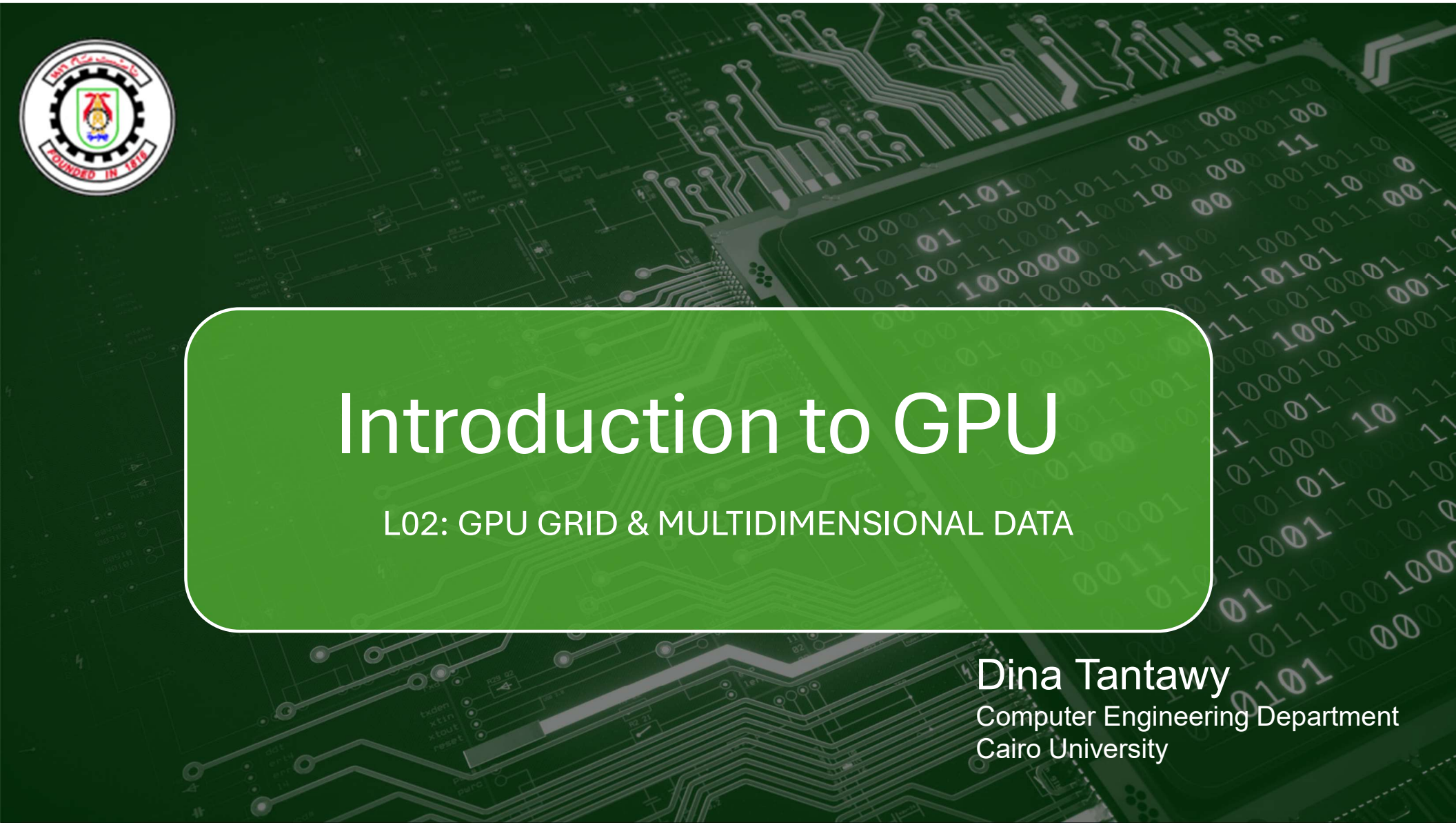


# Introduction to GPU

## L02: GPU GRID & MULTIDIMENSIONAL DATA

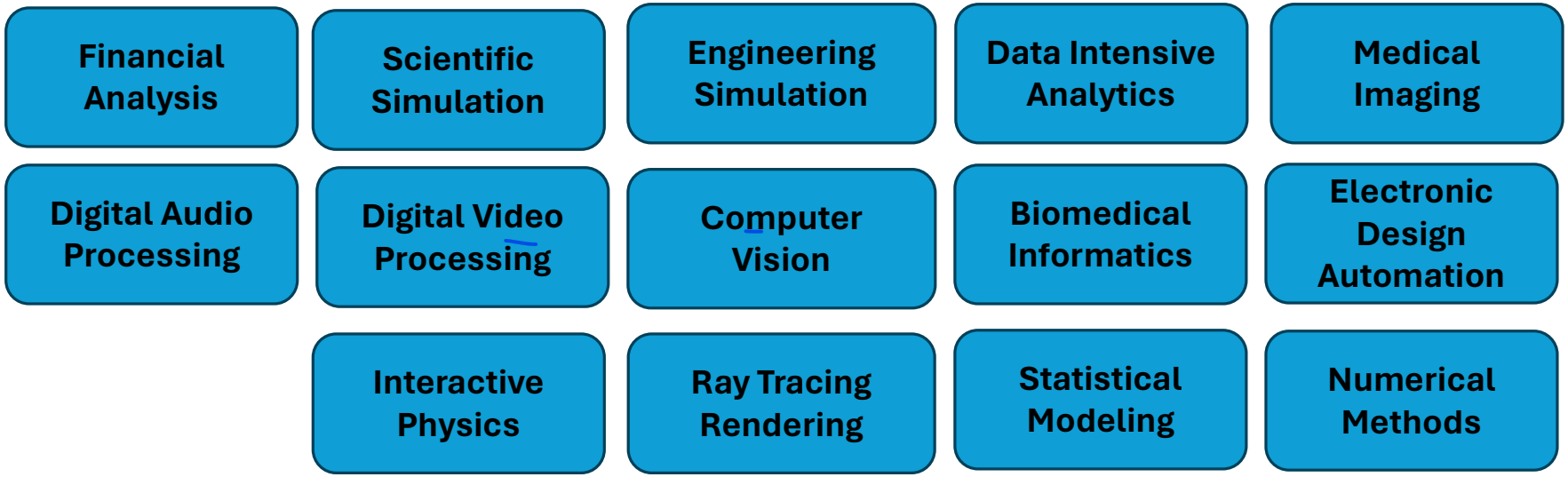
Dina Tantawy
Computer Engineering Department
Cairo University

general notes:
Copying data from host memory to device memory and vice versa takes much time -> bottelneck.
The function which is going to be executed on the GPU is called the kernel.
e7na a7san 7aga blnesblna en 3dd el blocks aw el threads ykon powers of 32 bayn.

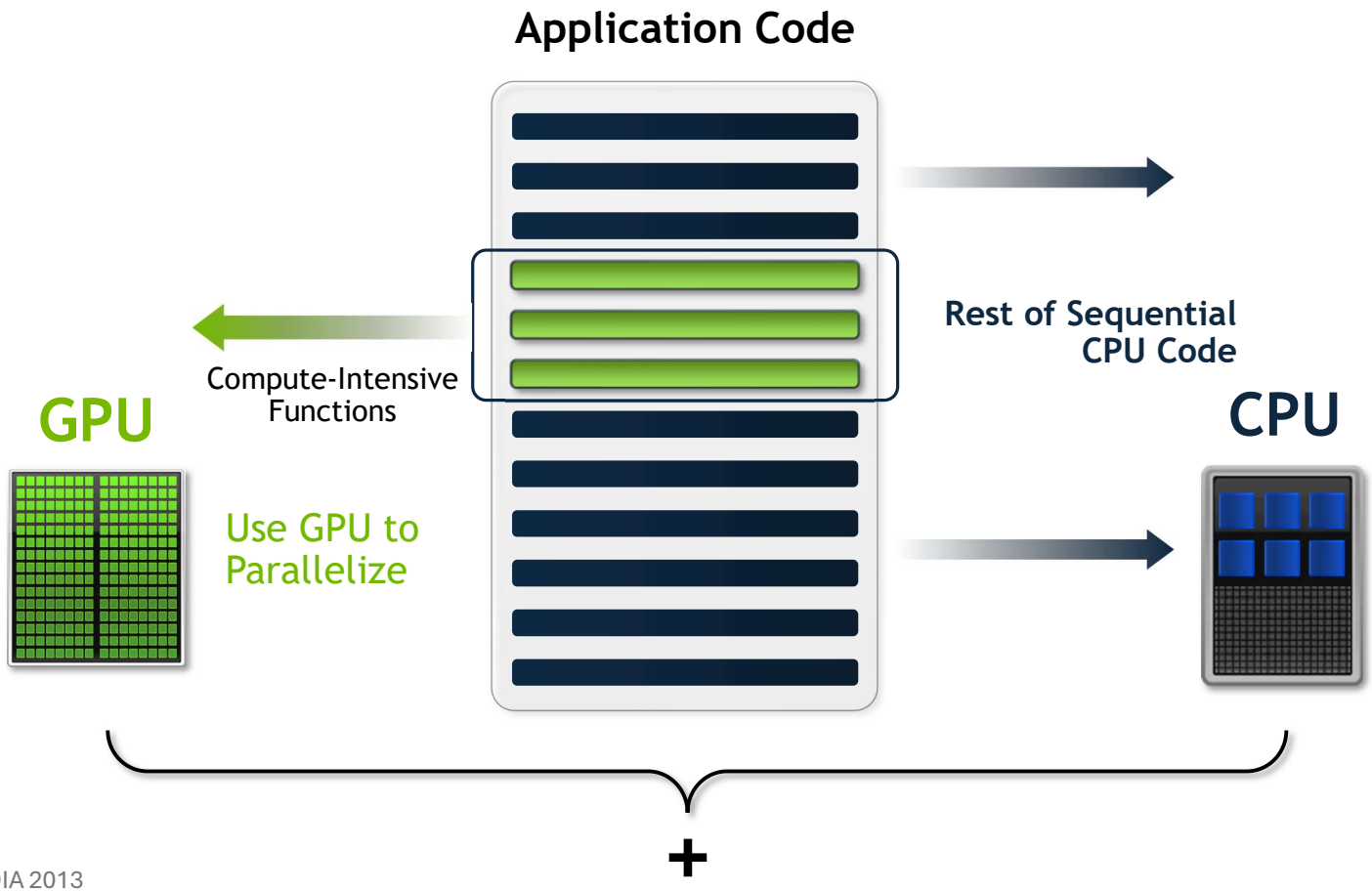
# Agenda

- Review
- Kernel Implementation
- GPU Grid
  - 1D example
  - 2D example
  - 3D exercise
- Architecture of GPU
  - SMs
  - Thread Scheduling (to be continued next time)

# Heterogeneous Parallel Computing in Many Disciplines

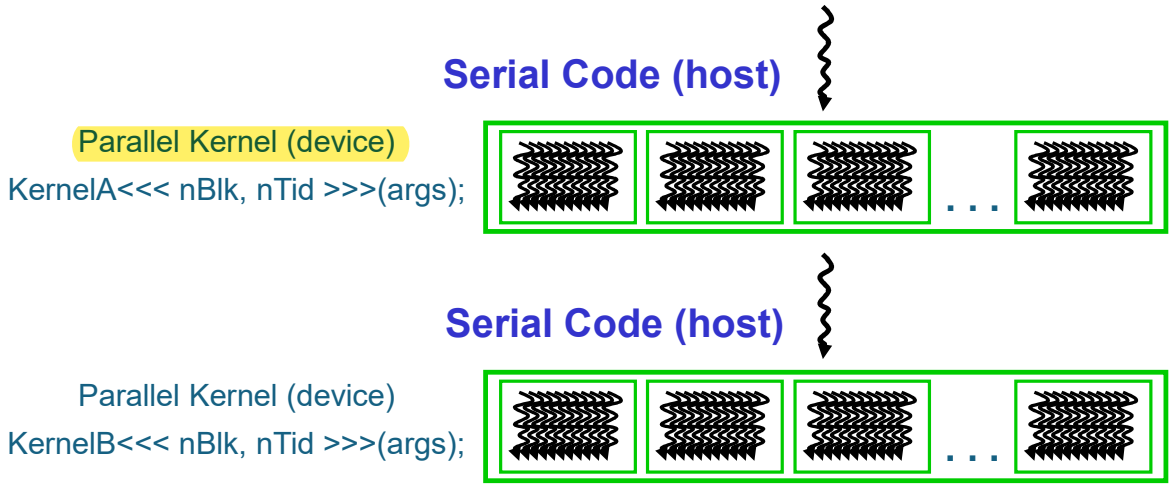


# Small Changes, Big Speed-up



# CUDA Execution Model

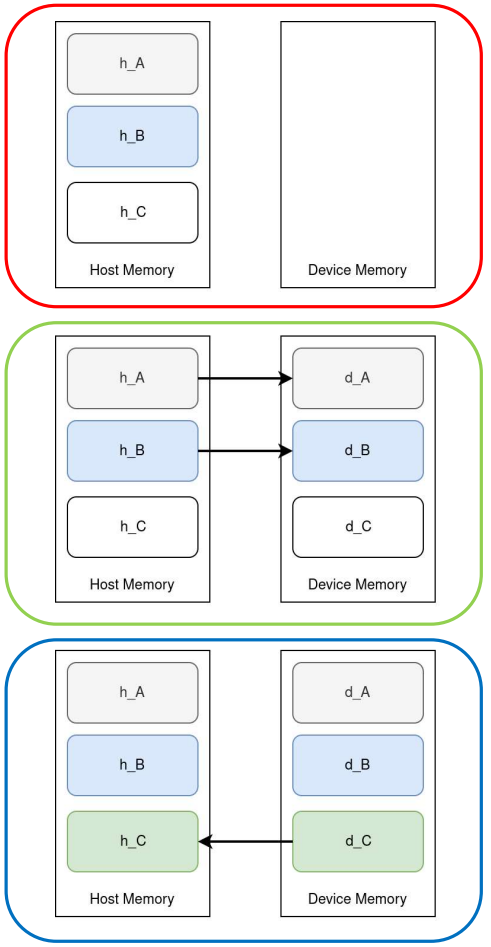
– Heterogeneous host (CPU) + device (GPU) application C program
  – Serial parts in host C code
  – Parallel parts in device SPMD kernel code



# Vector Addition, Explicit Memory Management

```
... Allocate h_A, h_B, h_C ...

void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

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

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

     // Kernel invocation code – to be shown later

     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

bn7gz mkan fl memory bta3 el GPU bl 7gm elly 3auzeno, w bnb3t el addresses elly hn5zn fehom.



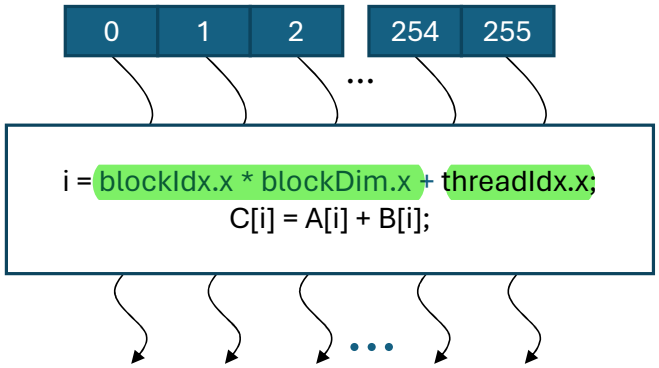
# Example: Vector Addition Kernel Launch (Host Code)

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and copies omitted
 // Run 1 blocks of 256 threads each
  vecAddKernel<<<1,256>>>(d_A, d_B, d_C, n);
}
```

Number of blocks

Number of threads




e7na bn7gz 3dd mo3yn mn el blocks, w bn2ol kol block feh kam thread htshtghl feh.
el hardware byedy idx lkol haga automatic, msh na el

el block hwa unit el gpu bystkhdmha 34an ye7gz resources.

lw ana 3ndy 256 thread, w 3auz a3mlhom 3la  2 blocks, sa3tha h3rfha keda -> <2, 128> -> fa kol block hyakhu 128, fa total hyb2a 256.

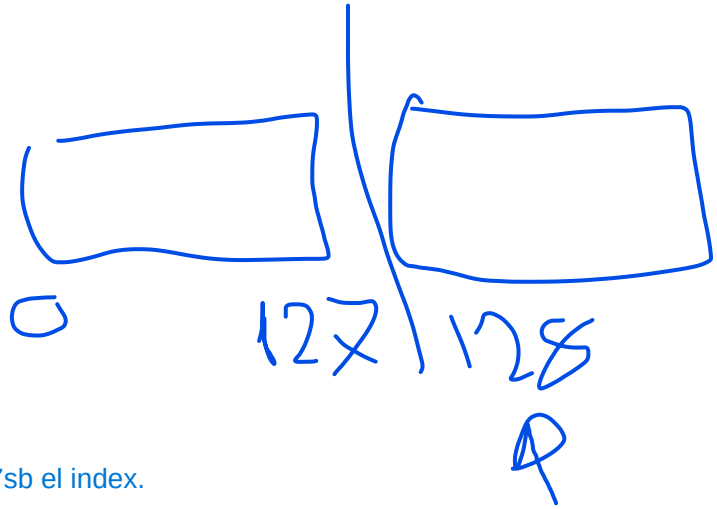
NVCC should be smart enough to divide the functions, which should go to the host (CPU) , and which go to the device (GPU)

by3rf keda ezay, by3dy 3la el code, lw l2a --global-- by3rf en el mfrod byb3t el esm lel cpu, wl implementation lel GPU -> bttndh mn el CPU, lakn bttnfz mn el GPU.

# Example: Vector Addition Kernel

## Device Code

```
// 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];
}
```




ay kernel equation, lazm dymn nla2y feha equation bt7sb el index.

dayman lama bnegy nshtghll threading, lazm n7ot boundry condition, 34an my7slsh 3ndy error.

w da 34an e7na sa3at bnla2y enna 7gzna threads aktur mn el threads elly ana m7tagha fe3ln.

34an a3rf el index da, bst5dm el stride equation.

threadId.x + blockDim.x*blockIdx.x

how to write the stride equation in case on 2D

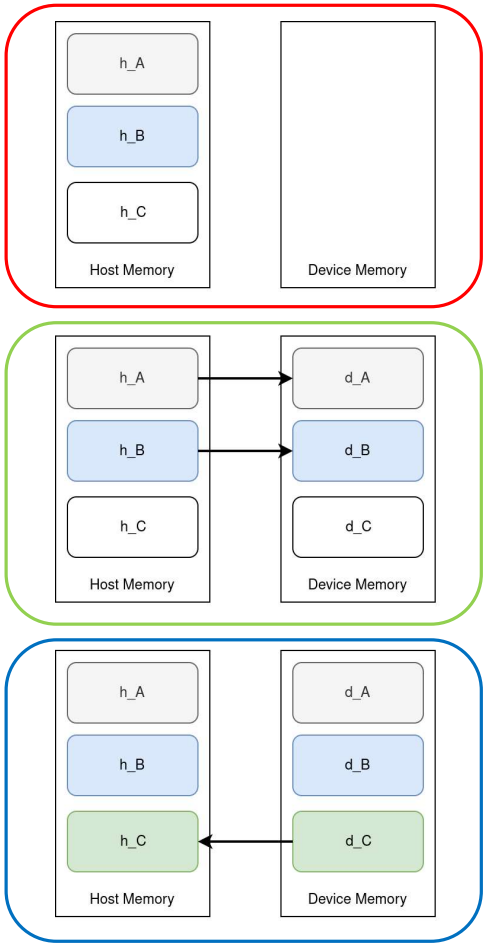
--global-- -> el cpu hwa el byndh el function 3n tre2 el header, wl GPU hwa el bynfzha 3n tre2 el implementation.
--device-- -> bytnfz 3la el GPU bs
--host-- -> kol haga 3la el CPU.

PTX -> esm el instruction set 3la el GPU.

lw 3auz anfzha 3la el CPU wl GPU fnfs el w2t, bn3ml
--device--
--host--
fo2 esm el function.
momkrn a7tag el case de lw 3ndy utility functions.

# Vector Addition, Explicit Memory Management

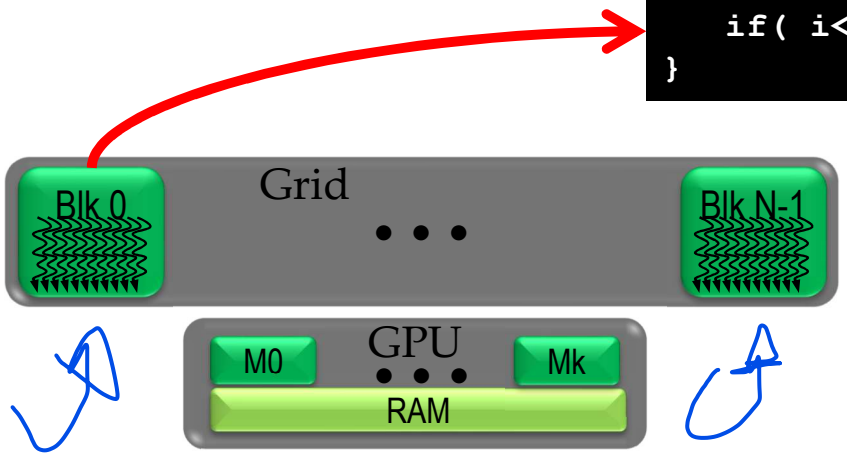
```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

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

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel invocation code – to be shown later
    vecAdd(d_A, d_B, d_C,256);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```



# Kernel execution in a nutshell

```
__host__
void vecAdd(…)
{
vecAddKernel<<<1,256>>>(d_A,d_B,d_C,n);
}
```

```
__global__
void vecAddKernel(float *A,
     float *B, float *C, int n)
{
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;

   if( i<n ) C[i] = A[i]+B[i];
}
```

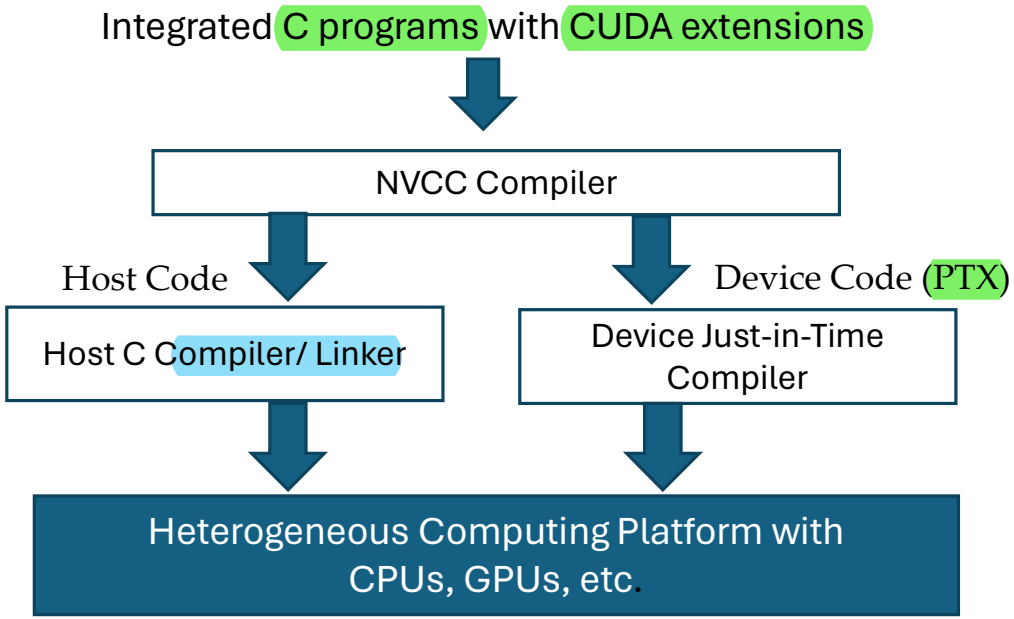


# More on CUDA Function Declarations

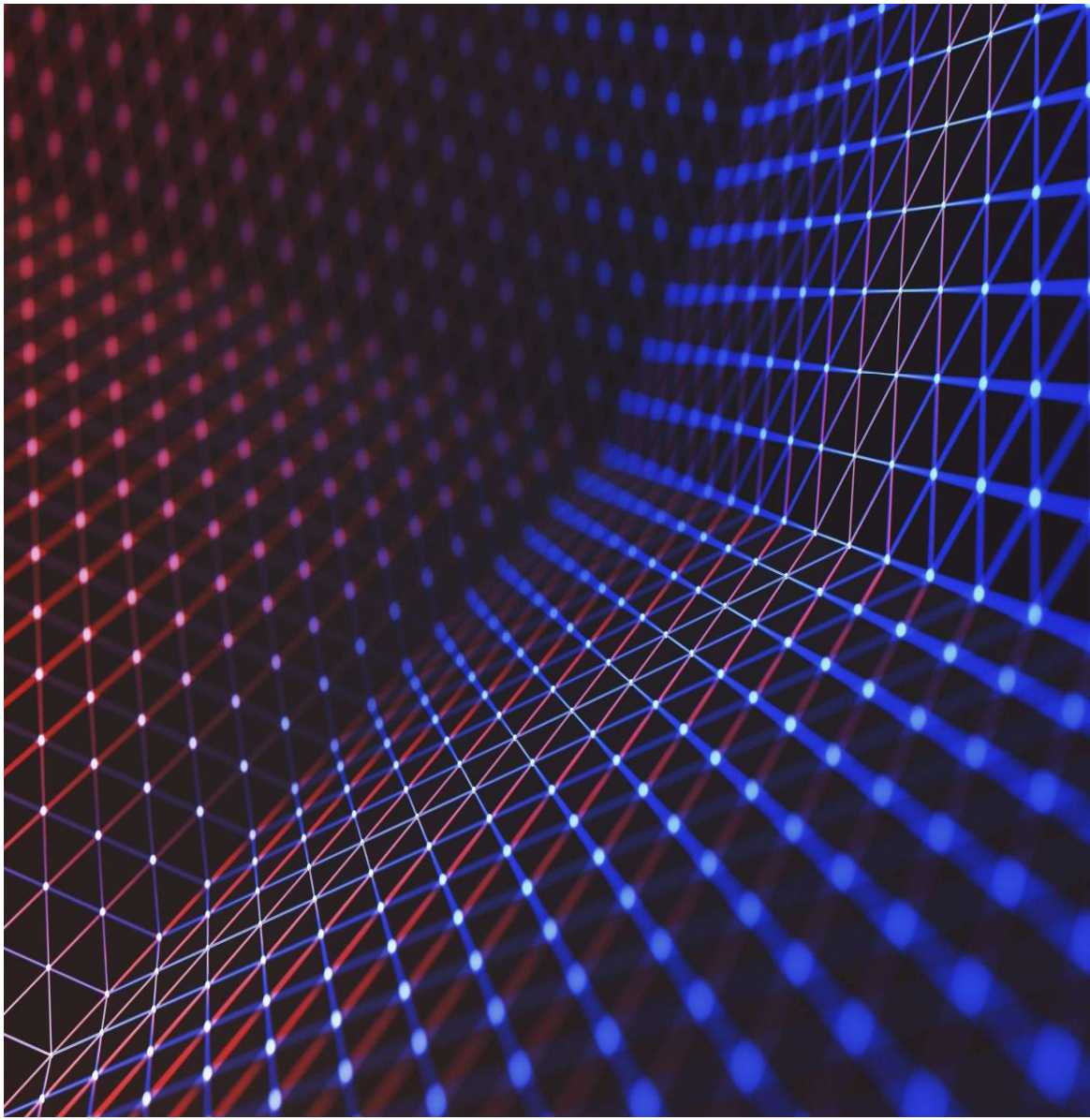
- __global__ defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return void
- __device__ and __host__ can be used together
- __host__ is optional if used alone

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

# Compiling A CUDA Program



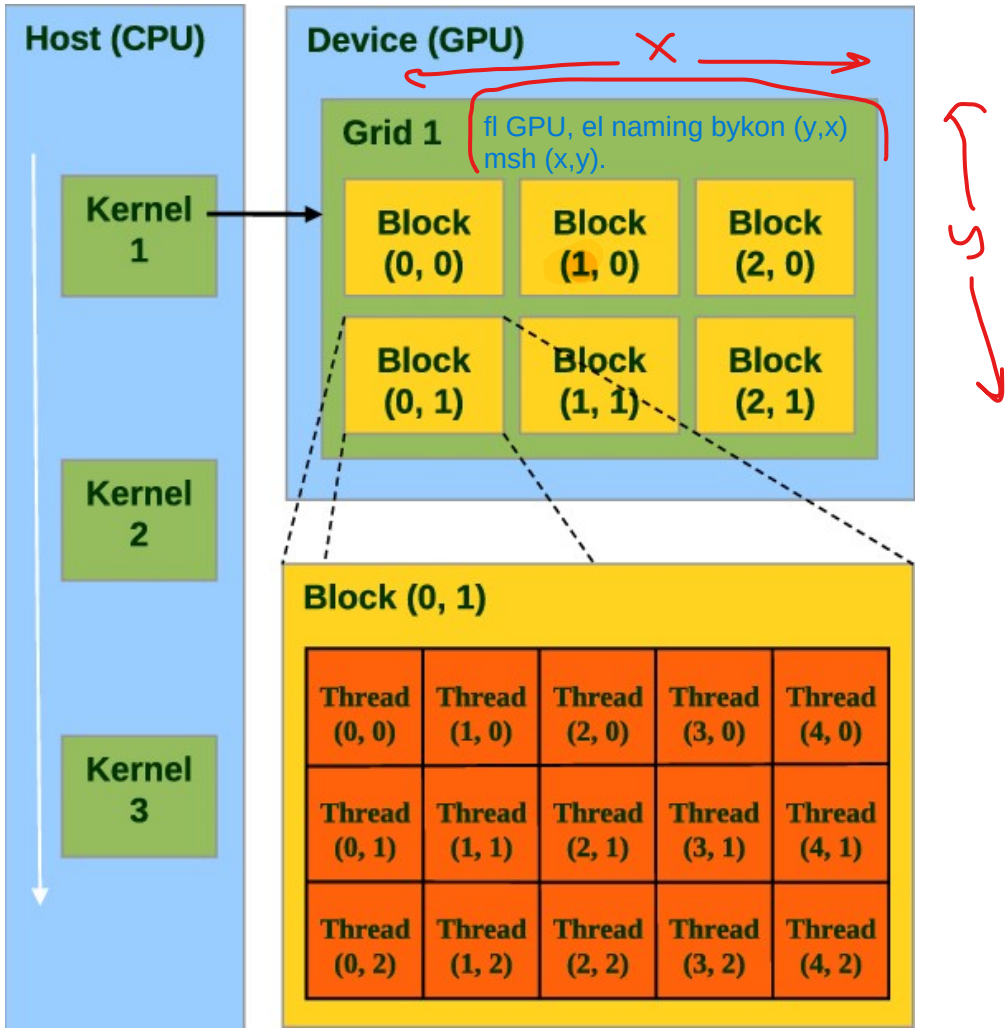
# Understand Grid



gp -> grid blocks
bt -> block threads

# Computational Grid

- Computational Grid consists of a number of blocks
- Each block consists of a number of threads.




dim3 gb( 3, 2, 1)  -> dim3 is a predefined struct, bst5dmha 34an a3rf el grid.
x, y, ,z

dim3 bt( 5, 3, 1)

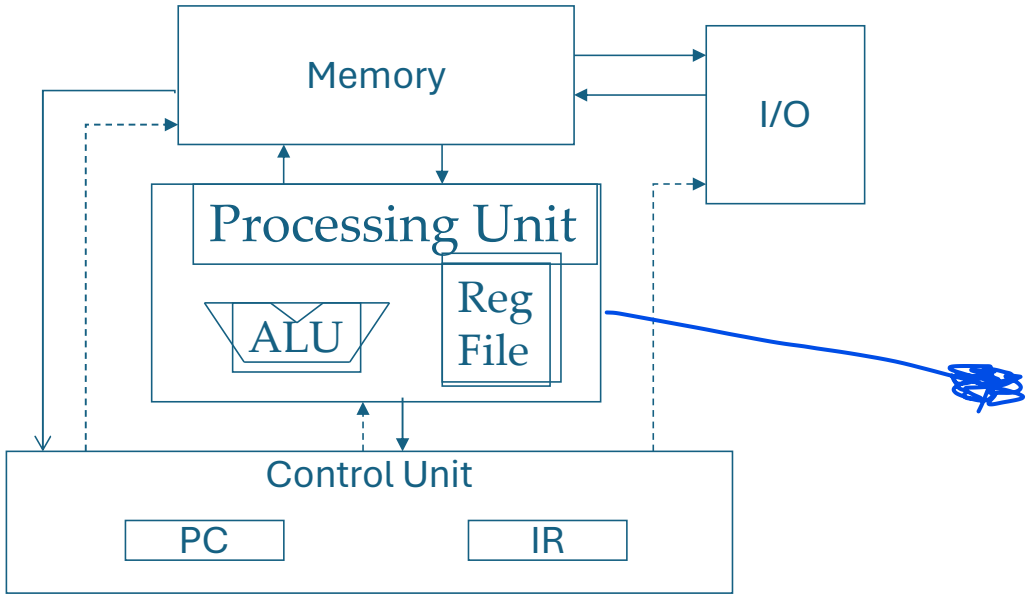
lama agy andh 3la el kernel
<<< gb, bt>>>

da m3nah eny b3rf gb -> 3dd el blocks, fa lw 3rft 3,2,1 -> da m3nah eno hy3rf (3 * 2 * 1) blocks w kol block hyb2a 3ndy 5*3*1 threads.

usuall questions on this part
# of Blocks = (x * y * z)
# of Threds per Block ( l * m * n )
total Number of Threads = # of Blocks * # of Threads per Block
# of actual Threads -> must be <= total Number of Threads.
# of actual Threads -> the given dimenstions multiplied together -> l2 3ndk image 300 * 200 msln yb2a = 6 * 10^4.

# A Thread as a Von-Neumann Processor

A thread is a “virtualized” or “abstracted”
Von-Neumann Processor

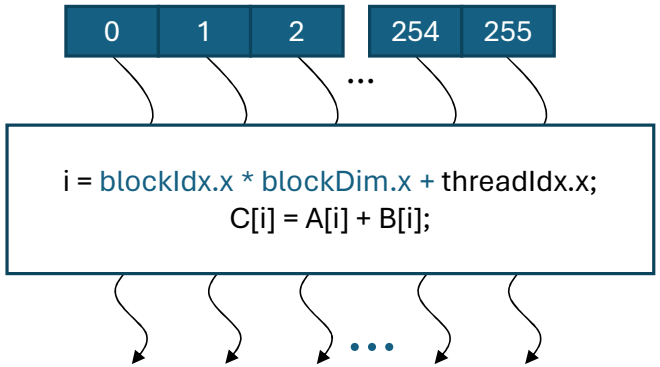



de kol thread 3ndha el goz2 bta3ha. kol wahda bt7sb lw7dha.

kol mgmo3a mn el threads btshtghl 3la control unit wa7da, da ntega le wgod SIMD units.
we kol el threads elly fe nfs el block 3ndhom shared memory -> w byb2a sahl enohom y3mlo sync.

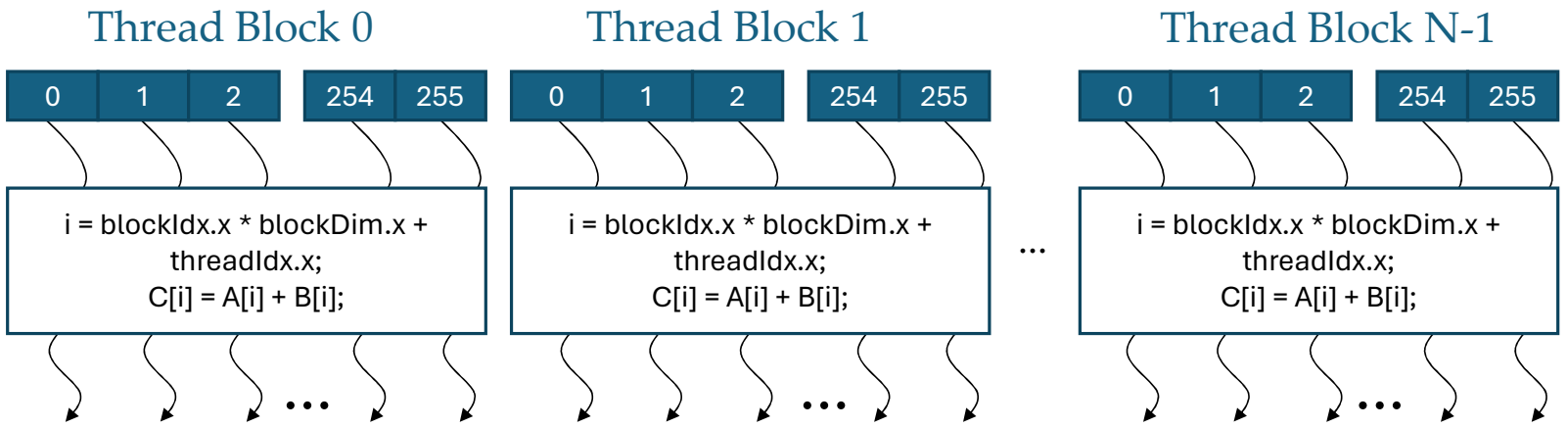
# Arrays of Parallel Threads

– A CUDA kernel is executed by a grid (array) of threads
  – All threads in a grid run the same kernel code (Single Program Multiple Data)
  – Each thread has indexes that it uses to compute memory addresses and make control decisions



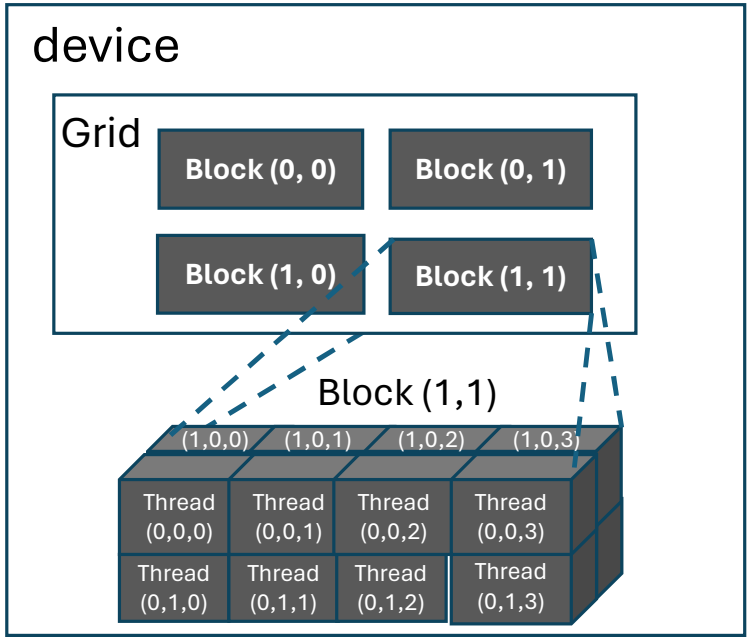
# Thread Blocks: Scalable Cooperation

- Divide thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks do not interact



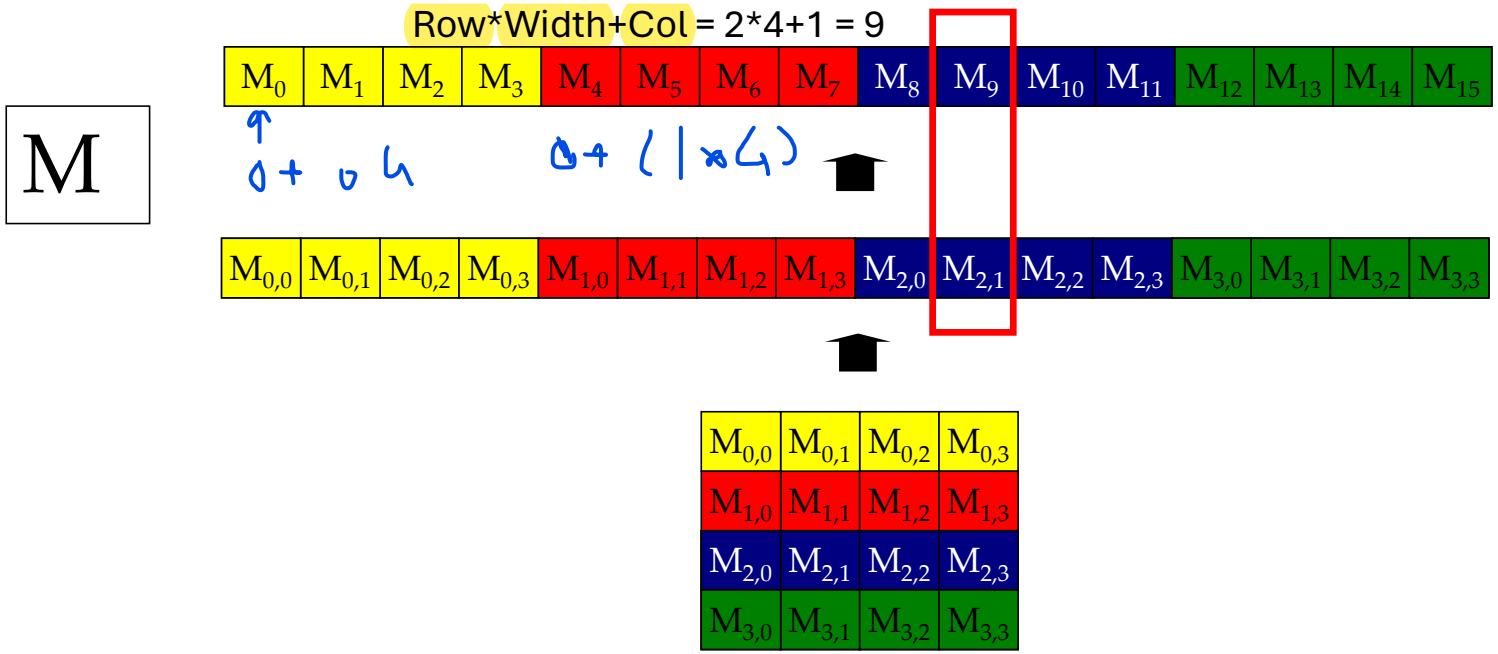
# blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



# Row-Major Layout in C/C++ (2D)

ThreadIdx.x + blockDim.x * blockIdx.x



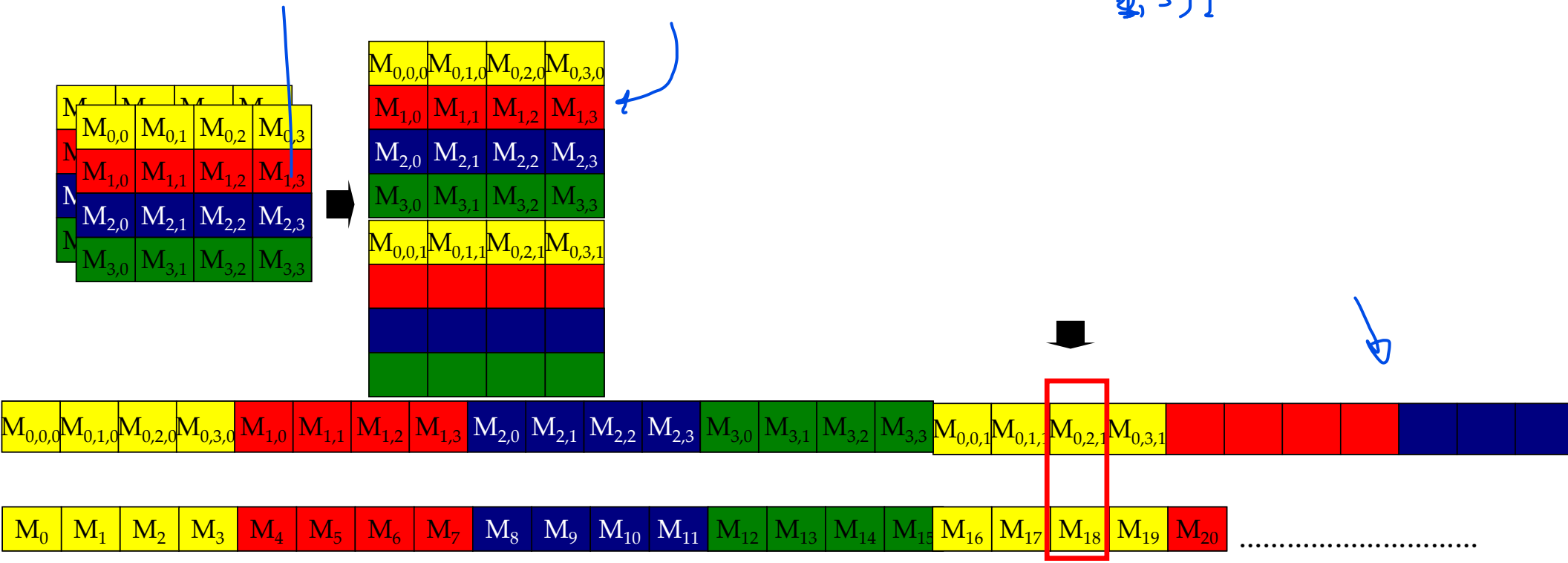

we apply matrix linearization

Col -> by7ddlk el index elly enta 3uzo fe ay row ( col )
Row * Width -> by7ddlk enta fe anhy row bzbt. -> Stride equation.

# Row-Major Layout in C/C++ (3D)

we always work with 0-index

3 + ( 1 * 4 ) + (4 * 4 + 1) = 7 + 16 = 23




Col + Width * Row + Width* Height * depth = 2 + 4*0+4*4 = 18

in case of 4 dim ?

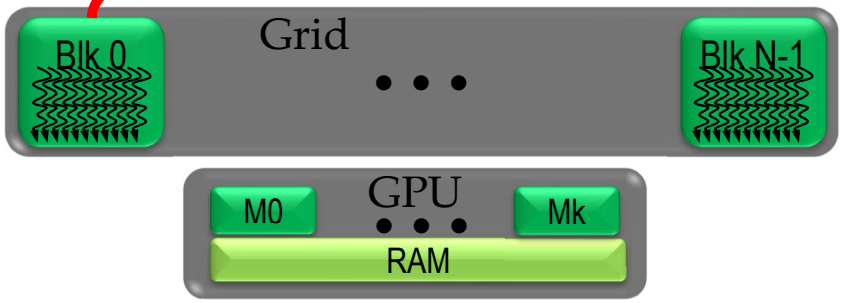
col + width * Row + width * Height * depth + width * height * depth * 4thFactorIdx.

idx row Square Cube

# Kernel execution in a nutshell

```
__host__
void vecAdd(…)
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);
vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B,d_C,n);
}
```

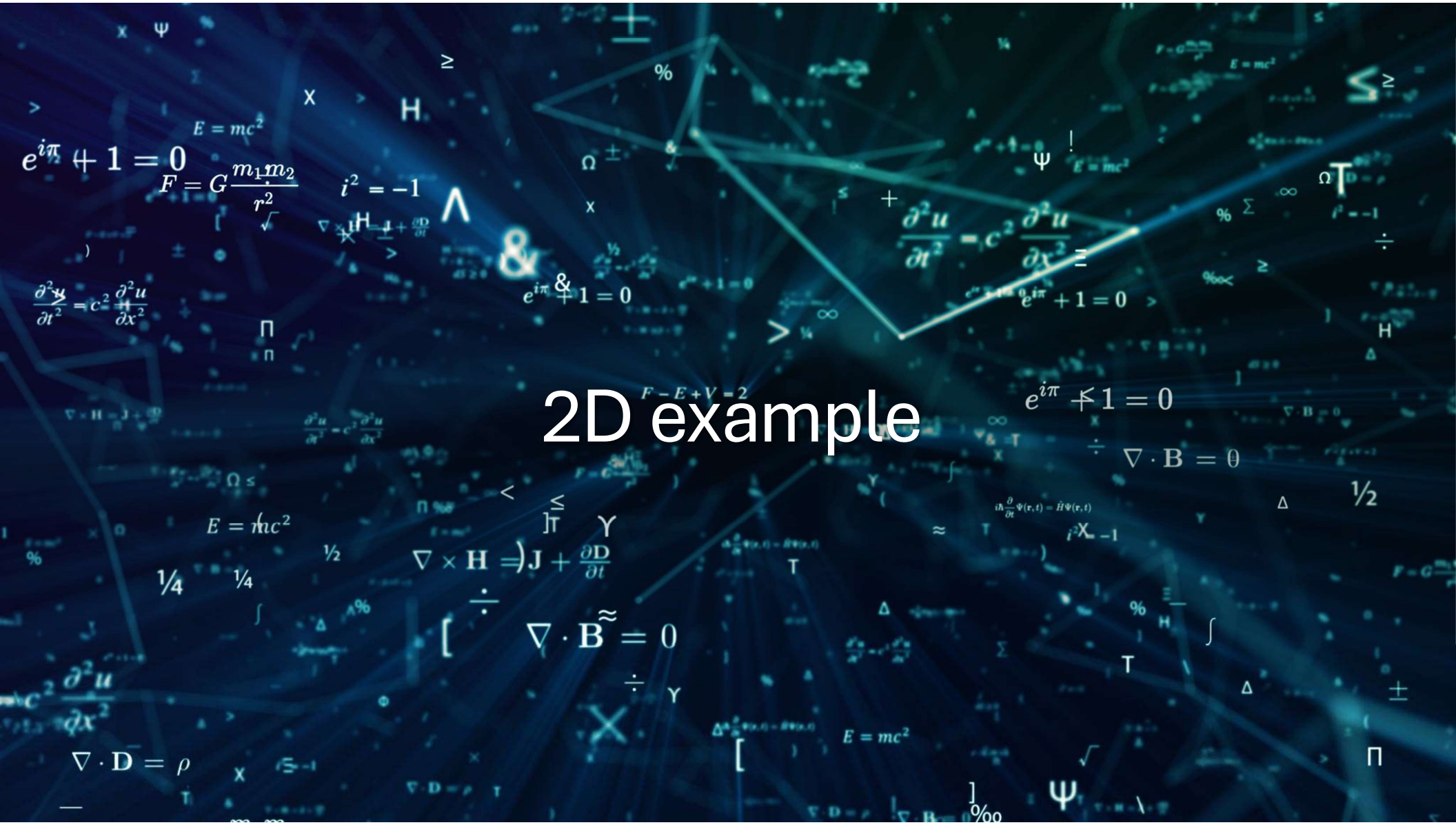
```
__global__
void vecAddKernel(float *A,
     float *B, float *C, int n)
{
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;

   if( i<n ) C[i] = A[i]+B[i];
}
```



# More on Kernel Launch (Host Code)

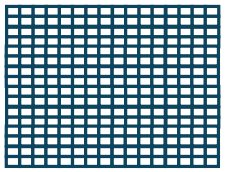
Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  dim3 DimGrid((n-1)/256 + 1, 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

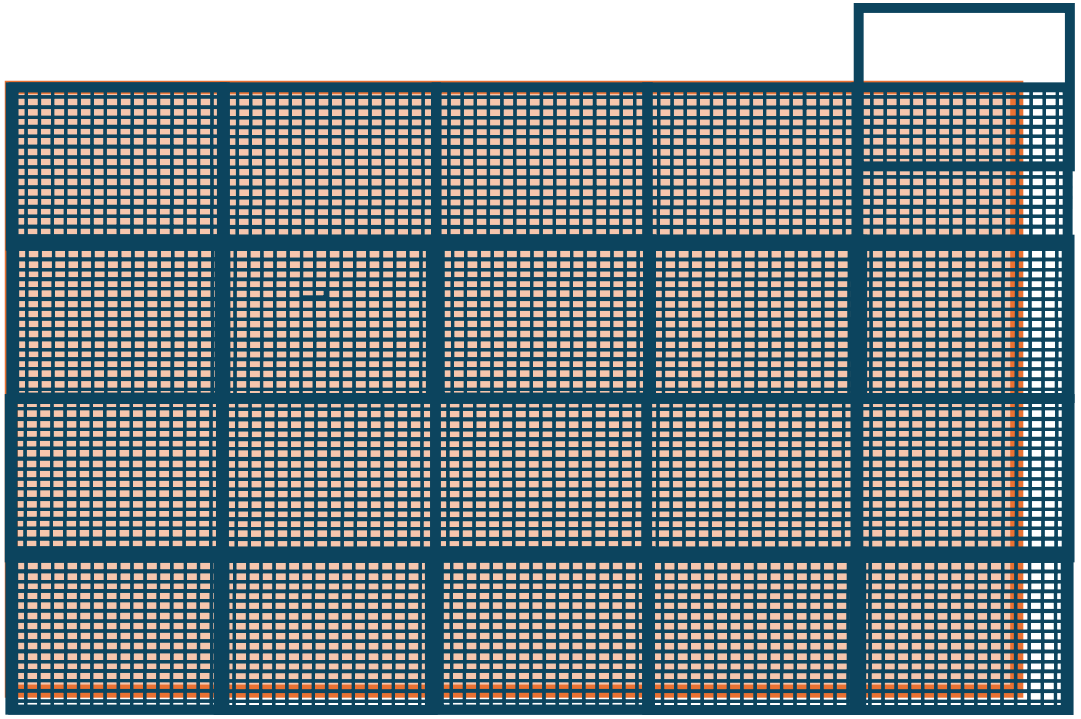

2D example

# Processing a Picture with a 2D Grid

- Write a kernel to scale the following image by 2
- Each block should have 16x16 thread,
- What is the correct configuration?



16×16 blocks



62×76 picture

number of required blocks = ceil(62 / 16) -> w hyb2a 3ndy 2 rows mlhumsh lazma

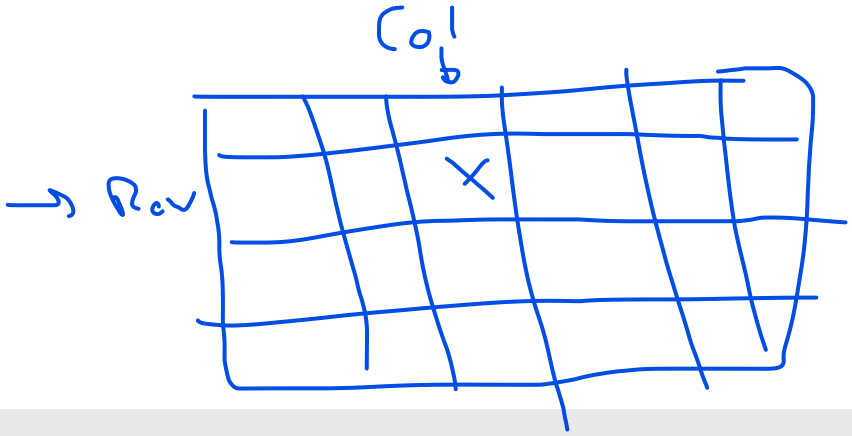
number of required threads = ceil(76 / 16)

int gb =dim3 ( (61 / 16) + 1 , (75 / 16) + 1 ,1 )

gb = dim3 ( 4, 5, 1) -> # grid blocks = 20
int bt = dim3 ( 16, 16, 1) -> # threads = 256

hence -> total # of threads = 5120
hence -> the actual # of threads = 4712
hence -> unused threads = 5120 - 4712 = 408

# Source Code of a PictureKernel




```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int height, int width)
{

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

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

  // each thread computes one element of d_Pout if in range
  if ((Row < height) && (Col < width)) {
    d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
  }
}
```

bgeb mkan el pixel fe anhy column.

bgeb mkan el pixel fe anhy row.

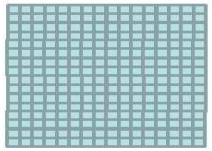
never forget the boundry condition.

Scale every pixel value by 2.0

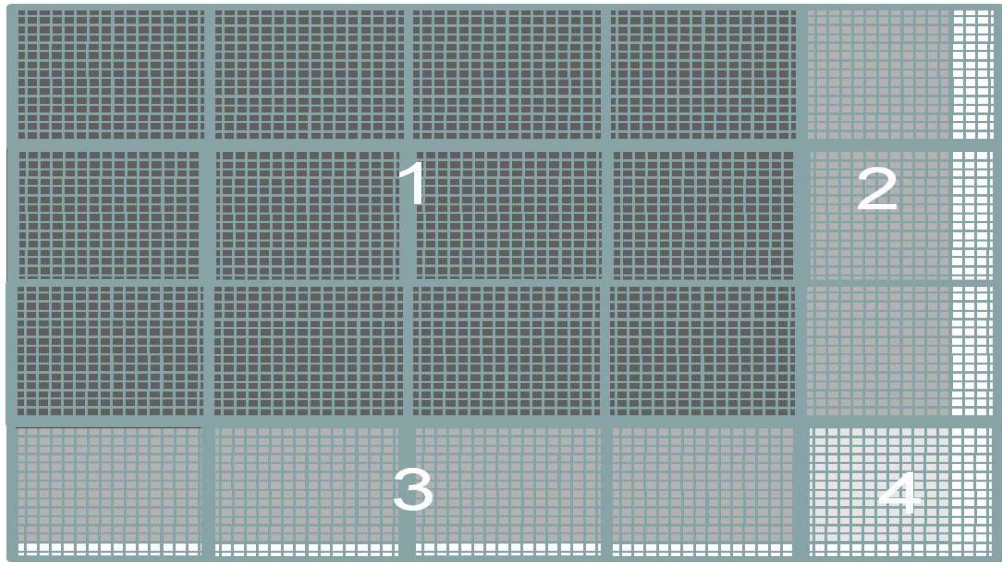
Host Code for Launching PictureKernel

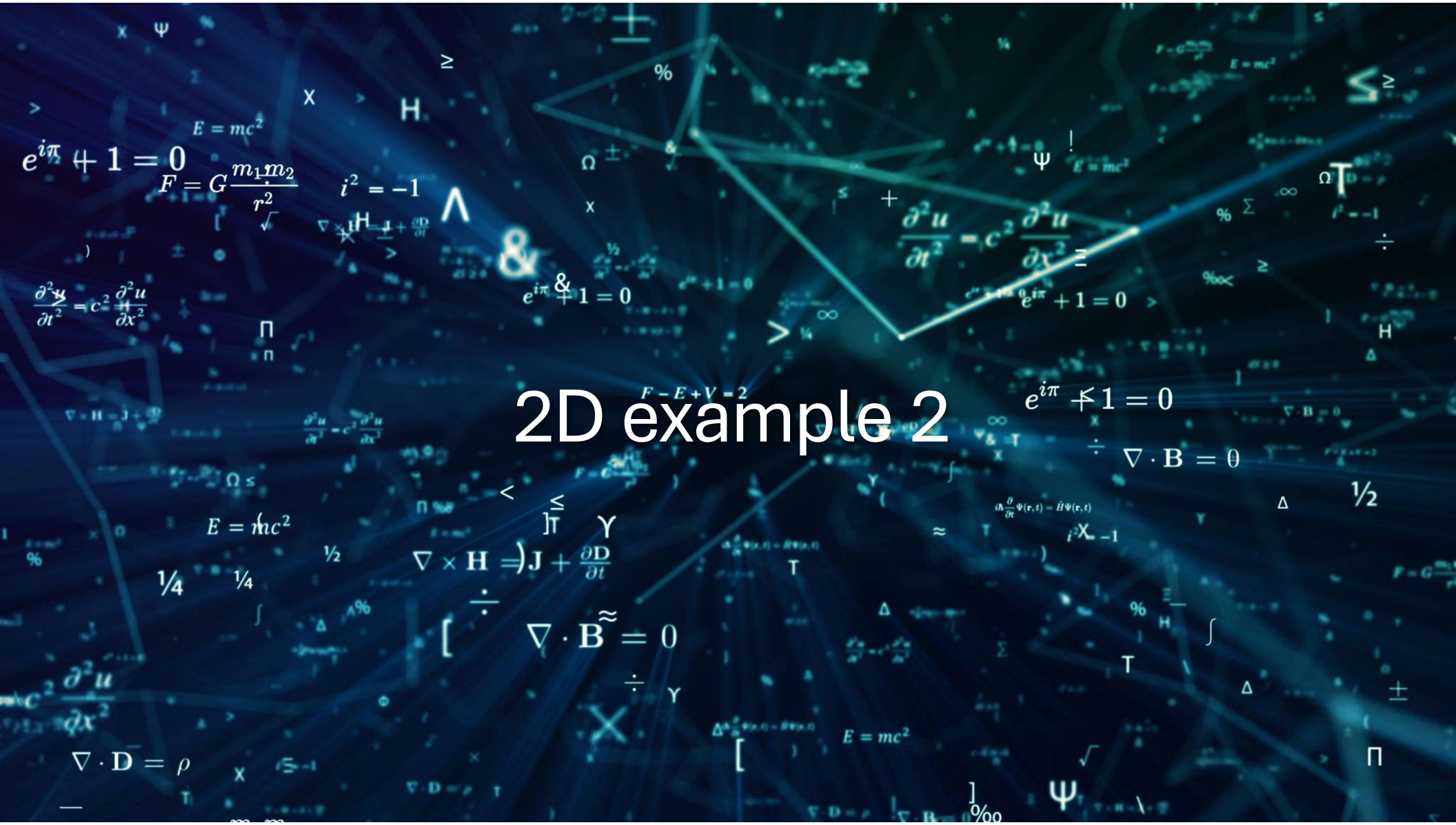
- // assume that the picture is m × n,
- // m pixels in y dimension and n pixels in x dimension
- // input d_Pin has been allocated on and copied to device
- // output d_Pout has been allocated on device
- ...
- dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
- dim3 DimBlock(16, 16, 1);
- PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
- ...

# Covering a 62×76 Picture with 16×16 Blocks

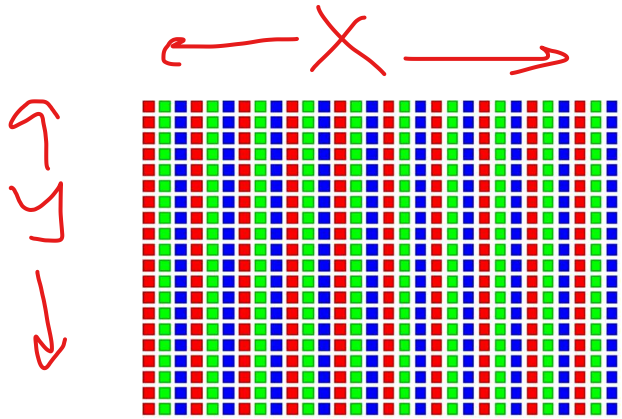


16×16 block

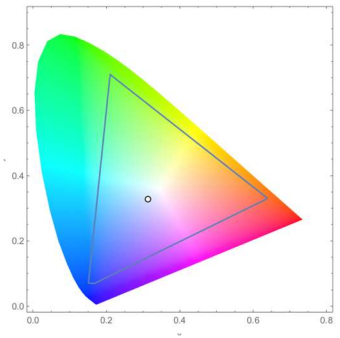



2D example 2

## RGB Color Image Representation



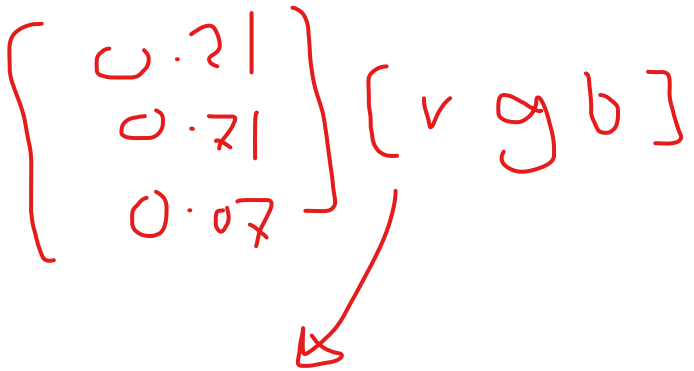

- Each pixel in an image is an RGB value
- The format of an image's row is
(r g b) (r g b) … (r g b)
- RGB ranges are not distributed uniformly
- Many different color spaces, here we show the constants to convert to AdobeRGB color space
  - The vertical axis (y value) and horizontal axis (x value) show the fraction of the pixel intensity that should be allocated to G and B. The remaining fraction (1-y–x) of the pixel intensity that should be assigned to R
  - The triangle contains all the representable colors in this color space



# RGB to Grayscale Conversion

A grayscale digital image is an image in which the value of each pixel carries only intensity information.

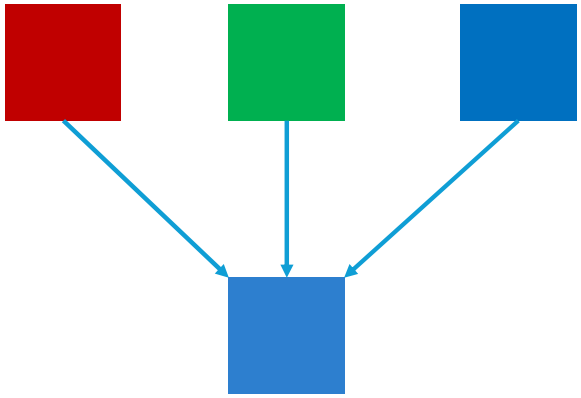
## Color Calculating Formula




- For each pixel (r g b) at (I, J) do:

  grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b

- This is just a dot product <[r,g,b],[0.21,0.71,0.07]> with the constants being specific to input RGB space



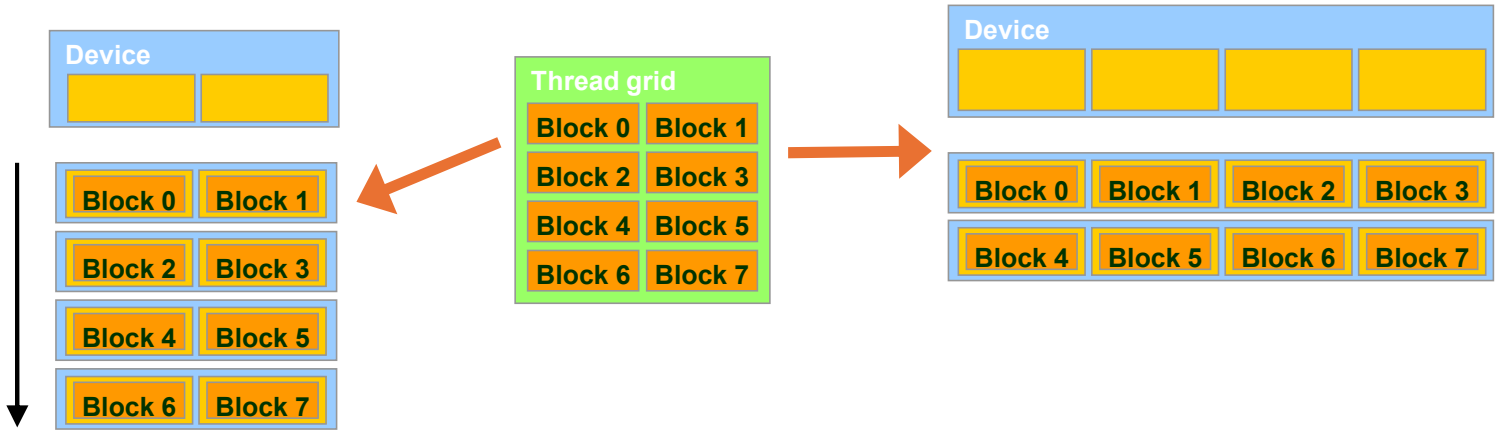
# RGB to Grayscale Conversion Code

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {
   // get 1D coordinate for the grayscale image
   int grayOffset = y*width + x;
   // one can think of the RGB image having
   // CHANNEL times columns than the gray scale image
   int rgbOffset = grayOffset*CHANNELS;
   unsigned char r =  rgbImage[rgbOffset    ]; // red value for pixel
   unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
   unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

# Into the GPU

## Transparent Scalability

- It is the ability of the system and applications to expand in scale without change to the system structure or the application algorithms.
- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors

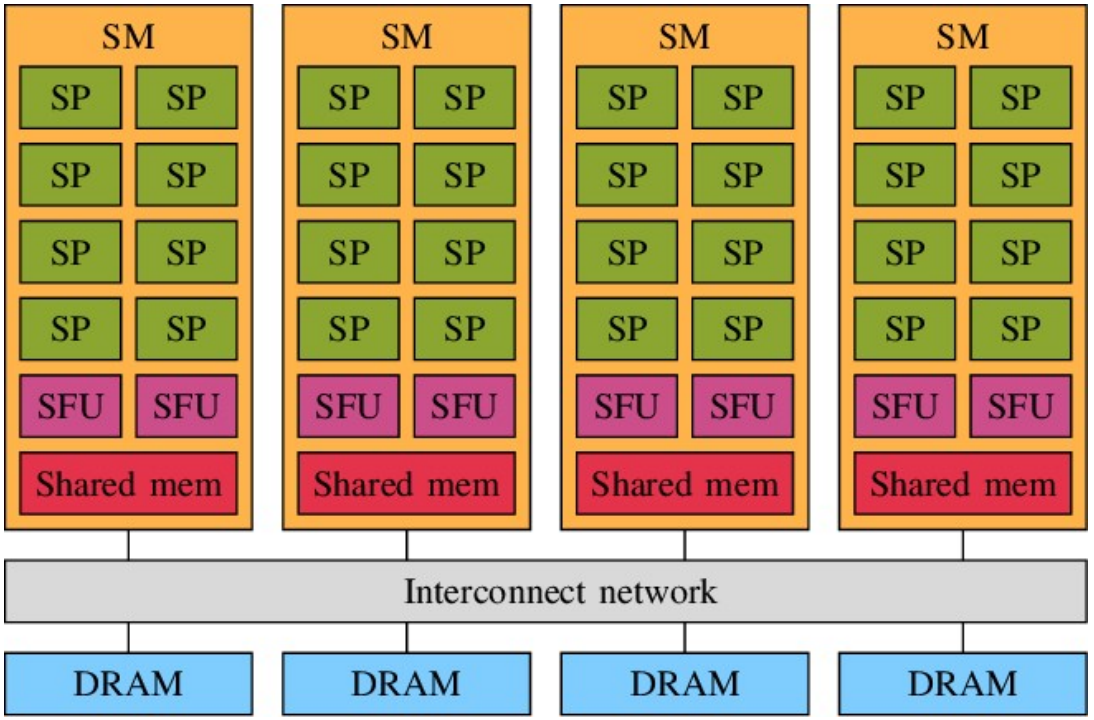


# SMs

The NVIDIA GPU consists of several Streaming Multiprocessors (SMs), four are shown in this figure. Each SM holds several Scalar Processors (SPs), usually eight.

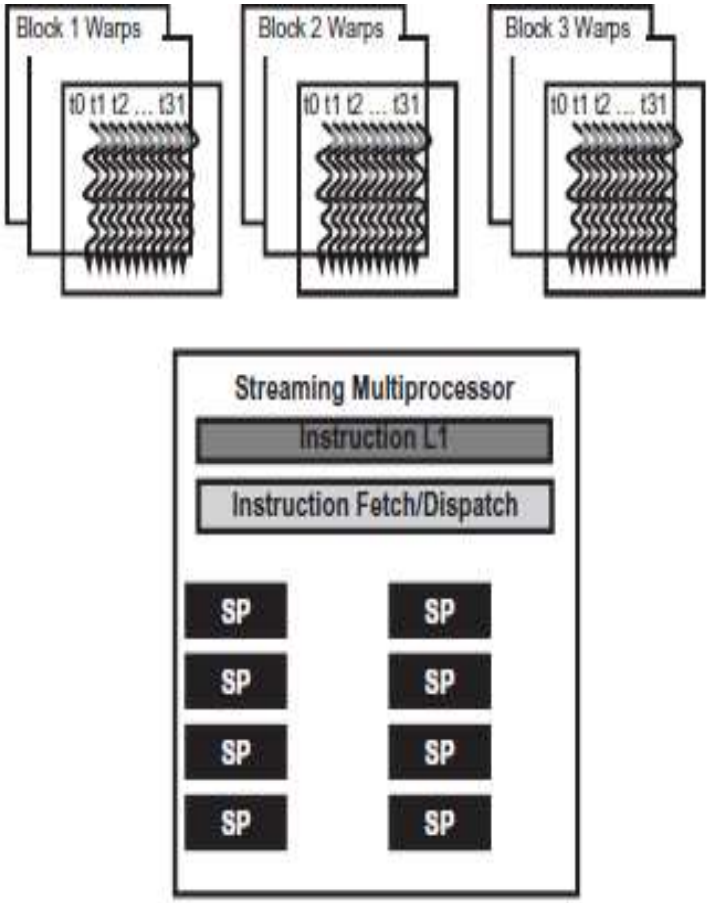
Each SM also has two Special Function Units (SFUs) and a small shared memory (Shared mem).

All SMs access the off-chip DRAM via the interconnect network

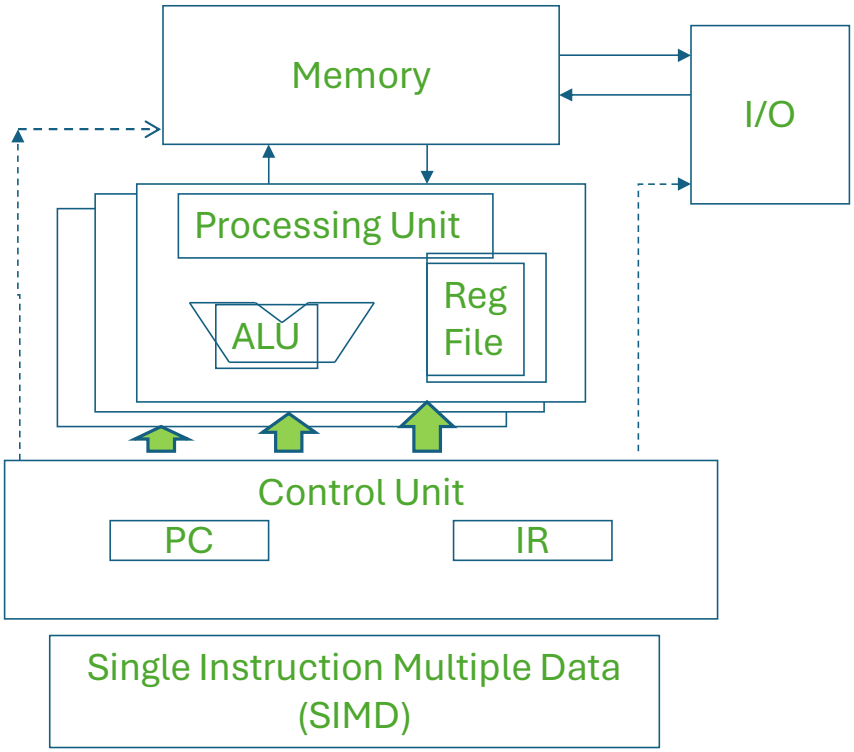


# Executing Thread Blocks

- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Up to 32 blocks to each SM as resources allow
  - Volta SM can take up to 2048 threads
    - Could be 256 (threads/block) * 8 blocks
    - Or 512 (threads/block) * 4 blocks, etc
- SM maintains thread/block idx #s
- SM manages/schedules thread execution



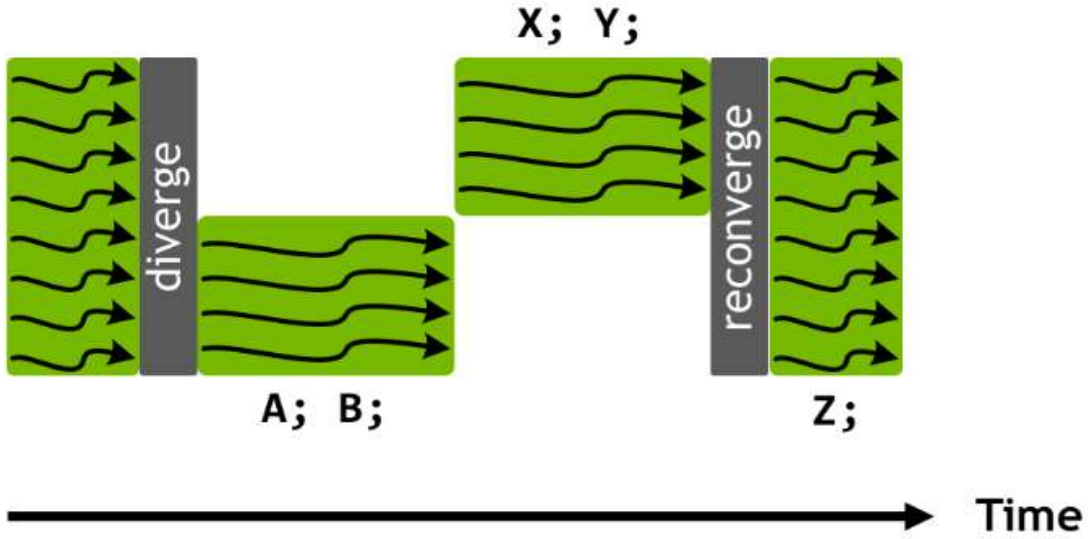
# The Von-Neumann Model with SIMD units



## Warps as Scheduling Units

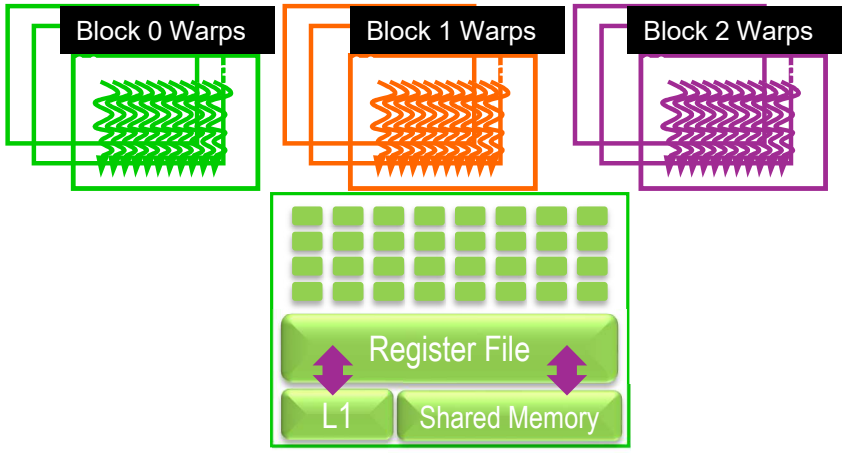
- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Warp Example

– If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  – Each Block is divided into 256/32 = 8 Warps
  – There are 8 * 3 = 24 Warps



Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution based on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

## Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?

## Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?

  - For 4X4, we have 16 threads per Block.
  - Each SM can take up to 2048 threads, which translates to 128 Blocks.
  - However, each SM can only take up to 32 Blocks, so only 512 threads will go into each SM!

## Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?

  - For 8X8, we have 64 threads per Block.
  - Since each SM can take up to 2048 threads, it can take up to 32 Blocks and achieve full capacity unless other resource considerations overrule.

  - For 30X30, we would have 900 threads per Block. Only two blocks could fit into an SM for Volta, so only 1800/2048 of the SM thread capacity would be utilized.

## Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?

  - For 30X30, we would have 900 threads per Block.
  - Only two blocks could fit into an SM for Volta, so only 1800/2048 of the SM thread capacity would be utilized.

# Summary

- Kernel Implementation
  - Host / device / global codes
- GPU Grid
  - Linearization of Indices is a key
- Architecture of GPU
  - SMs
  - Thread Scheduling (to be continued next time)

Thank You