Types of Parallelism :-

Task Parallelism

→ Different Operations performed on same or different data.

→ usually a modest number of tasks unleash a modest amount of parallelism.

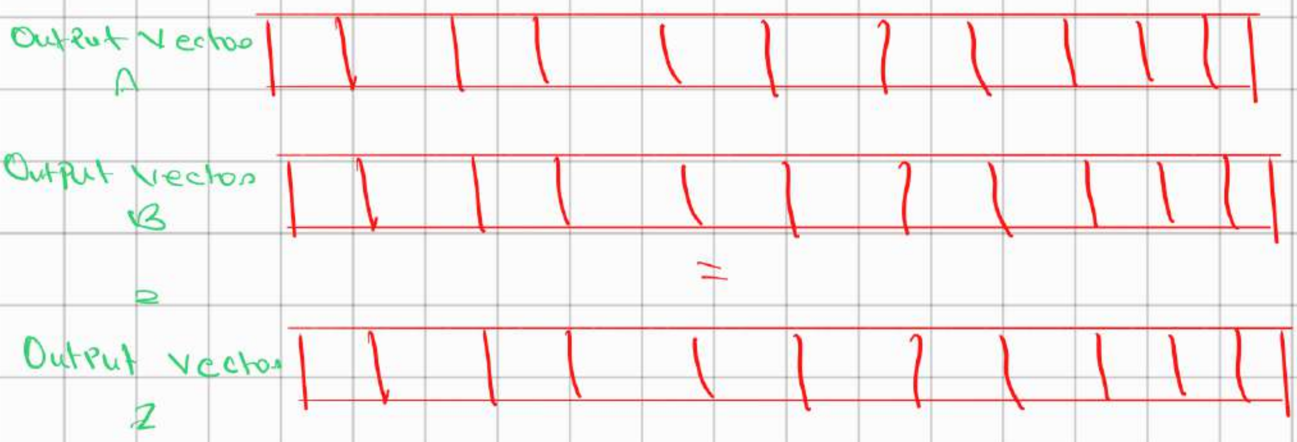
Data Parallelism.

→ Same Operations performed on different data.

→ Potentially massive amount of data unleash massive amount of parallelism.

• Most suitable for GPU.

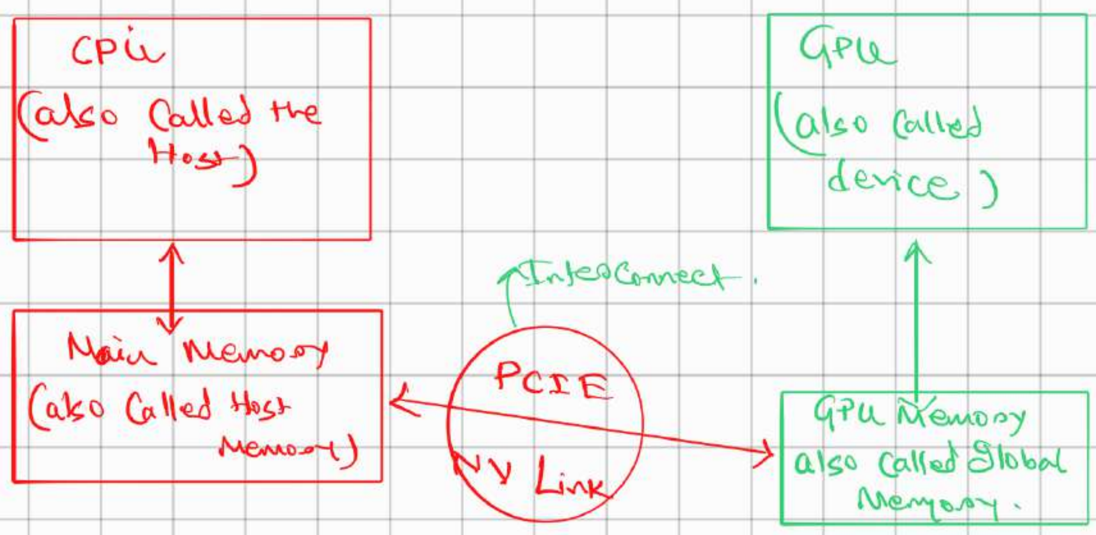
Vector Addition :- (Hello World for Cuda programming].



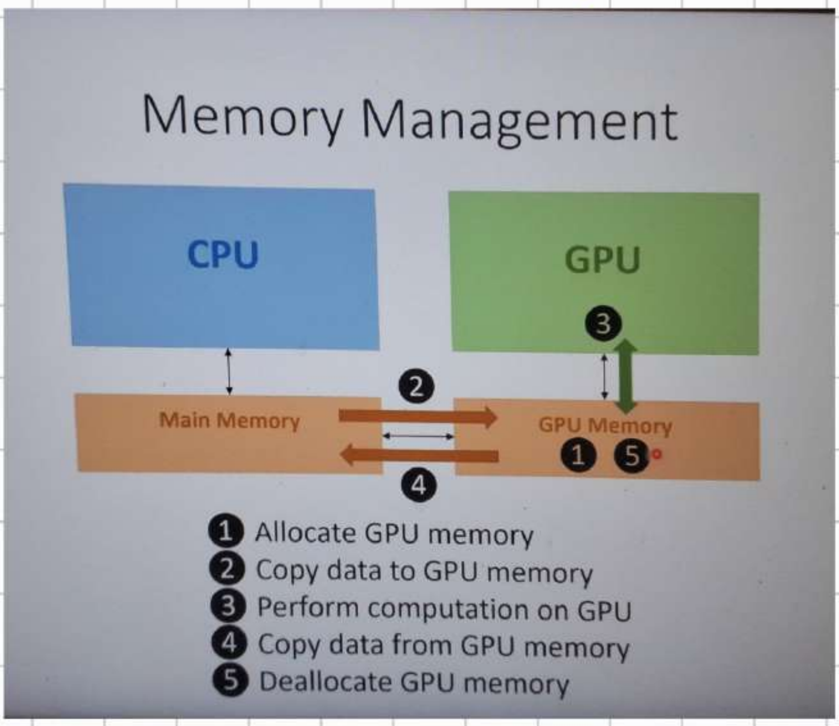

```
int main(int argc, char**argv) {

    cudaDeviceSynchronize();

    // Allocate memory and initialize data
    Timer timer;
    unsigned int N = (argc > 1)?(atoi(argv[1])):(1 << 25);
    float* x = (float*) malloc(N*sizeof(float));
    float* y = (float*) malloc(N*sizeof(float));
    float* z = (float*) malloc(N*sizeof(float));
    for (unsigned int i = 0; i < N; ++i) {
        x[i] = rand();
        y[i] = rand();
    }

    // Vector addition on CPU
    startTime(&timer);
    vecadd_cpu(x, y, z, N);
    stopTime(&timer);
    printElapsedTime(timer, "CPU time", CYAN);

    // Vector addition on GPU
    startTime(&timer);
    vecadd_gpu(x, y, z, N);
    stopTime(&timer);
    printElapsedTime(timer, "GPU time", DGREEN);

    // Free memory
    free(x);
    free(y);
    free(z);

    return 0;

}
```

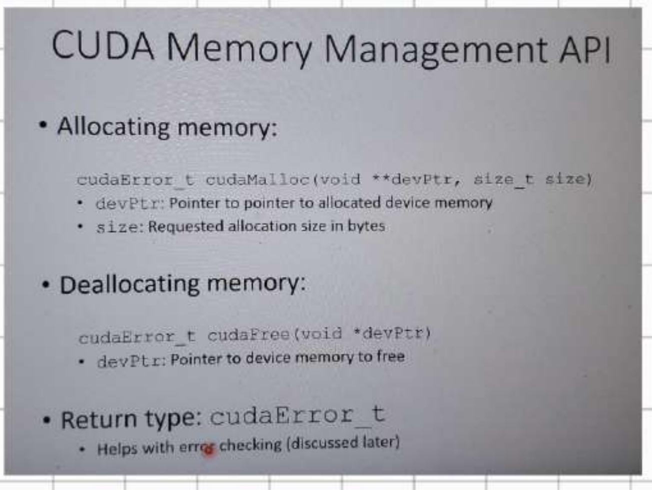
going to implement vec

System Organization:-



The CPU and GPU have Seperate memory and Cannot access each other's Memory.





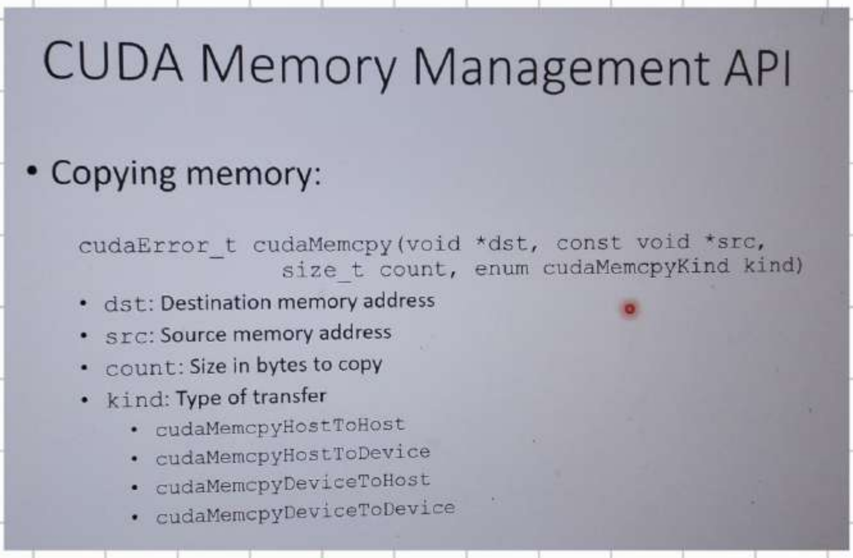
Cuda Memory Management API:-

- Allocating Memory:- CudaError_t CudaMalloc(void **devPtr, size_t size)
  - devPtr: Pointer to Pointer to Allocate device Memory.
  - Size:- Requested Allocation size in Bytes.

```
void VecAdd_gpu(float *x, float *y, float *z, int N){
    //Allocate GPU Memory
    float *x_d, *y_d, *z_d;
    CudaMalloc((void**) &x_d, N* sizeof(float));
```

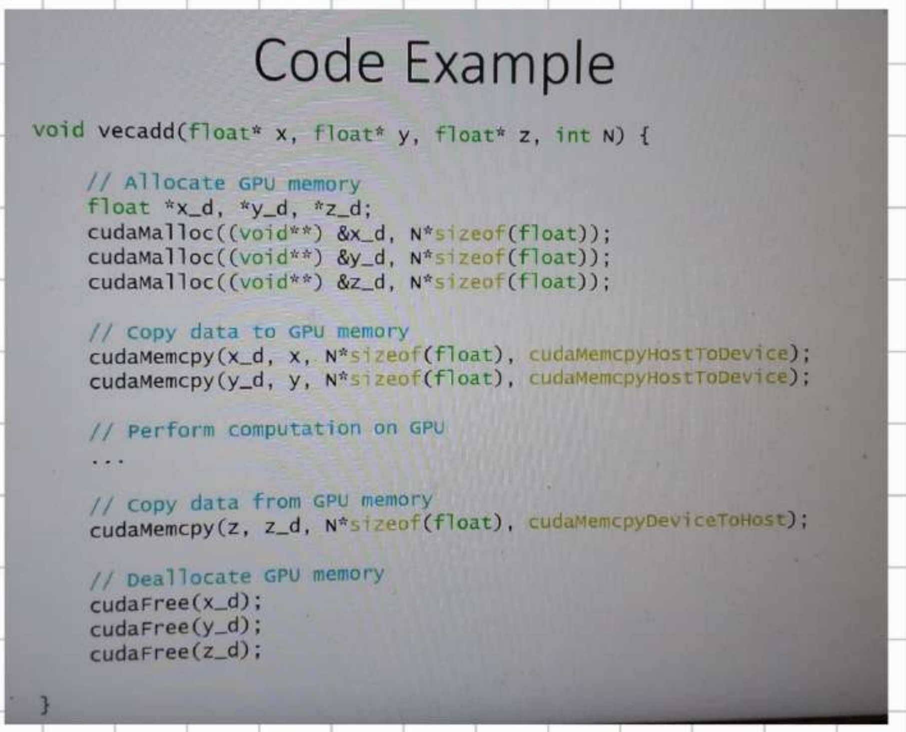
→ functions Cannot modify Arguments so if I want CudaMalloc to modify x_d. I should give a pointer to x_d.

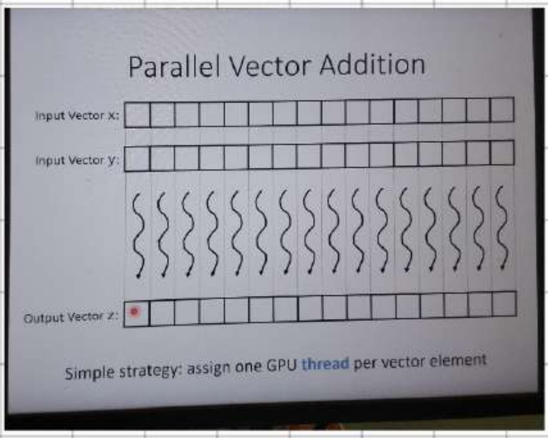
```
    CudaMalloc((void**) &y_d, N * Sizeof(float));
    CudaMalloc((void**) &z_d, N * sizeof(float));

    //Copy to the GPU.
    // Run the GPU Code
    // Copy from the GPU.
    // Deallocate GPU Memory. //CudaFree(x_d);
```




```
// Copy to the GPU.
   cudaMemCpy ( x_d, x , N* sizeof (float), cudaMemCpyHostToDevice);
   cudaMemCpy (y_d, y, N *size of (float), cudaMemCpyHostToDevice).
// Call a GPU kernel function.(launch a grid of threads)

   Const Unsigned int numThreadsPerBlock = 512;
   Const Unsigned Int numBlocks = (N + 512 - 1)/512 → This is effectively a ceil operation

   vecadd_kernel<<< numBlocks, numThreadsPerBlock >>> (x_d, y_d, z_d, N);
   CudaDeviceSynchronize( );
// Copy from GPU to CPU.
   cudaMemCpy (z, z_d, N * size of (float), CudaMemCpyDeviceToHost);

// Deallocate GPU memory
   CudaFree(x_d);
   CudaFree(y_d);
   CudaFree( z_d);
```

* If I have N elements in my Vector that means I want to have N threads in total.

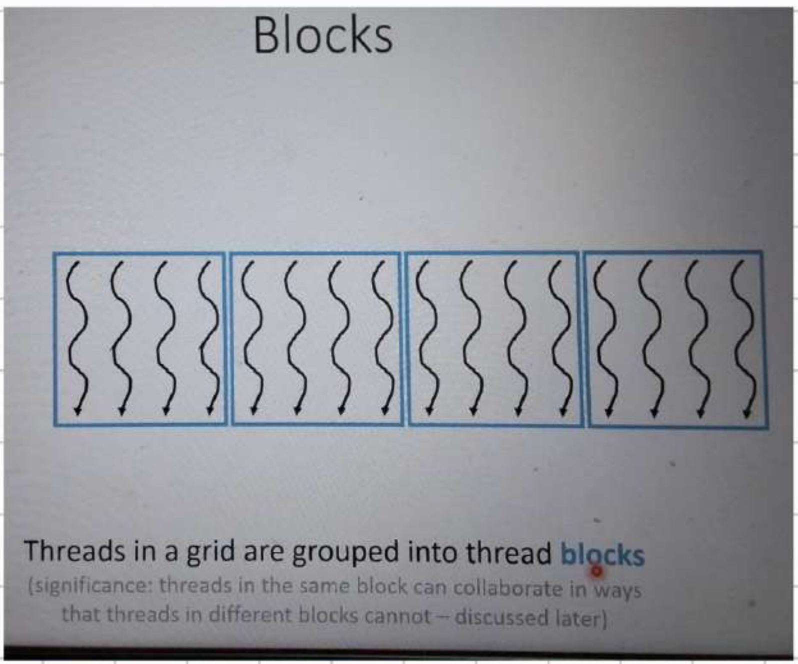




How threads on the GPU are Organized?

→ An array of GPU threads is called the grid.

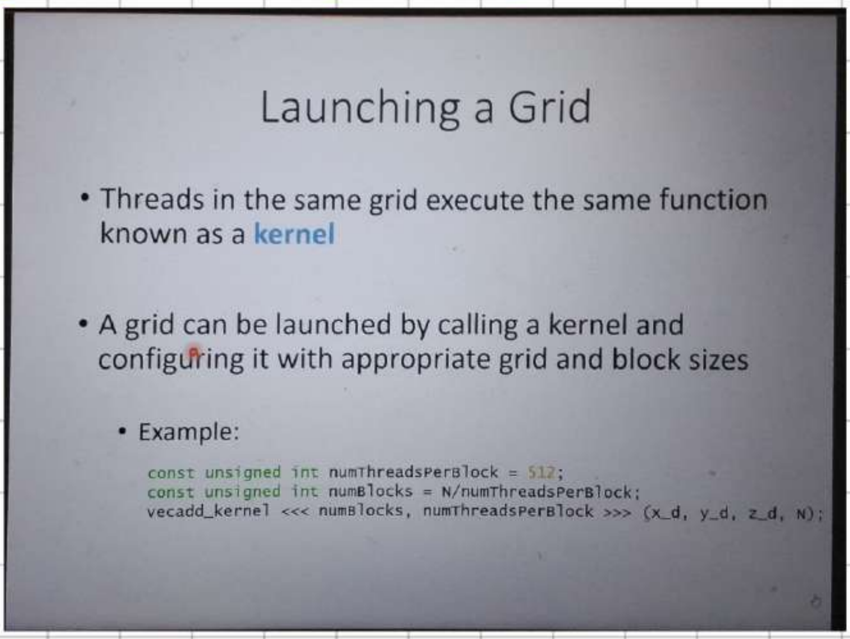
From the CPU we create whole bunch of threads on the GPU. to Perform some Operation these threads are Organised on the grid.

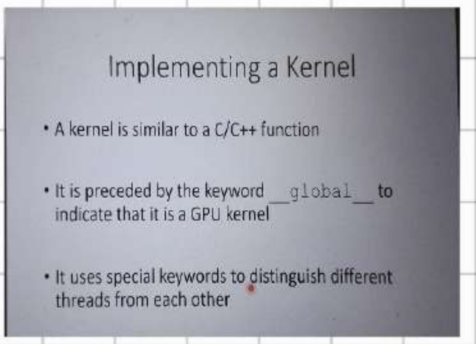


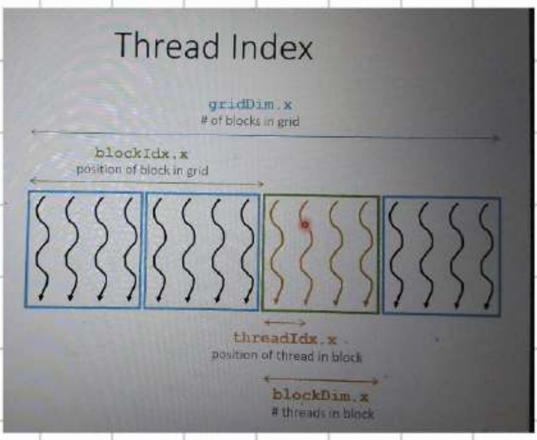
I would like to launch a grid of threads and to launch a grid of threads. I actually needs to specify how many blocks I need in the grid, and how many threads I want in each of my grid.

LAUNCHING A GRID

- Threads in the same grid execute the same function known as a kernel. and the way to launch a grid is by calling this special function a kernel and telling this function what is the grid size in other words the numbers of the blocks size. numbers of threads in each block.








```
__global__ void vecadd_kernel(float *x, float *y, float *z, int N)
{
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i<N){
        z[i] = x[i] + y[i];
    }
}
```

→ Single Program Multiple Data.

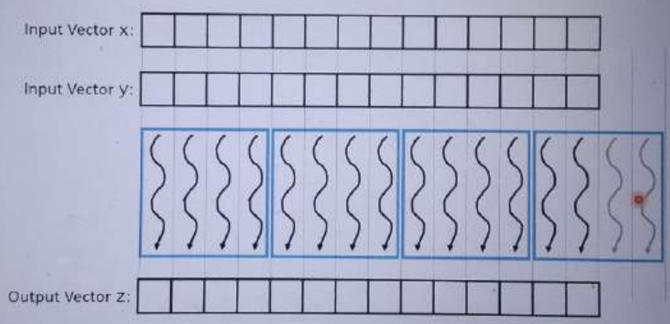
Boundary Conditions

- Recall launch configurations:

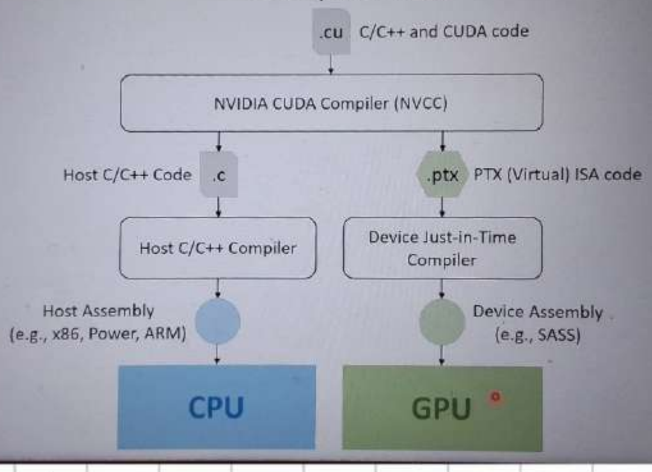
```
const unsigned int numThreadsPerBlock = 512;
const unsigned int numBlocks = N/numThreadsPerBlock;
vecadd_kernel <<< numBlocks, numThreadsPerBlock >>> (x_d, y_d, z_d, N);
```

- If N is not a multiple of numThreadsPerBlock, fewer threads will be launched than desired
- Solution: use the ceiling to launch extra threads then omit the threads after the boundary

Execution with Boundary Checks



Compilation



Function Declarations

- Keywords that differentiate where a function is intended to run

Keyword	Callable From	Executed On
__host__ (default)	Host	Host
__global__	Host (or Device)	Device
__device__	Device	Device

Function Declarations

- Keywords that differentiate where a function is intended to run

Keyword	Callable From	Executed On
__host__ (default)	Host	Host
__global__	Host (or Device)	Device
__device__	Device	Device

- Why need __host__ if it is the default?

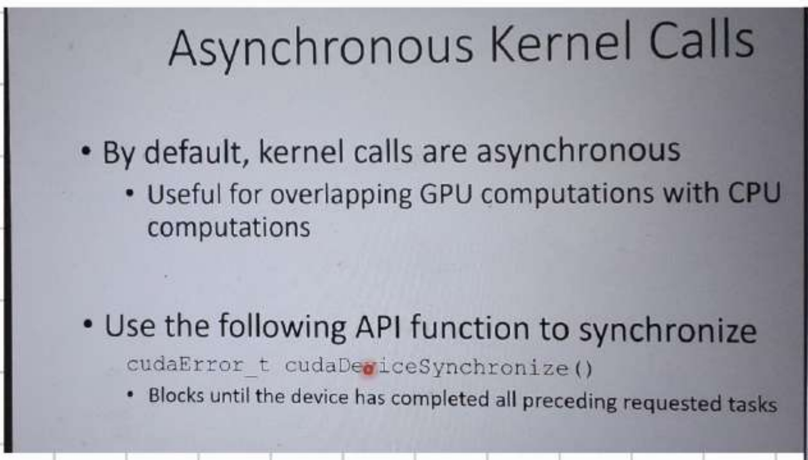
```
#include "timer.h"

__host__ __device__ float f(float a, float b) {
    return a + b;
}

void vecadd_cpu(float* x, float* y, float* z, int N) {
    for(unsigned int i = 0; i < N; ++i) {
        z[i] = f(x[i], y[i]);
    }
}

__global__ void vecadd_kernel(float* x, float* y, float* z, int N) {
    unsigned int i = blockDim.x*blockIdx.x + threadIdx.x;
    if(i < N) {
        z[i] = f(x[i], y[i]);
    }
}
```

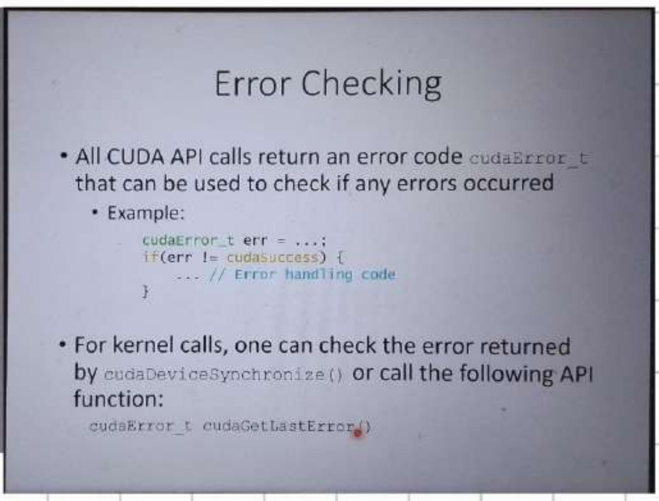
→ We can use this function is both CPU and GPU.

Function Declarations

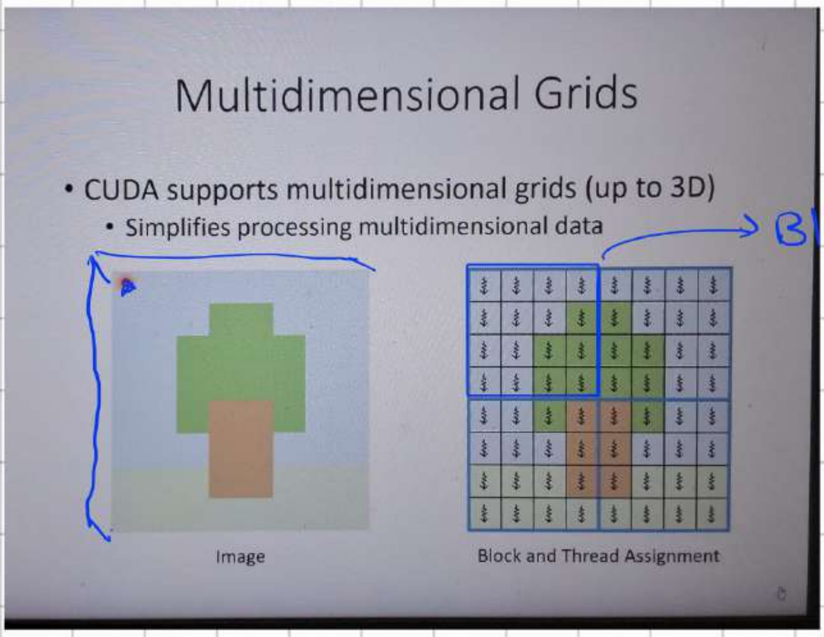
- The keyword __host__ is useful when needing to mark a function as executable on both the host and the device

```
__host__ __device__ float f(float a, float b) {
    return a + b;
}
void vecadd(float* x, float* y, float* z, int N) {
    for(unsigned int i = 0; i < N; ++i) {
        z[i] = f(x[i], y[i]);
    }
}
__global__ void vecadd_kernel(float* x, float* y, float* z, int N) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < N) {
        z[i] = f(x[i], y[i]);
    }
}
```

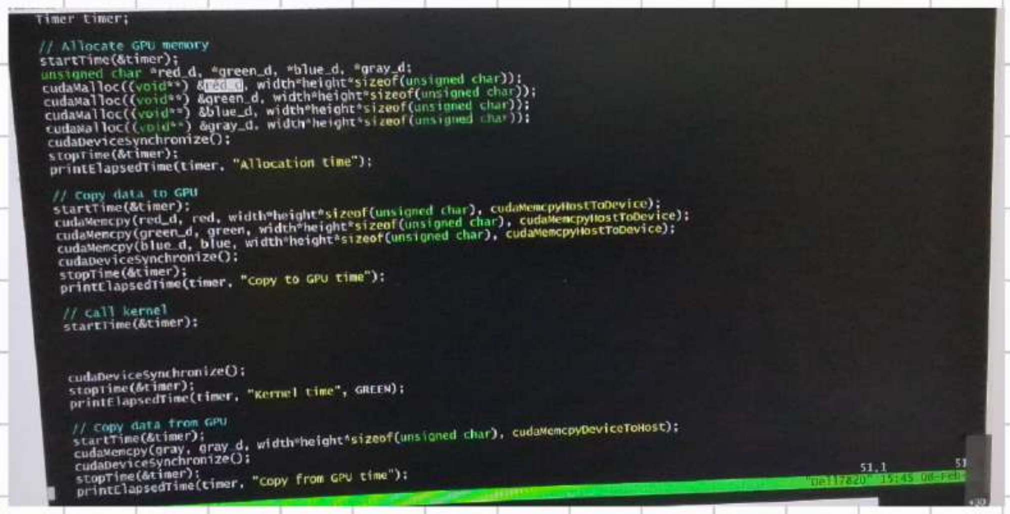




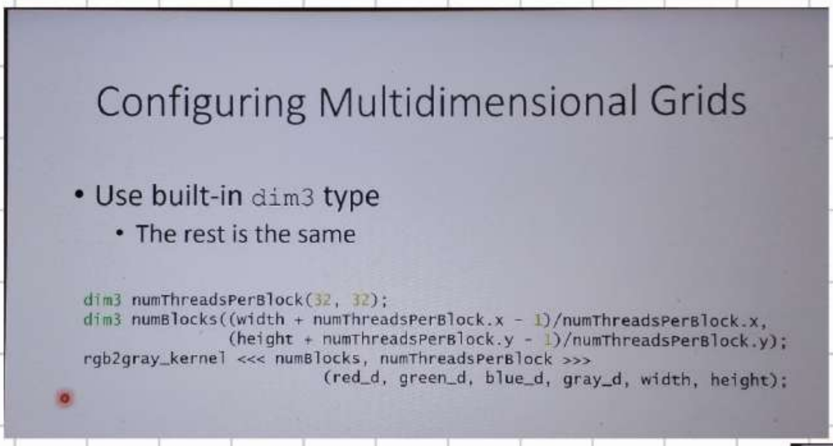
LECTURE 3 - Multi Dimensional GRIDS and Data.

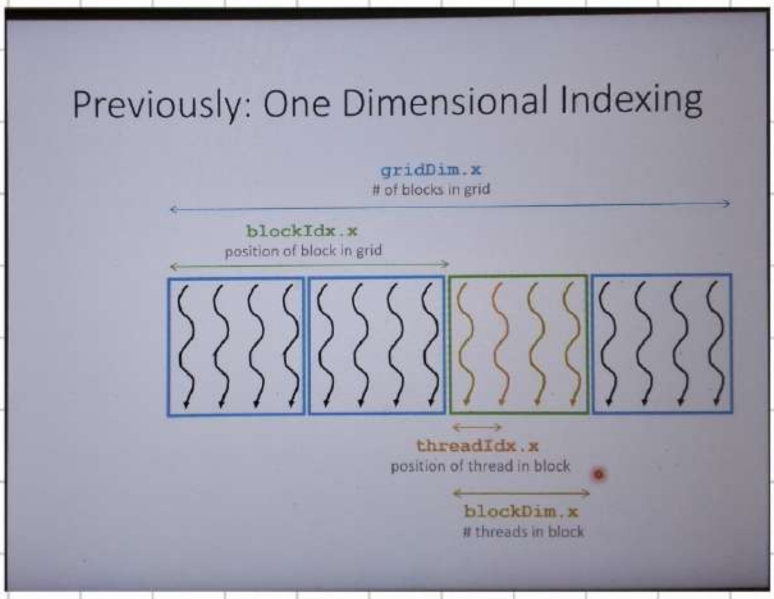


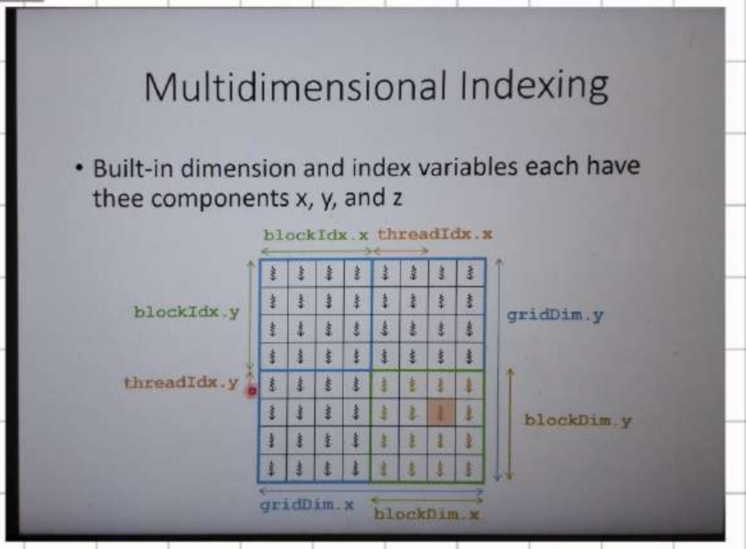
Block.



```
Timer timer;

// Allocate GPU memory
startTime(&timer);
unsigned char *red_d, *green_d, *blue_d, *gray_d;
cudaMalloc((void**) &red_d, width*height*sizeof(unsigned char));
cudaMalloc((void**) &green_d, width*height*sizeof(unsigned char));
cudaMalloc((void**) &blue_d, width*height*sizeof(unsigned char));
cudaMalloc((void**) &gray_d, width*height*sizeof(unsigned char));
cudaDeviceSynchronize();
stopTime(&timer);
printElapsedTime(timer, "Allocation time");

// Copy data to GPU
startTime(&timer);
cudaMemcpy(red_d, red, width*height*sizeof(unsigned char), cudaMemcpyHostToDevice);
cudaMemcpy(green_d, green, width*height*sizeof(unsigned char), cudaMemcpyHostToDevice);
cudaMemcpy(blue_d, blue, width*height*sizeof(unsigned char), cudaMemcpyHostToDevice);
cudaDeviceSynchronize();
stopTime(&timer);
printElapsedTime(timer, "Copy to GPU time");

// Call kernel
startTime(&timer);


cudaDeviceSynchronize();
stopTime(&timer);
printElapsedTime(timer, "Kernel time", GREEN);

// Copy data from GPU
startTime(&timer);
cudaMemcpy(gray, gray_d, width*height*sizeof(unsigned char), cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
stopTime(&timer);
printElapsedTime(timer, "Copy from GPU time");
```

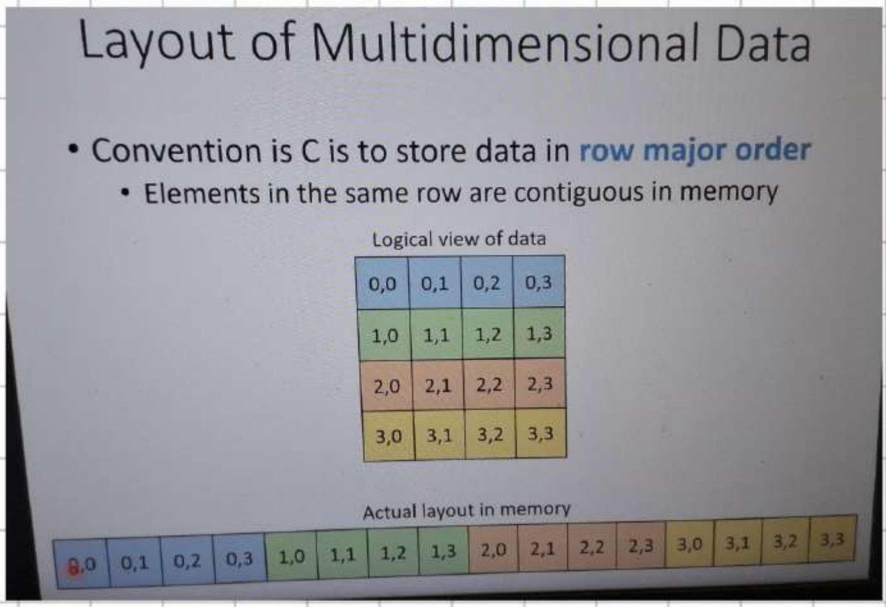




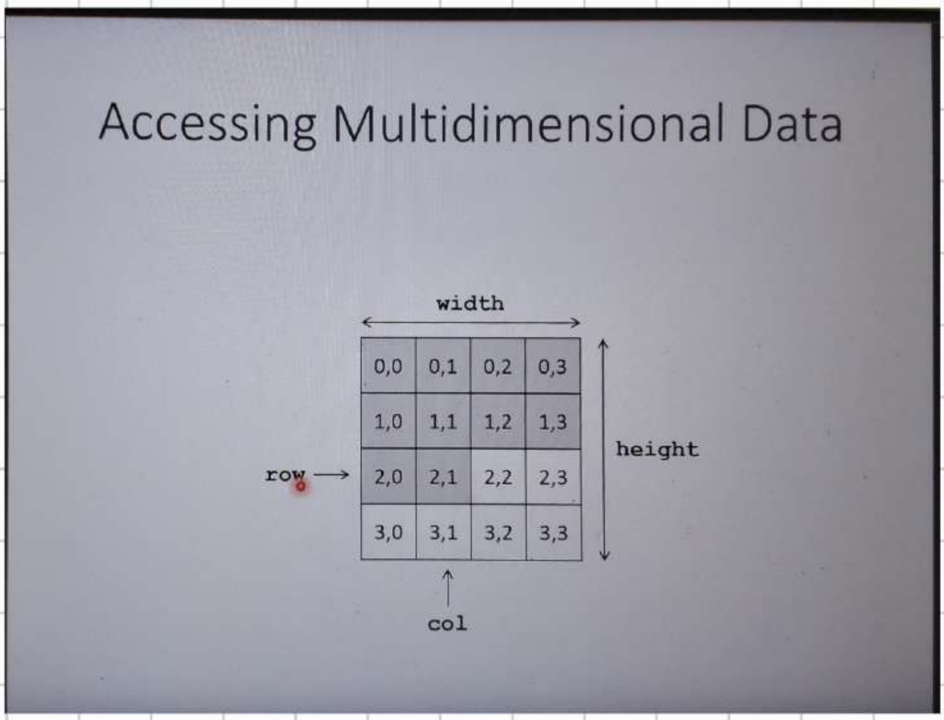


blockDim.x = 4   blockDim.y = 4

blockIdx.x * blockDim.x + threadIdx

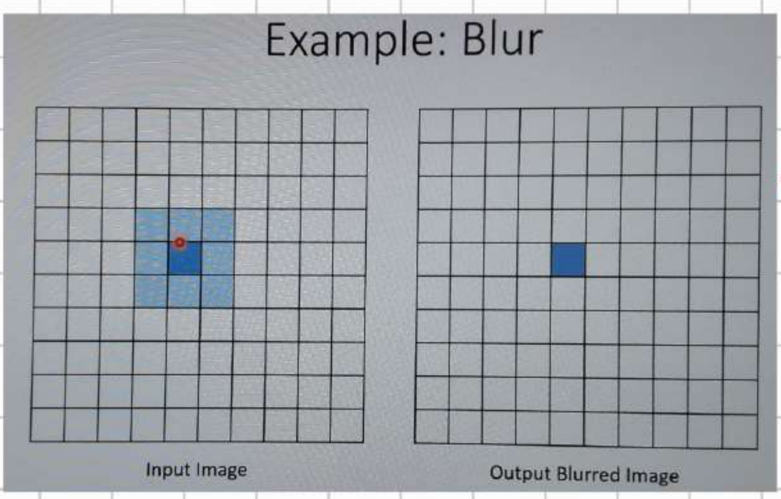


So Now the Question is How I will Convert My Row and Column and Convert such that it Converts into a Row Major Array.



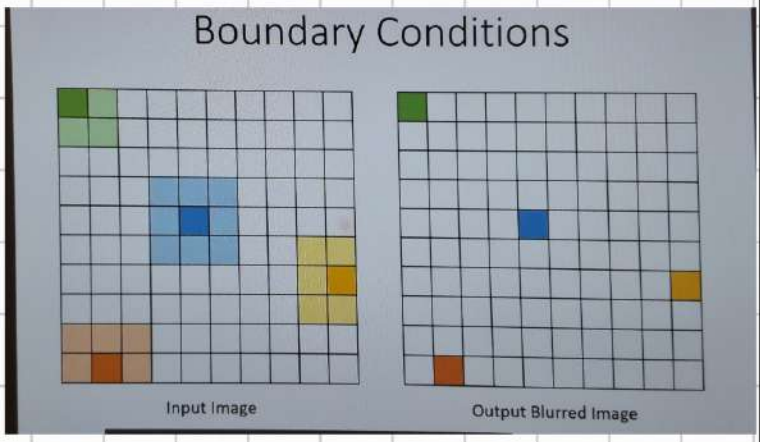
Row * Width + Column.

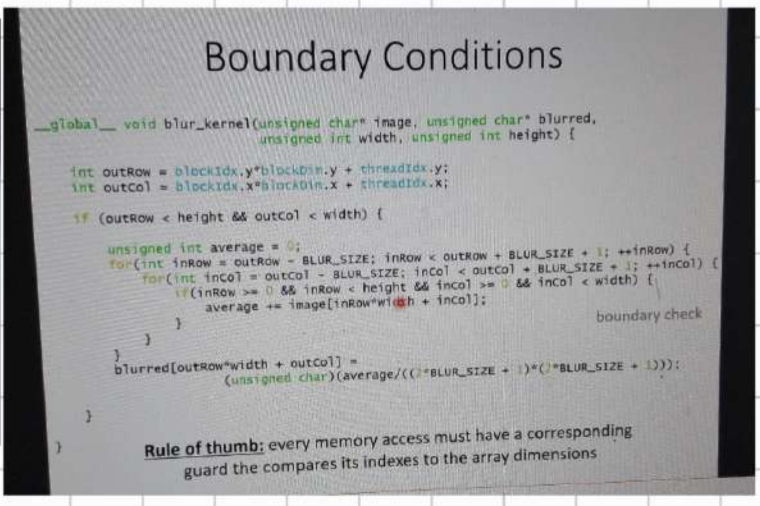
Example: Blur



☆ We can think of thread as being responsible for an Output Pixel. So what we can do is we can have a thread for every Output Pixel. and each thread is responsible for finding the Corresponding input Pixel and looping over the Surrounding Pixels. and Computing the Average.

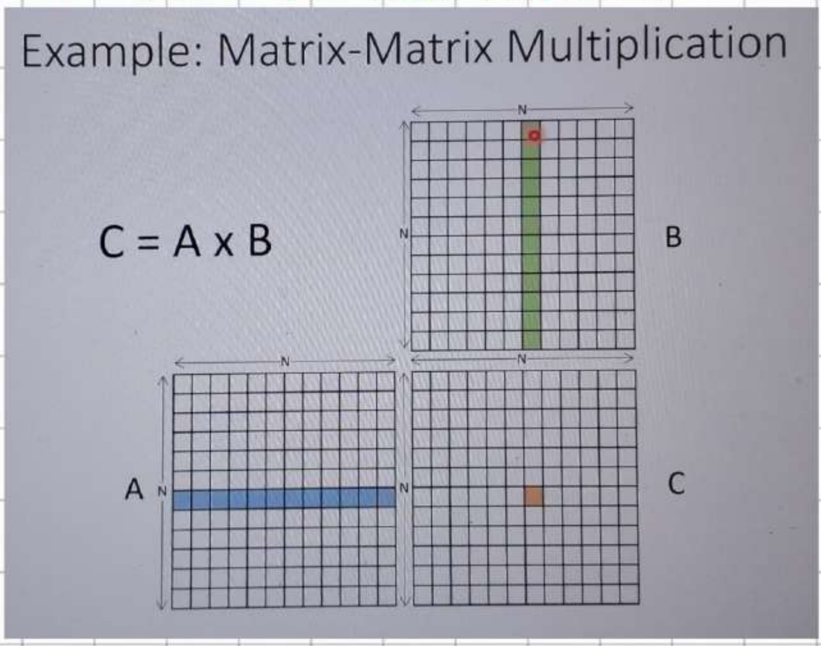
there may be more great Approach for solving this Problem

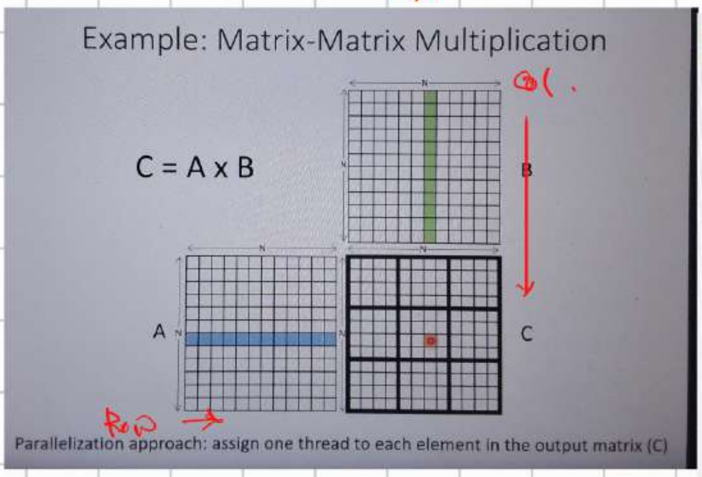


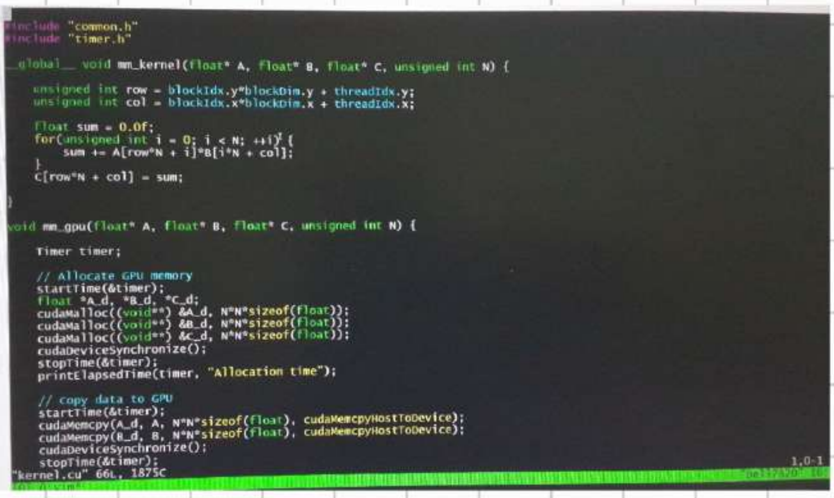


☆ One of the key thing Rule of thumb is for boundary Condition is every time you have a Memory Access. You should have a Corresponding guard that checks if the index of the axis is in bounds.

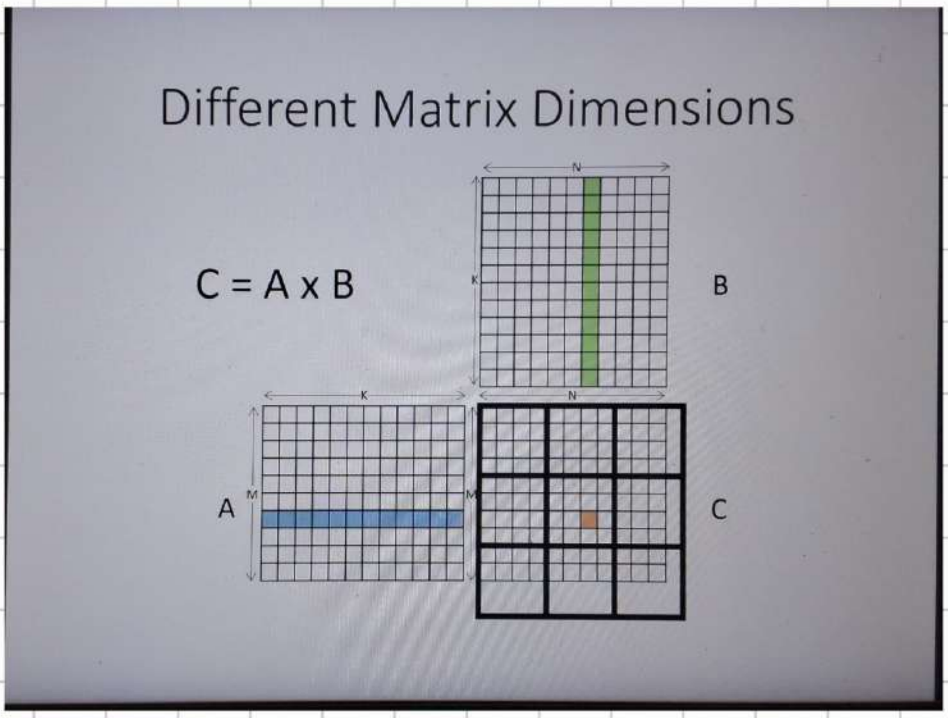
Example: MATRIX MATRIX Multiplication







```
#include "common.h"
#include "timer.h"

__global__ void mm_kernel(float* A, float* B, float* C, unsigned int N) {

    unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;

    float sum = 0.0f;
    for(unsigned int i = 0; i < N; ++i) {
        sum += A[row*N + i]*B[i*N + col];
    }
    C[row*N + col] = sum;

}

void mm_gpu(float* A, float* B, float* C, unsigned int N) {

    Timer timer;

    // Allocate GPU memory
    startTime(&timer);
    float *A_d, *B_d, *C_d;
    cudaMalloc((void**) &A_d, N*N*sizeof(float));
    cudaMalloc((void**) &B_d, N*N*sizeof(float));
    cudaMalloc((void**) &C_d, N*N*sizeof(float));
    cudaDeviceSynchronize();
    stopTime(&timer);
    printElapsedTime(timer, "Allocation time");

    // copy data to GPU
    startTime(&timer);
    cudaMemcpy(A_d, A, N*N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, N*N*sizeof(float), cudaMemcpyHostToDevice);
    cudaDeviceSynchronize();
    stopTime(&timer);
"kernel.cu" 66L, 1875C                                                   1,0-1
```



Assignments.