

Types of Parallelism:

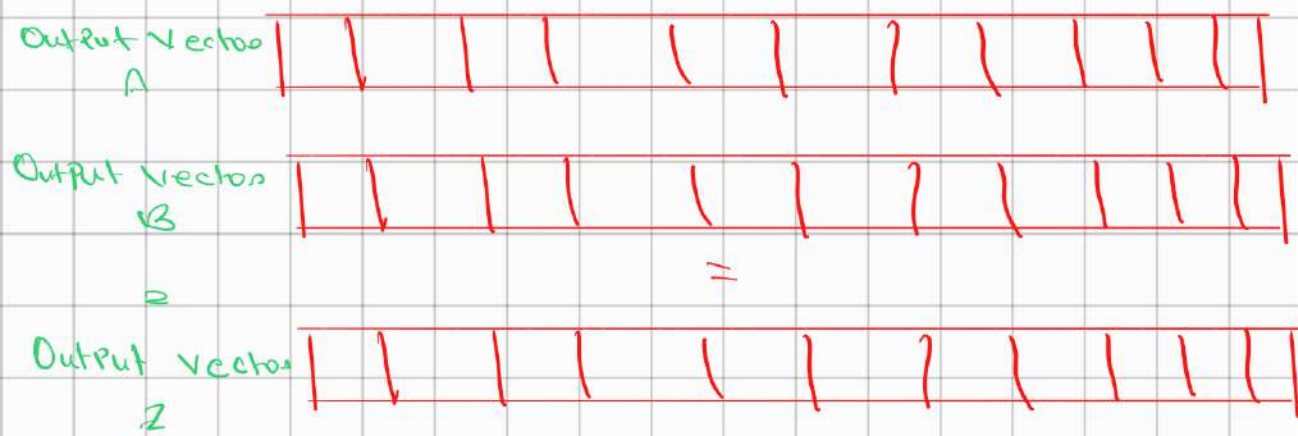
Task Parallelism

- 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);

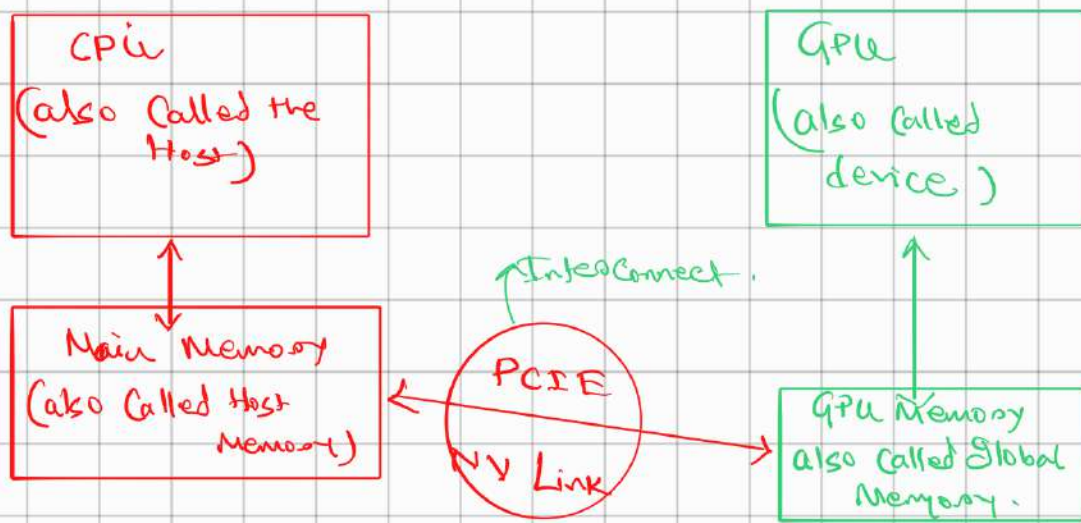
    // Vector addition on GPU
    startTime(&timer);
    vecadd_gpu(x, y, z, N);
    stopTime(&timer);
    printElapsedTime(timer, "GPU time", DGREEN);

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

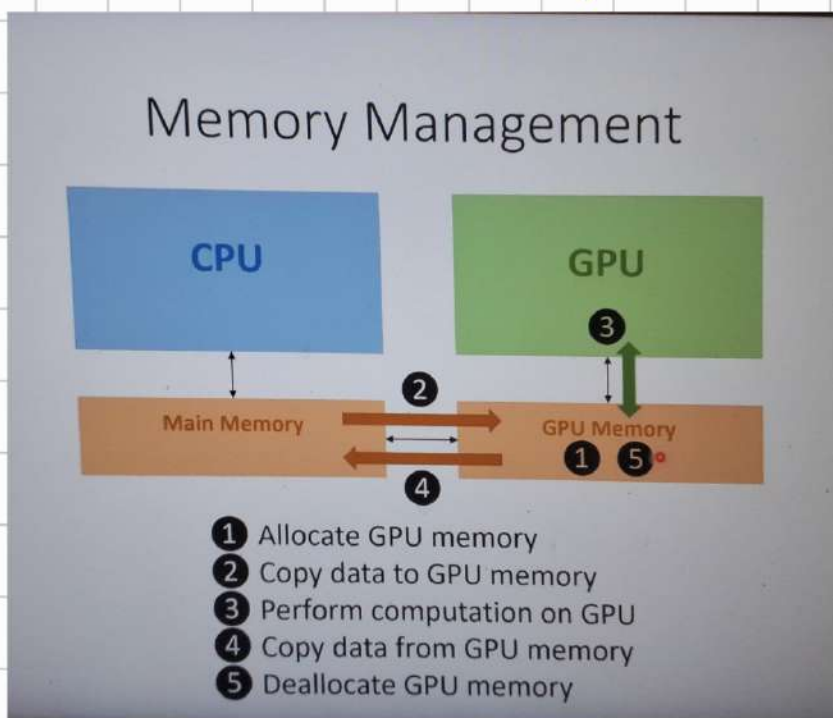
    return 0;
}
```

going to implement vec

System Organization:-



The CPU and GPU have separate 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 allocated device memory
 - size: Requested allocation size in bytes
- Deallocating memory:

```
cudaError_t cudaFree(void *devPtr)
```

 - devPtr: Pointer to device memory to free
- Return type: `cudaError_t`
 - Helps with error checking (discussed later)

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);
```


CUDA Memory Management API

• Copying memory:

```
cudaError_t cudaMemcpy(void *dst, const void *src,
                        size_t count, enum cudaMemcpyKind kind)
```

- dst: Destination memory address
- src: Source memory address
- count: Size in bytes to copy
- kind: Type of transfer
 - cudaMemcpyHostToHost
 - cudaMemcpyHostToDevice
 - cudaMemcpyDeviceToHost
 - cudaMemcpyDeviceToDevice

// Copy to the GPU.

`cudaMemcpy(x_d, x, N * sizeof(float), cudaMemcpyHostToDevice);`

`cudaMemcpy(y_d, y, N * sizeof(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

`VecAddKernel<< numBlocks, numThreadsPerBlock >>> (x_d, y_d, z_d, N);`

`CudaDeviceSynchronize();`

// Copy from GPU to CPU.

`cudaMemcpy(z, z_d, N * sizeof(float), cudaMemcpyDeviceToHost);`

IF I have N elements in my vector that means I want to have N threads in total.

// Deallocate GPU memory

`CudaFree(x_d);`

`CudaFree(y_d);`

`CudaFree(z_d);`

Code Example

```
void vecadd(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));
    cudaMalloc((void**) &y_d, N*sizeof(float));
    cudaMalloc((void**) &z_d, N*sizeof(float));

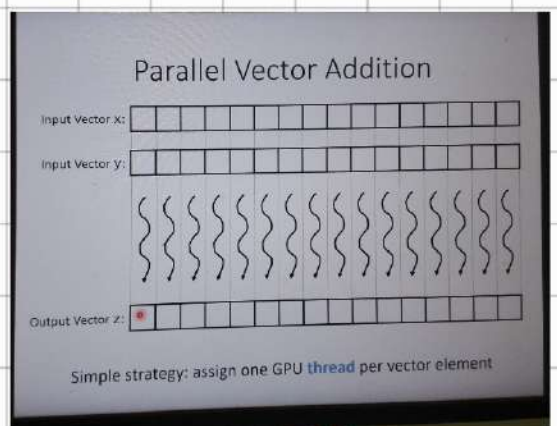
    // Copy data to GPU memory
    cudaMemcpy(x_d, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(y_d, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform computation on GPU
    ...

    // Copy data from GPU memory
    cudaMemcpy(z, z_d, N*sizeof(float), cudaMemcpyDeviceToHost);

    // Deallocate GPU memory
    cudaFree(x_d);
    cudaFree(y_d);
    cudaFree(z_d);

}
```

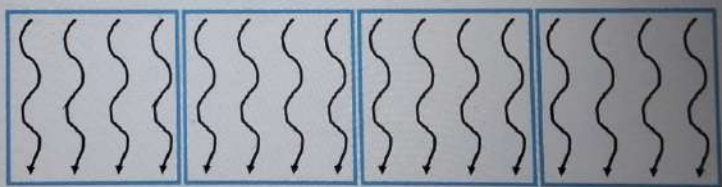


How threads on the GPU are organized?

→ 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.

Blocks



Threads in a grid are grouped into thread **blocks**

(significance: threads in the same block can collaborate in ways that threads in different blocks cannot – discussed later)

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 number of the blocks size - number of threads in each block.

Launching a Grid

- Threads in the same grid execute the same function known as a **kernel**
- A grid can be launched by calling a kernel and configuring it with appropriate grid and block sizes

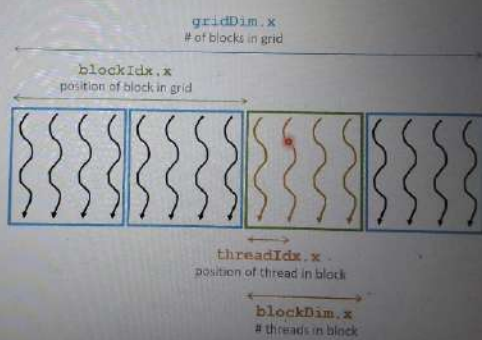
• Example:

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

Implementing a Kernel

- A kernel is similar to a C/C++ function
- It is preceded by the keyword `__global__` to indicate that it is a GPU kernel
- It uses special keywords to distinguish different threads from each other

Thread Index



```
__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.
```

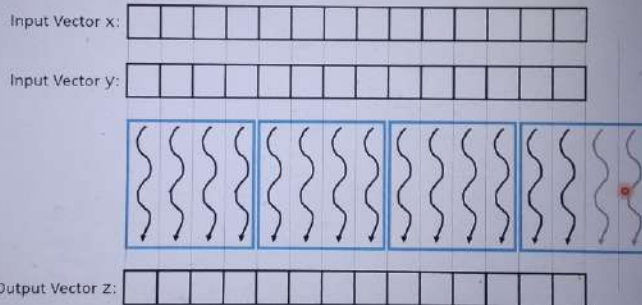

Boundary Conditions

- Recall launch configurations:

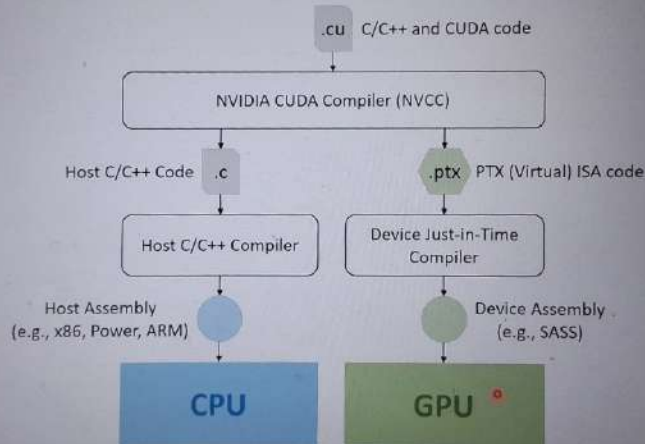
```
const unsigned int numThreadsPerBlock = 512;
const unsigned int numBlocks = N/numThreadsPerBlock;
vecadd_kernel <<< numBlocks, numThreadsPerBlock >>> (x_d, y_d, z_d, N);
```

- 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 |
|------------------------------------|---------------------|-------------|
| <code>__host__</code> (default) | Host | Host |
| <code>__global__</code> | Host (or Device) | Device |
| <code>__device__</code> | Device | Device |

Function Declarations

- Keywords that differentiate where a function is intended to run

| Keyword | Callable From | Executed On |
|------------------------------------|---------------------|-------------|
| <code>__host__</code> (default) | Host | Host |
| <code>__global__</code> | Host (or Device) | Device |
| <code>__device__</code> | 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 on 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]);
    }
}
```


Asynchronous Kernel Calls

- By default, kernel calls are asynchronous
 - Useful for overlapping GPU computations with CPU computations
- Use the following API function to synchronize
 - `cudaError_t cudaDeviceSynchronize()`
 - Blocks until the device has completed all preceding requested tasks

Error Checking

- All CUDA API calls return an error code `cudaError_t` that can be used to check if any errors occurred
 - Example:

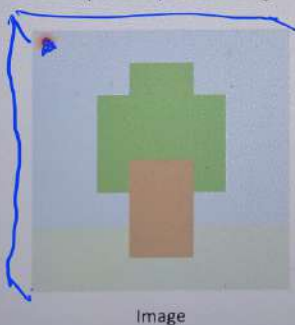
```
cudaError_t err = ...;
if(err != cudaSuccess) {
    ... // Error handling code
}
```
- For kernel calls, one can check the error returned by `cudaDeviceSynchronize()` or call the following API function:

```
cudaError_t cudaGetLastError()
```

LECTURE 3r Multi Dimensional Grids and Data.

Multidimensional Grids

- CUDA supports multidimensional grids (up to 3D)
 - Simplifies processing multidimensional data



Image



Block and Thread Assignment

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);

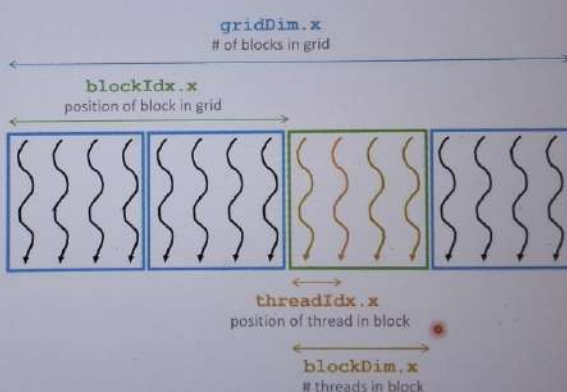
// Copy data from GPU
startTime(&timer);
cudaMemcpy(gray, gray_d, width*height*sizeof(unsigned char), cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
stopTime(&timer);
printElapsedTime(timer, "copy from GPU time");
```

Configuring Multidimensional Grids

- Use built-in `dim3` type
 - The rest is the same

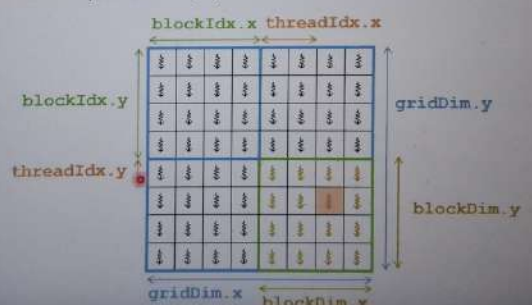
```
dim3 numThreadsPerBlock(32, 32);
dim3 numBlocks((width + numThreadsPerBlock.x - 1)/numThreadsPerBlock.x,
               (height + numThreadsPerBlock.y - 1)/numThreadsPerBlock.y);
rgb2gray_kernel <<< numBlocks, numThreadsPerBlock >>>
               (red_d, green_d, blue_d, gray_d, width, height);
```

Previously: One Dimensional Indexing



Multidimensional Indexing

- Built-in dimension and index variables each have three components x, y, and z



$blockDim.x = 7$ $blockDim.y = 9$

blockIdx.x * blockDim.x + threadIdx.x

Layout of Multidimensional Data

- Convention is C is to store data in **row major order**
 - Elements in the same row are contiguous in memory

Logical view of data

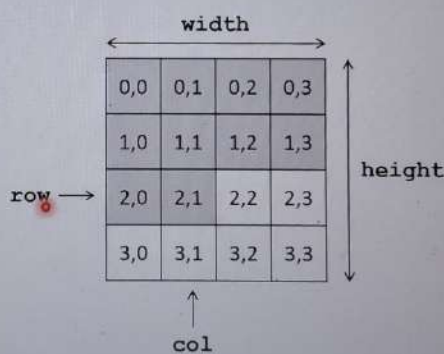
| | | | |
|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 |
| 1,0 | 1,1 | 1,2 | 1,3 |
| 2,0 | 2,1 | 2,2 | 2,3 |
| 3,0 | 3,1 | 3,2 | 3,3 |

Actual layout in memory

| | | | | | | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 | 1,0 | 1,1 | 1,2 | 1,3 | 2,0 | 2,1 | 2,2 | 2,3 | 3,0 | 3,1 | 3,2 | 3,3 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

So Now the Question is How I will convert my Row and Column and convert such that it converts into a Row Major Array.

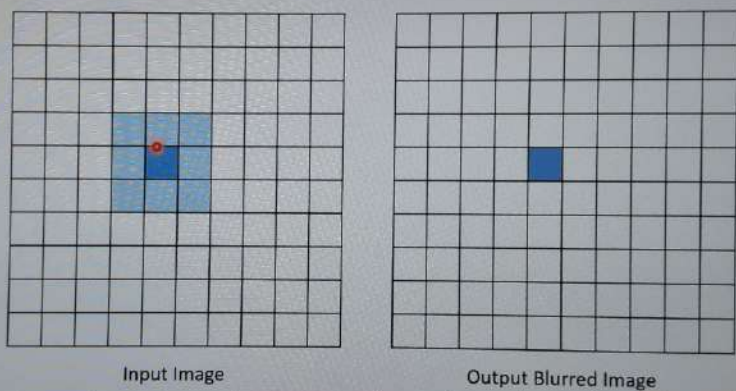
Accessing Multidimensional Data



Row * Width + Column

Example: Blur

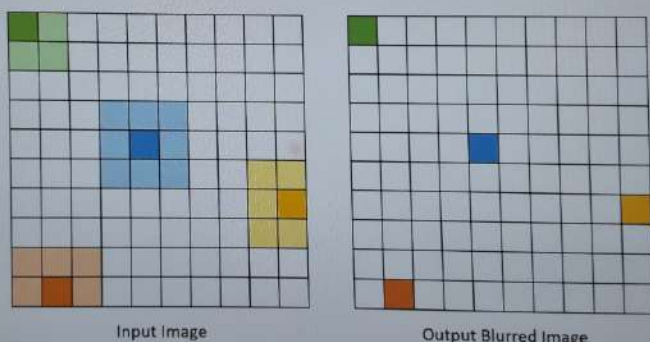
Example: Blur



★ 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

Boundary Conditions



Boundary Conditions

```
__global__ void blur_kernel(unsigned char* image, unsigned char* blurred,
                           unsigned int width, unsigned int height) {

    int outrow = blockIdx.y * blockDim.y + threadIdx.y;
    int outcol = blockIdx.x * blockDim.x + threadIdx.x;

    if (outrow < height && outcol < width) {

        unsigned int average = 0;
        for(int inrow = outrow - BLUR_SIZE; inrow < outrow + BLUR_SIZE + 1; ++inrow) {
            for(int incol = outcol - BLUR_SIZE; incol < outcol + BLUR_SIZE + 1; ++incol) {
                if(inrow >= 0 && inrow < height && incol >= 0 && incol < width) {
                    average += image[inrow*width + incol];
                }
            }
        }
        blurred[outrow*width + outcol] =
            (unsigned char)(average/((BLUR_SIZE + 1)*(BLUR_SIZE + 1)));
    }
}
```

Rule of thumb: every memory access must have a corresponding guard that compares its indexes to the array dimensions

22

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

