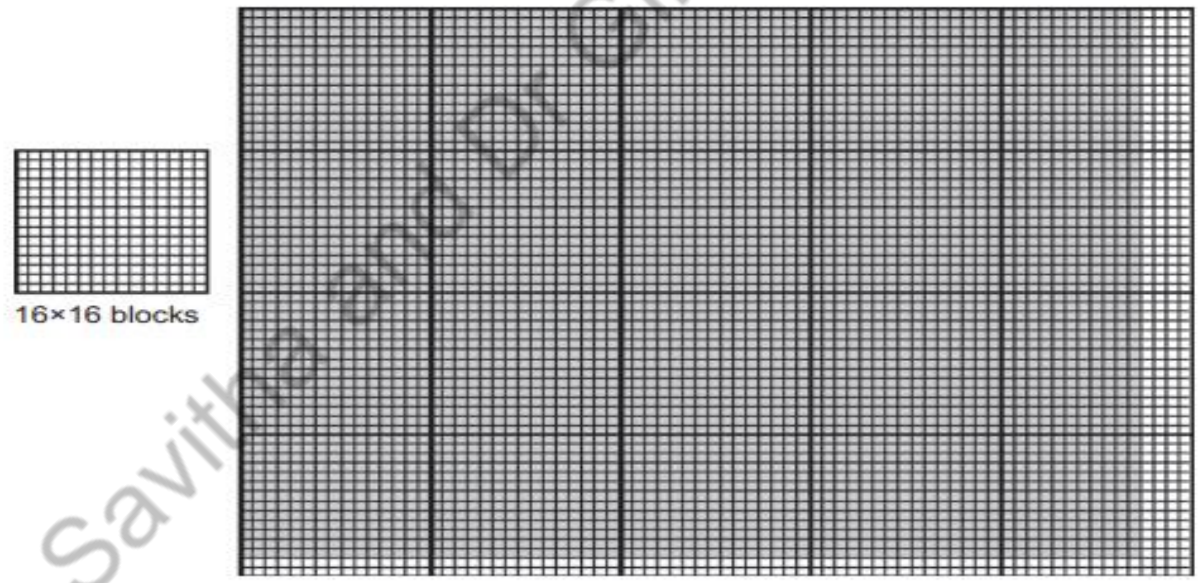
dimBlock, dimGrid

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data
- For example, pictures are a 2D array of pixels
- It is often convenient to use a 2D grid that consists of 2D blocks to process the pixels in a picture
- Consider a picture of 76 x 62 picture

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- Assume that we decided to use a 16 x 16 block, with 16 threads in the x direction and 16 threads in the y direction
- We will need five blocks in the x direction and four blocks in the y direction, which results in 5 x 4 = 20 blocks



MAPPING THREADS TO MULTIDIMENSIONAL DATA

- Assume that the host code uses an integer variable n to track the number of pixels in the x direction, and another integer variable m to track the number of pixels in the y direction
- We further assume that the input picture data has been copied to the **device memory** and can be accessed through a pointer variable **d_Pin**
- The output picture has been allocated in the device memory and can be accessed through a pointer variable **d_Pout**

```
dim3 dimBlock(ceil(n/16.0), ceil(m/16.0), 1);
dim3 dimGrid(16, 16, 1);
pictureKernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m);
```

MAPPING THREADS TO MULTIDIMENSIONAL DATA

```
__global__ void PictureKernell(float* d_Pin, float* d_Pout, int n, int m) {

  // Calculate the row # of the d_Pin and d_Pout element to process
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element to process
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < m) && (Col < n)) {
    d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
  }

}
```

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- We can easily extend our discussion of 2D arrays to 3D arrays
- This is done by placing each “plane” of the array one after another
- The programmer also needs to determine the values of blockDim.z and gridDim.z when launching a kernel
- In the kernel, the array index will involve another global index:

int Plane = blockIdx.z*blockDim.z + threadIdx.z

- One would of course need to test if all the three global indices—Plane, Row, and Col—fall within the valid range of the array

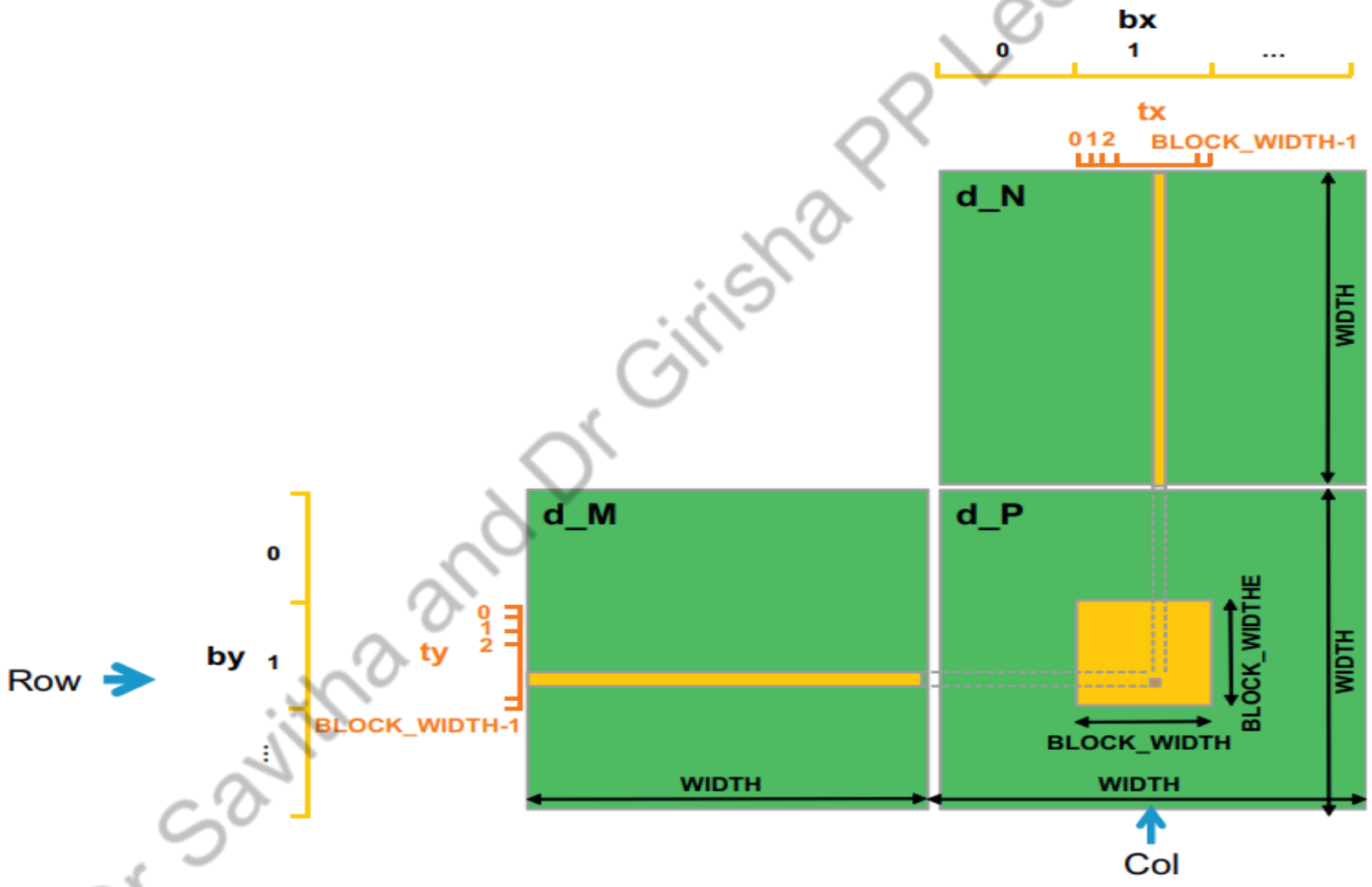
MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

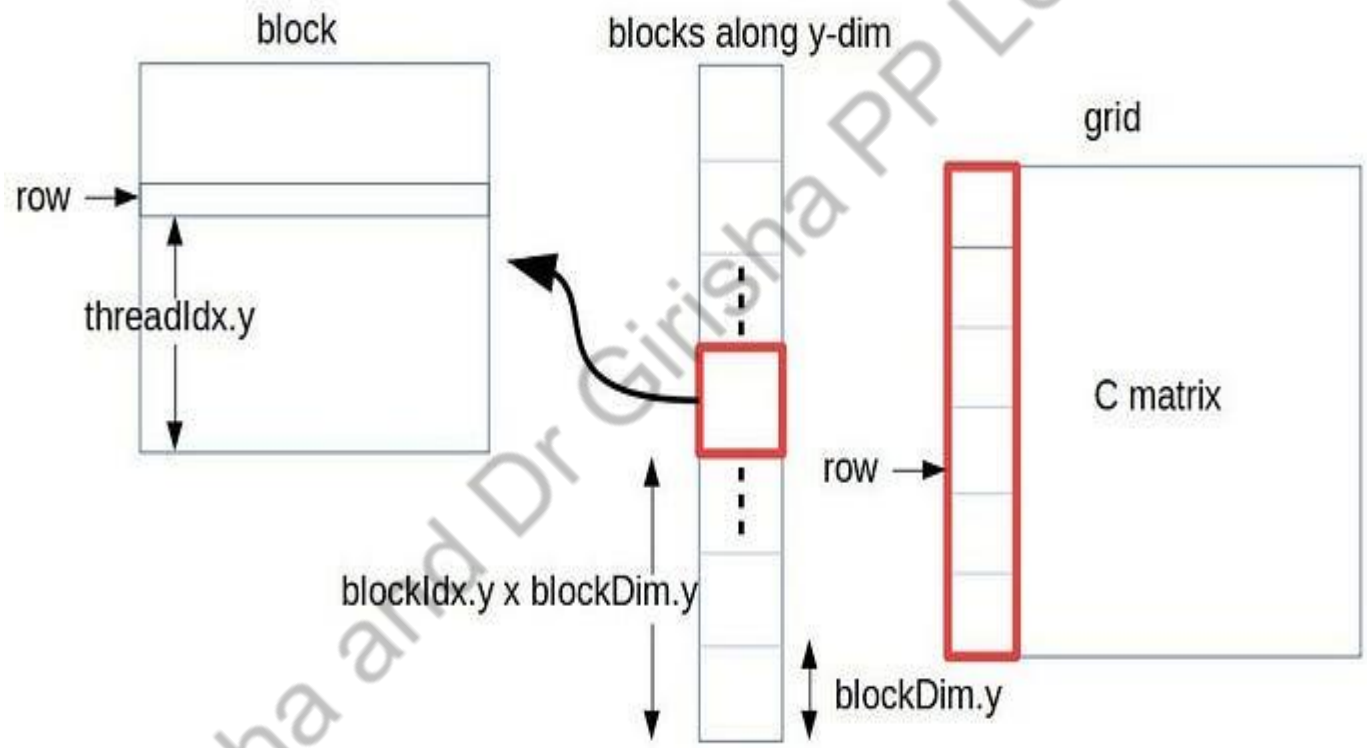
- We have studied **vecAddkernel()** and **pictureKernel()** where each thread performs only one floating-point arithmetic operation on one array element
  - two simple kernels were selected for teaching the mapping of threads to data using **threadIdx, blockIdx, blockDim**, and **gridDim** variables
  - the number of threads that we create is a multiple of the block dimension
  - As a result, we will likely end up with more threads than data elements
  - Not all threads will process elements of an array. We use an if statement to test if the global index values of a thread are within the valid range

MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- Matrix- matrix multiplication between an I x J matrix d_M and a J x K matrix d_N produces an I x K matrix d_P.
- When performing a matrix multiplication, each element of the product matrix d_P is an inner product of a row of d_M and a column of d_N
- The inner product between two vectors is the sum of products of corresponding elements

MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL





MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {

  // Calculate the row index of the d_Pelement and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (intk = 0; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }

}
```

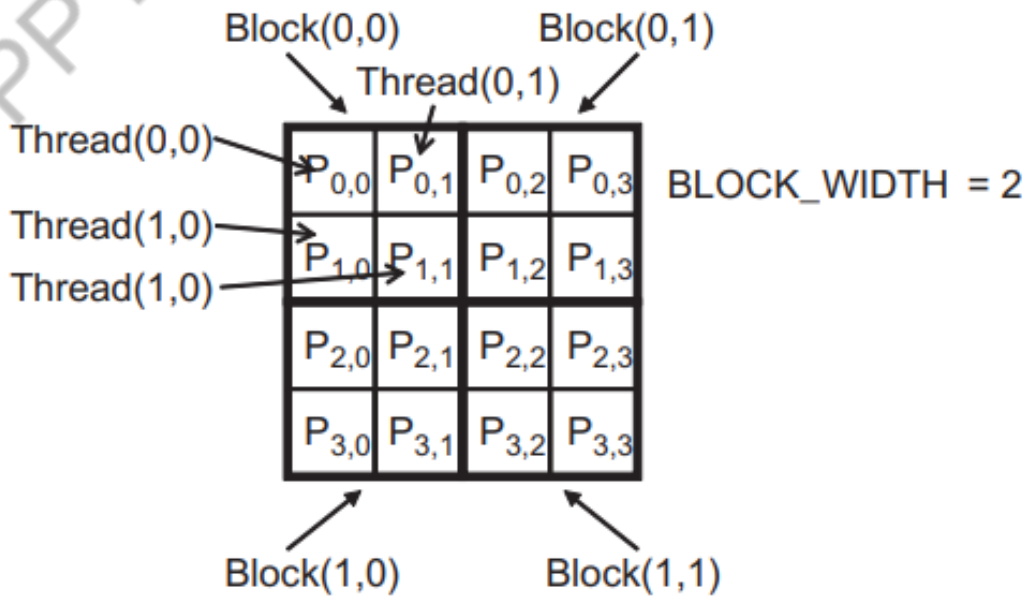
MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- Let us assume that, BLOCK_WIDTH=16
- Assume that we have a Width value of 1,000. That is, we need to do 1,000 x 1,000 matrix multiplication
- For a BLOCK_WIDTH value of 16, we will generate 16 x 16 blocks
- There will be 64 x 64 blocks in the grid to cover all d_P elements.

```
#define BLOCK_WIDTH 16

// Setup the execution configuration
    int NumBlocks = Width/BLOCK_WIDTH;
    if (Width % BLOCK_WIDTH) NumBlocks++;
    dim3 dimGrid(NumBlocks, NumbBlocks);
    dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

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

MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- For a BLOCK_WIDTH=32, there will be 32 x 32 blocks in the grid to cover all d_P elements
- Let us consider a smaller matrix of size 4x4
- BLOCK_WIDTH = 2
- The d_P matrix is now divided into four tiles and each block calculates one tile



MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- For the small matrix multiplication, threads in block(0,0) produce four dot products
- The Row and Col variables of thread(0,0) in block(0,0) are 0*0 + 0 = 0 and 0*0 + 0 =0
- It maps to P(0,0) and calculates the dot product of row 0 of M and column 0 of N