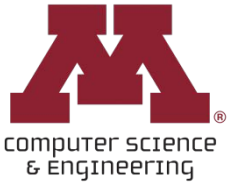
CSCI 5451: Introduction to Parallel Computing

Lecture 18: Introduction to CUDA



University of Minnesota
Driven to Discover®

Announcements (11/03)

- HW2 due yesterday
- HW3 out tomorrow
  - Builds on HW2
  - Block-Cyclic Cannon's algorithm
  - Sparse matrix multiplication
  - Implementation of variable_cannon.c will be released on Thursday

Lecture Overview

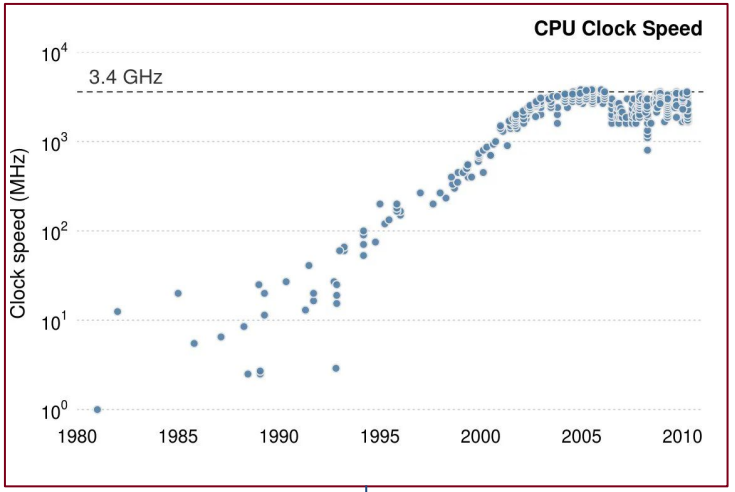
- CUDA Background
- Vector Addition with CUDA
- Multidimensional Grids
  - Gray Scale Conversion
  - Image Blurring

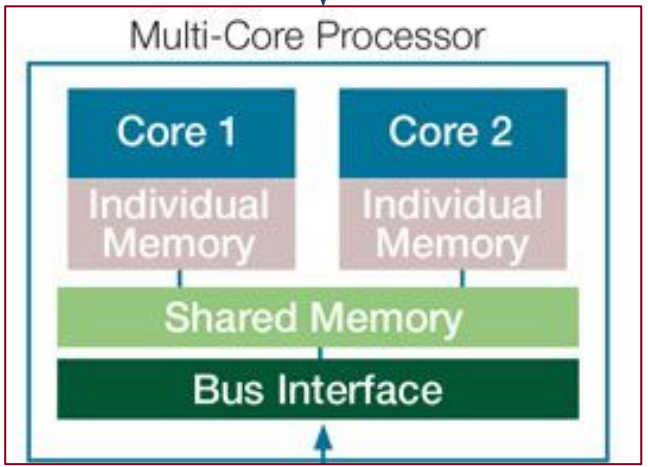
Lecture Overview

- **CUDA Background**
- Vector Addition with CUDA
- Multidimensional Grids
  - Gray Scale Conversion
  - Image Blurring

Faster Machines

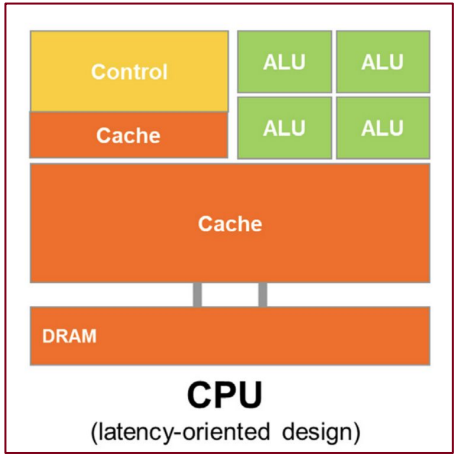
- Slowing clock rates meant that multicore systems became more appealing
- Shared memory systems with 2,4,8,etc. cores led to significant improvements
- But they had a drawback - they require there to be *independent cores*
- Each core implements a full instruction set, has caches, significantly complex pipelining, etc.

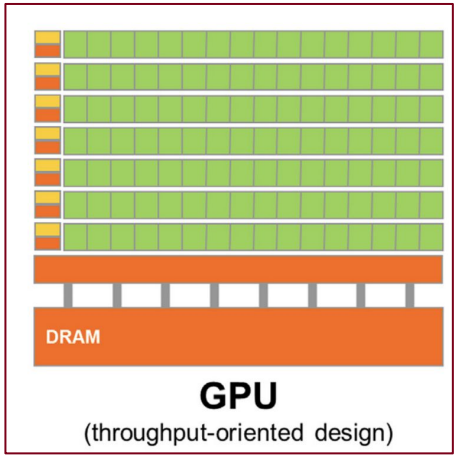




Multicore to GPU many-threading

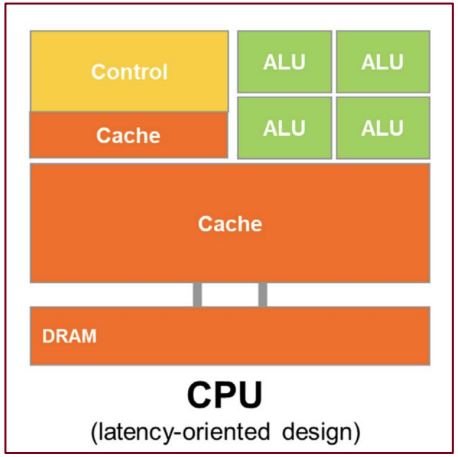
- Multicore has upper bounds because we need an *immense* amount of logic to handle each independent thread of execution
- Multicore CPUs still rely on individual cores which better handle sequential computation
- GPUs *natively* use less logic, and focus less on program *latency* and more on program *throughput*
- GPUs use more *arithmetic execution units*, and more *memory access channels*

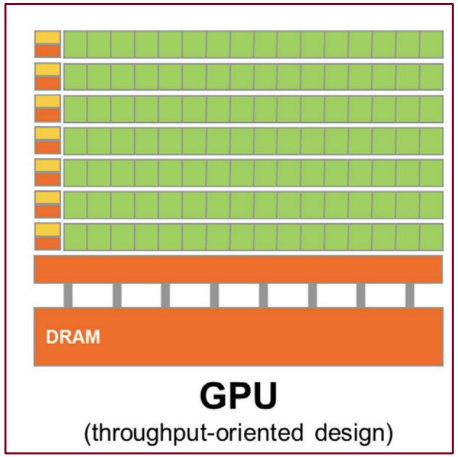




Multicore to GPU many-threading

- Multicore has upper bounds because we need an *immense* amount of logic to handle each independent thread of execution
- Multicore CPUs still rely on individual cores which better handle sequential computation
- GPUs *natively* use less logic, and focus less on program *latency* and more on program *throughput*
- **GPUs use more *arithmetic execution units*, and more *memory access channels***





FLOPs comparison

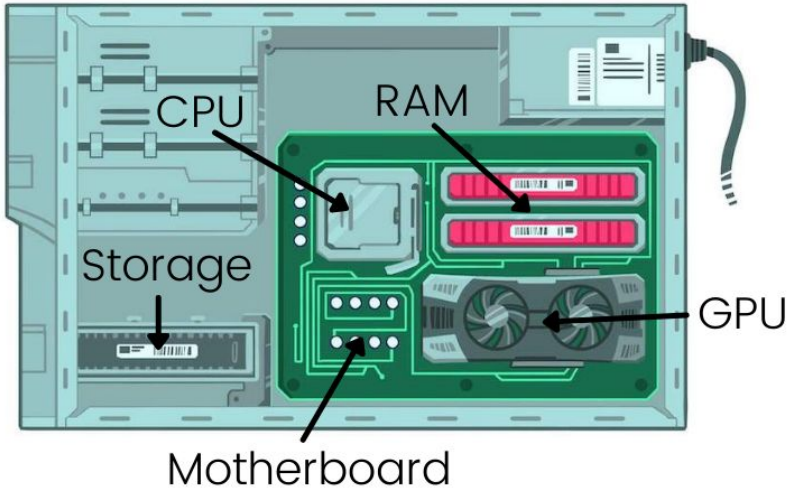
- Intel 24-core processor (Core i9-14900K)
  - .33 TFLOPs (double precision - 64bit)
  - .66 TFLOPs (single precision - 32bit)
- A100 GPU
  - 9.7 TFLOPs (double precision - 64bit)
  - 156 TFLOPs (single precision - 32bit)
  - 312 TFLOPs (half precision - 16bit)

GPU Origins

- Formed in graphics rendering with two requirements
  - Ensure that many computations can be performed in parallel for viewpoint transformations & object rendering
  - Move as much data as possible from DRAM to the arithmetic execution units to ensure a high fps
- The focus on the above two points differentiated NVIDIA and led to their success in graphics
- The above two points have further dovetailed with developments in cryptocurrencies & (more prominently) deep learning
- All these use cases require ***more compute throughput***

Where is the GPU?

- GPUs are a separate device which can be connected to a given CPU
- GPUs can be connected directly to a CPU on the same motherboard (either with a PCIe bus or with faster GPU-specific links)
- GPUs can be connected to a CPU indirectly via some networking hardware
- GPUs can also be connected to each other to create a larger *node* of gpus on their own motherboard



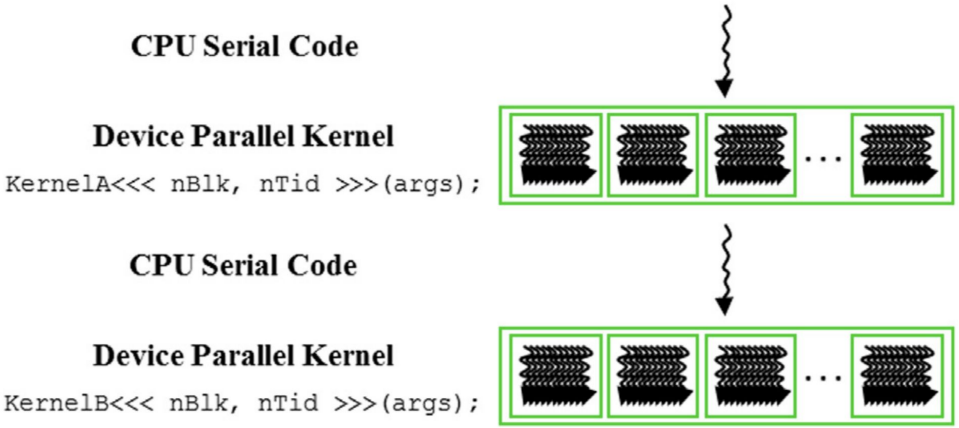
How do we program a GPU?

- The CUDA platform is the software that interfaces with the GPU
- Every C program can be compiled as a CUDA program - even if they do not use the GPU
- CUDA ***kernel*** → function which executes on the GPU
- A CUDA program starts executing on the ***host*** (CPU) and makes use of the ***device*** (GPU) when the kernel is ***invoked*** (called)
- Invoking a kernel will result in ***launching*** a large number of threads in parallel
- All the launched threads are called a ***grid***

Executing a simplified kernel

We see a simplified example of a program which executes two kernels

1. Some serial portion of code
2. The first kernel is invoked and creates ***nTid * nBlk*** threads
3. After finishing kernel execution, more serial code executes
4. The second kernel is invoked and creates ***nTid * nBlk*** threads



Lecture Overview

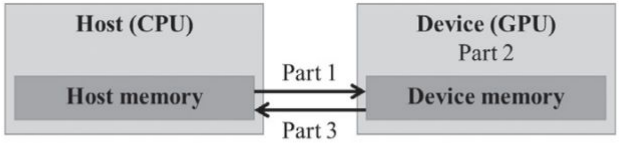
- CUDA Background
- **Vector Addition with CUDA**
- Multidimensional Grids
  - Gray Scale Conversion
  - Image Blurring

Vector Addition in CUDA

- Let's examine vector addition in CUDA
- To get started, let's first take a look at a serial version
- We want to take this program & execute it in parallel on our GPU

```
01  // Compute vector sum C_h = A_h + B_h
02  void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
03      for (int i = 0; i < n; ++i) {
04          C_h[i] = A_h[i] + B_h[i];
05      }
06  }
07  int main() {
08      // Memory allocation for arrays A, B, and C
09      // I/O to read A and B, N elements each
10      ...
11      vecAdd(A, B, C, N);
12  }
```

Vector Addition in CUDA



In order to create our kernel, we will have to perform three steps

- Copy any necessary memory over to the device (GPU)
- Get the device (GPU) to perform the necessary computation
- Copy the final result from the device (GPU) back to the host (CPU)

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

    // Part 1: Allocate device memory for A, B, and C
    // Copy A and B to device memory
    ...

    // Part 2: Call kernel – to launch a grid of threads
    // to perform the actual vector addition
    ...

    // Part 3: Copy C from the device memory
    // Free device vectors
    ...
}
```

Transferring 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 memory
  - **Pointer** to freed object

cudaMemcpy()

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

Transferring memory

- We have to allocate space on the device for the two input vectors (A_d, B_d) as well as the output vector (C_d)
- We have to copy the values from the input vectors on the host (A_h, B_h) onto the device vectors
- Once we are done computing, we need to copy the output vector from the device back onto the host
- We have to free all device structures to prevent any memory leaks

```
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

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

    cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Transferring memory

- We have to allocate space on the device for the two input vectors (A_d, B_d) as well as the output vector (C_d)
- We have to copy the values from the input vectors on the host (A_h, B_h) onto the device vectors
- Once we are done computing, we need to copy the output vector from the device back onto the host
- We have to free all device structures to prevent any memory leaks

```
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

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

    cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Transferring memory

- We have to allocate space on the device for the two input vectors (A_d, B_d) as well as the output vector (C_d)
- We have to copy the values from the input vectors on the host (A_h, B_h) onto the device vectors
- Once we are done computing, we need to copy the output vector from the device back onto the host
- We have to free all device structures to prevent any memory leaks

```
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

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

    cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Transferring memory

- We have to allocate space on the device for the two input vectors (A_d, B_d) as well as the output vector (C_d)
- We have to copy the values from the input vectors on the host (A_h, B_h) onto the device vectors
- Once we are done computing, we need to copy the output vector from the device back onto the host
- We have to free all device structures to prevent any memory leaks

```
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

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

    cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Transferring memory

- We have to allocate space on the device for the two input vectors (A_d, B_d) as well as the output vector (C_d)
- We have to copy the values from the input vectors on the host (A_h, B_h) onto the device vectors
- Once we are done computing, we need to copy the output vector from the device back onto the host
- We have to free all device structures to prevent any memory leaks

```
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

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

    cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Invoking a CUDA kernel

- CUDA kernels are launched in **blocks** of **threads**
- Each thread will operate in parallel
- The maximum number of threads in a given block is 1024
- The number of blocks to launch is given as the first argument inside of <<< >>>
- The number of threads to launch is given as the second argument inside of <<< >>>
- The arguments to the kernel function are supplied after <<< >>>

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

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Invoking a CUDA kernel

- CUDA kernels are launched in **blocks** of **threads**
- Each thread will operate in parallel
- The maximum number of threads in a given block is 1024
- The number of blocks to launch is given as the first argument inside of <<< >>>
- The number of threads to launch is given as the second argument inside of <<< >>>
- The arguments to the kernel function are supplied after <<< >>>

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

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Invoking a CUDA kernel

- CUDA kernels are launched in **blocks** of **threads**
- Each thread will operate in parallel
- The maximum number of threads in a given block is 1024
- The number of blocks to launch is given as the first argument inside of <<< >>>
- The number of threads to launch is given as the second argument inside of <<< >>>
- The arguments to the kernel function are supplied after <<< >>>

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

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Invoking a CUDA kernel

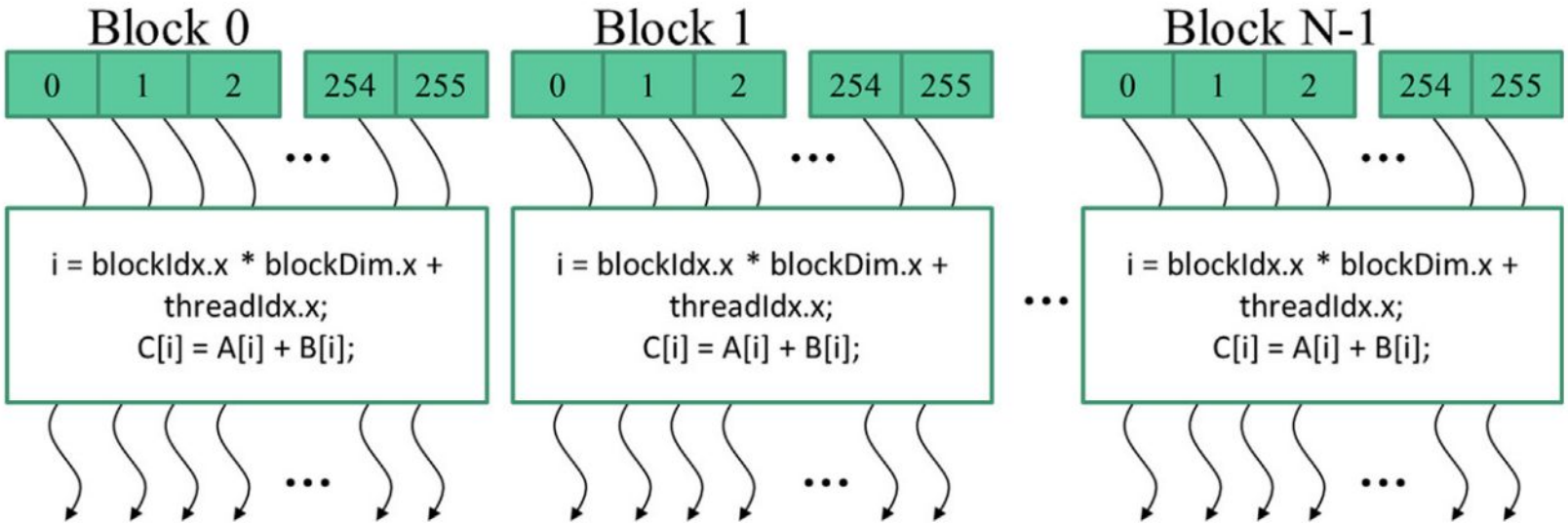
- CUDA kernels are launched in **blocks** of **threads**
- Each thread will operate in parallel
- The maximum number of threads in a given block is 1024
- The number of blocks to launch is given as the first argument inside of <<< >>>
- The number of threads to launch is given as the second argument inside of <<< >>>
- The arguments to the kernel function are supplied after <<< >>>

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

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

The CUDA Kernel

Each thread can get its unique id by determining

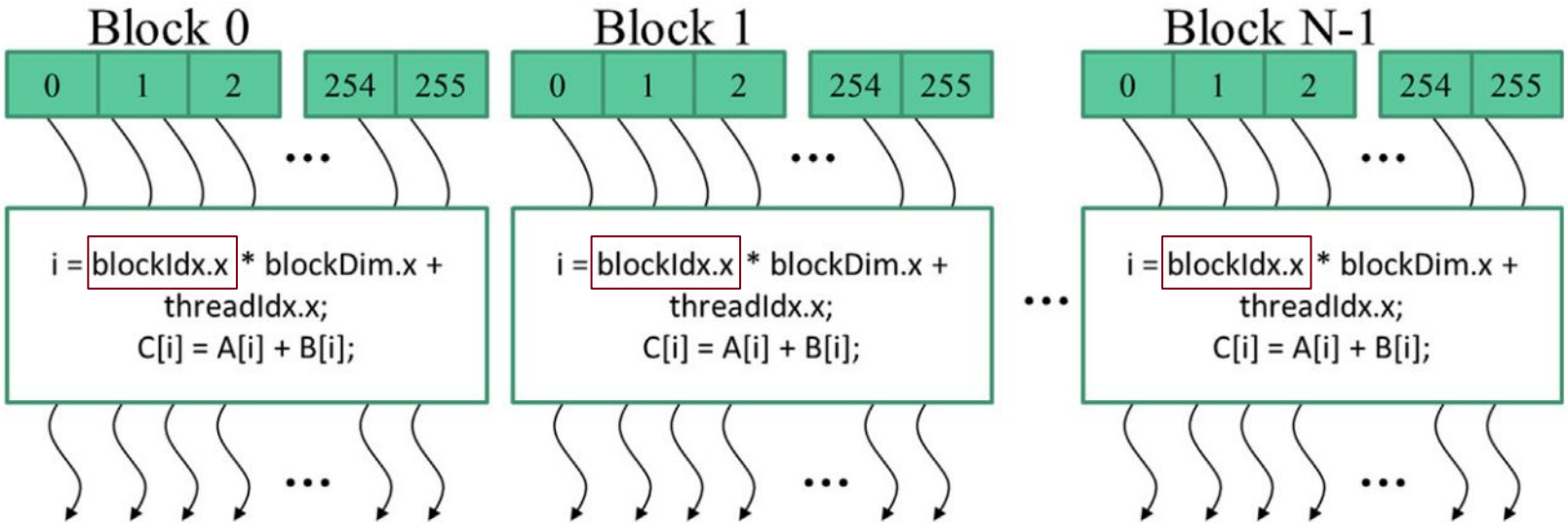
- ❏ Which block it is in
- ❏ How many threads are in each block
- ❏ What its index is within that given block



The CUDA Kernel

Each thread can get its unique id by determining

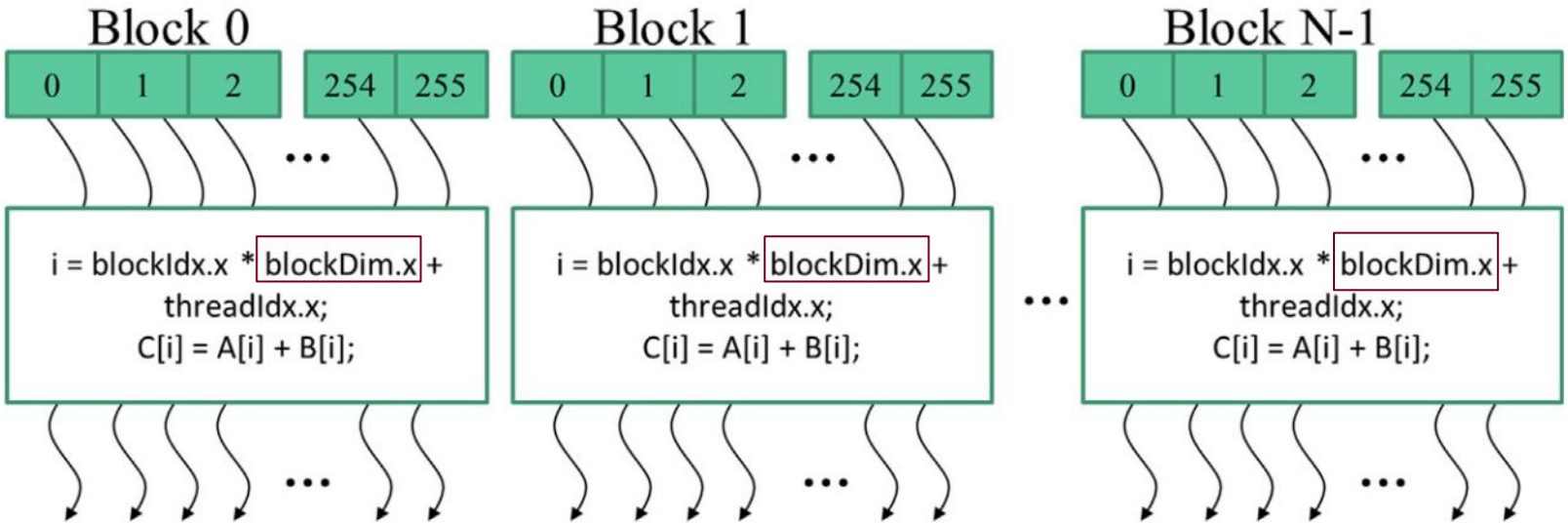
- **Which block it is in**
- How many threads are in each block
- What its index is within that given block



The CUDA Kernel

Each thread can get its unique id by determining

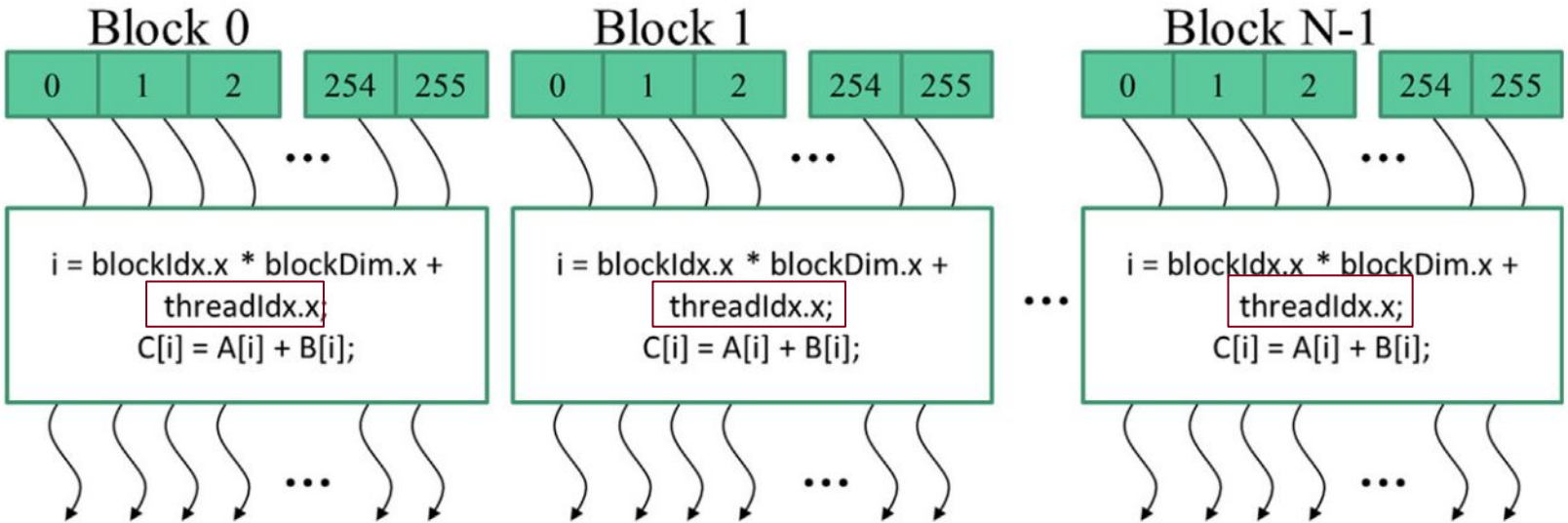
- ❑ Which block it is in
- ❑ **How many threads are in each block**
- ❑ What its index is within that given block



The CUDA Kernel

Each thread can get its unique id by determining

- ❑ Which block it is in
- ❑ How many threads are in each block
- ❑ **What its index is within that given block**



The CUDA Kernel

- Get unique thread index across grid
- Only execute if the thread index is less than the number of positions in C

```
// 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];
    }
}
```

The CUDA Kernel

- **Get unique thread index across grid**
- Only execute if the thread index is less than the number of positions in C

```
// 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];
    }
}
```

The CUDA Kernel

- Get unique thread index across grid
- **Only execute if the thread index is less than the number of positions in C**

```
// 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];
    }
}
```

The CUDA Kernel

Why is the if condition necessary?

- Get unique thread index across grid
- **Only execute if the thread index is less than the number of positions in C**

```
// 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];
    }
}
```

The CUDA Kernel

Why is the if condition necessary?

- We launch ***blocks*** of threads, not individual threads.

- Get unique thread index across grid
- **Only execute if the thread index is less than the number of positions in C**

```
// 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];
    }
}
```

Different Kernel Types

Qualifier Keyword	Callable From	Executed On	Executed By
__host__ (default)	Host	Host	Caller host thread
__global__	Host (or Device)	Device	New grid of device threads
__device__	Device	Device	Caller device thread

Full Vector Addition Example

- Some threads will be idle if the size of our array is not divisible by 256
- What happens if our array is enormous (e.g. 1e9)?

```
// 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];
    }
}
```

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

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Full Vector Addition Example

- Some threads will be idle if the size of our array is not divisible by 256
- What happens if our array is enormous (e.g. 1e9)?
  - Blocks of threads will be launched sequentially
  - We will see this in greater detail in the coming weeks

```
// 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];
    }
}
```

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

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Full Vector Addition Example

- Some threads will be idle if the size of our array is not divisible by 256
- What happens if our array is enormous (e.g. 1e9)?
  - Blocks of threads will be launched sequentially
  - We will see this in greater detail in the coming weeks
- Practical limitations of vector addition?

```
// 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];
    }
}
```

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

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Full Vector Addition Example

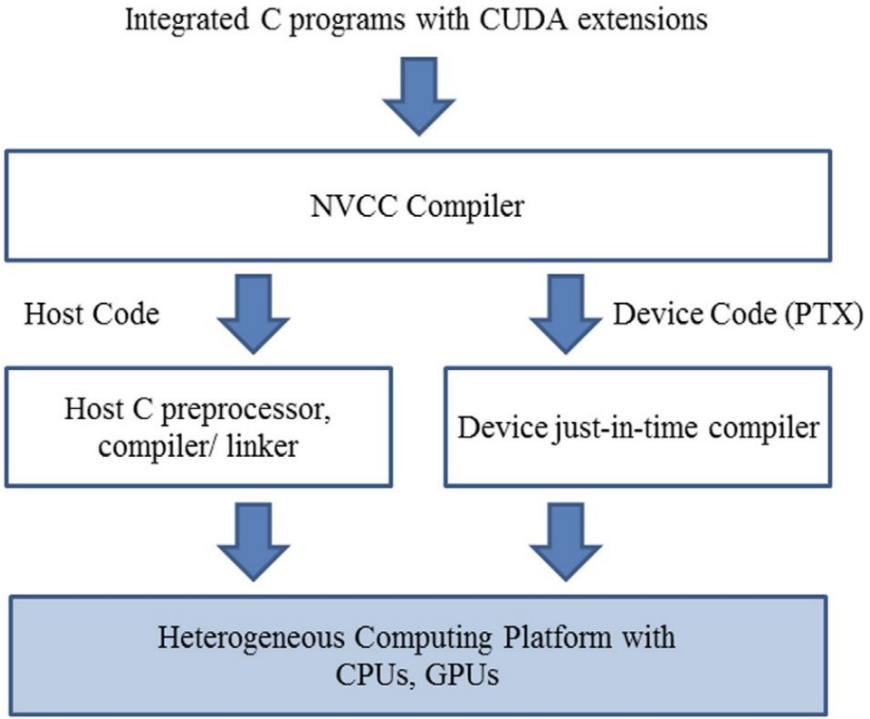
- Some threads will be idle if the size of our array is not divisible by 256
- What happens if our array is enormous (e.g. 1e9)?
  - Blocks of threads will be launched sequentially
  - We will see this in greater detail in the coming weeks
- Practical limitations of vector addition?
  - Vector addition speedups are limited by the time to transfer from host to device
  - CUDA works better when the compute intensity (how much we compute per each byte we load from memory) is very high
  - The compute intensity is only 1 in this example, so is unlikely to experience very good improvements

```
// 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];
    }
}
```

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

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Compiling a CUDA program

- CUDA programs make use of the NVCC compiler
- Files written to be cuda programs should have the .cu filetype (e.g. *VecAdd.cu*)
- Host code and device code are treated separately by the compiler
- We will dive into this process further in the coming weeks



Lecture Overview

- CUDA Background
- Vector Addition with CUDA
- **Multidimensional Grids**
  - **Gray Scale Conversion**
  - Image Blurring

Multidimensional Grids

- Our first example used only 1-dimensional grids of threads & blocks
- In other words, we could get the thread-specific identifier with *int i = threadIdx.x + blockDim.x * blockIdx.x*
- This 1-d output of threads was useful with vector addition, but 2-d & 3-d organizations of threads can be useful with image data & matrix computations

```
vecAddKernel<<<ceil(n/256.0), 256>>>(...);
```

```
// 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];
    }
}
```

Multidimensional Grids

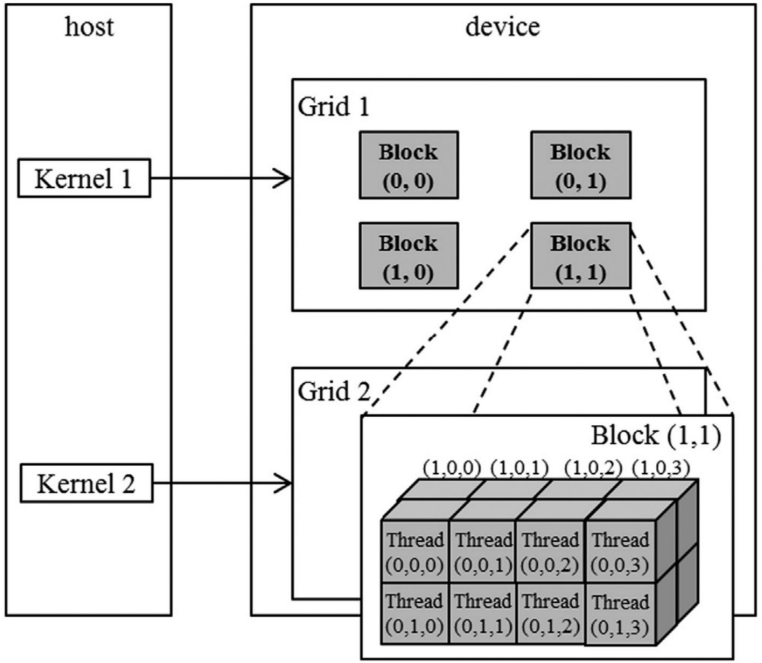
- Our first example used only 1-dimensional grids of threads & blocks
- In other words, we could get the thread-specific identifier with *int i = threadIdx.x + blockDim.x * blockIdx.x*
- This 1-d output of threads was useful with vector addition, but 2-d & 3-d organizations of threads can be useful with image data & matrix computations

```
vecAddKernel<<<ceil(n/256.0), 256>>>(...);
```

```
// 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];
    }
}
```

Creating Multidimensional Grids

- We can use the *dim3* data type to declare 3-dimensional grids of either blocks or threads
- The first dimension is *x*, the second is *y*, the third is *z*
  - Threads Ids → (*threadIdx.x, threadIdx.y, threadIdx.z*)
  - Dims → (*blockDim.x, blockDim.y, blockDim.z*)
  - Block Ids → (*blockIdx.x, blockIdx.y, blockIdx.z*)
- Kernel invocation accepts either *dim3* or *int* as input - when *int*, only the *x* direction is used
- The total number of threads still cannot exceed 1024 → (blockDim.x * blockDim.y * blockDim.z) <=1024




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

Grayscale Conversion

- We want to convert a *width* x *height* color image (*Pin*) into a grayscale image (*Pout*)
- Each thread will compute one output pixel
- The formula for grayscale conversion is presented below (this is applied at each pixel of the color image)

$$L = r*0.21 + g*0.72 + b*0.07$$

Grayscale Conversion Kernel

- Each thread first identifies which pixel it must compute the grayscale luminance value for
- If no such pixel value exists (boundary condition), the thread skips further execution
- The thread identifies the RGB values of the input image
- The thread computes the grayscale value

```
// The input image is encoded as unsigned chars [0, 255]
// Each pixel is 3 consecutive chars for the 3 channels (RGB)
__global__
void colortoGrayscaleConvertion(unsigned char * Pout,
                 unsigned char * Pin, int width, int height) {
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    if (col < width && row < height) {
        // Get 1D offset for the grayscale image
        int grayOffset = row*width + col;
        // One can think of the RGB image having CHANNEL
        // times more columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = Pin[rgbOffset    ]; // Red value
        unsigned char g = Pin[rgbOffset + 1]; // Green value
        unsigned char b = Pin[rgbOffset + 2]; // Blue value
        // Perform the rescaling and store it
        // We multiply by floating point constants
        Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

Grayscale Conversion Kernel

- **Each thread first identifies which pixel it must compute the grayscale luminance value for**
- If no such pixel value exists (boundary condition), the thread skips further execution
- The thread identifies the RGB values of the input image
- The thread computes the grayscale value

```
// The input image is encoded as unsigned chars [0, 255]
// Each pixel is 3 consecutive chars for the 3 channels (RGB)
__global__
void colortoGrayscaleConvertion(unsigned char * Pout,
                                unsigned char * Pin, int width, int height) {
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    if (col < width && row < height) {
        // Get 1D offset for the grayscale image
        int grayOffset = row*width + col;
        // One can think of the RGB image having CHANNEL
        // times more columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = Pin[rgbOffset    ]; // Red value
        unsigned char g = Pin[rgbOffset + 1]; // Green value
        unsigned char b = Pin[rgbOffset + 2]; // Blue value
        // Perform the rescaling and store it
        // We multiply by floating point constants
        Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

Grayscale Conversion Kernel

- Each thread first identifies which pixel it must compute the grayscale luminance value for
- **If no such pixel value exists (boundary condition), the thread skips further execution**
- The thread identifies the RGB values of the input image
- The thread computes the grayscale value

```
// The input image is encoded as unsigned chars [0, 255]
// Each pixel is 3 consecutive chars for the 3 channels (RGB)
__global__
void colortoGrayscaleConvertion(unsigned char * Pout,
                 unsigned char * Pin, int width, int height) {
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    if (col < width && row < height) {
        // Get 1D offset for the grayscale image
        int grayOffset = row*width + col;
        // One can think of the RGB image having CHANNEL
        // times more columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = Pin[rgbOffset    ]; // Red value
        unsigned char g = Pin[rgbOffset + 1]; // Green value
        unsigned char b = Pin[rgbOffset + 2]; // Blue value
        // Perform the rescaling and store it
        // We multiply by floating point constants
        Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

Grayscale Conversion Kernel

- Each thread first identifies which pixel it must compute the grayscale luminance value for
- If no such pixel value exists (boundary condition), the thread skips further execution
- **The thread identifies the RGB values of the input image**
- The thread computes the grayscale value

```
// The input image is encoded as unsigned chars [0, 255]
// Each pixel is 3 consecutive chars for the 3 channels (RGB)
__global__
void colortoGrayscaleConvertion(unsigned char * Pout,
                 unsigned char * Pin, int width, int height) {
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    if (col < width && row < height) {
        // Get 1D offset for the grayscale image
        int grayOffset = row*width + col;
        // One can think of the RGB image having CHANNEL
        // times more columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = Pin[rgbOffset    ]; // Red value
        unsigned char g = Pin[rgbOffset + 1]; // Green value
        unsigned char b = Pin[rgbOffset + 2]; // Blue value
        // Perform the rescaling and store it
        // We multiply by floating point constants
        Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

Grayscale Conversion Kernel

- Each thread first identifies which pixel it must compute the grayscale luminance value for
- If no such pixel value exists (boundary condition), the thread skips further execution
- The thread identifies the RGB values of the input image
- **The thread computes the grayscale value**

```
// The input image is encoded as unsigned chars [0, 255]
// Each pixel is 3 consecutive chars for the 3 channels (RGB)
__global__
void colortoGrayscaleConvertion(unsigned char * Pout,
                 unsigned char * Pin, int width, int height) {
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    if (col < width && row < height) {
        // Get 1D offset for the grayscale image
        int grayOffset = row*width + col;
        // One can think of the RGB image having CHANNEL
        // times more columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = Pin[rgbOffset    ]; // Red value
        unsigned char g = Pin[rgbOffset + 1]; // Green value
        unsigned char b = Pin[rgbOffset + 2]; // Blue value
        // Perform the rescaling and store it
        // We multiply by floating point constants
        Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

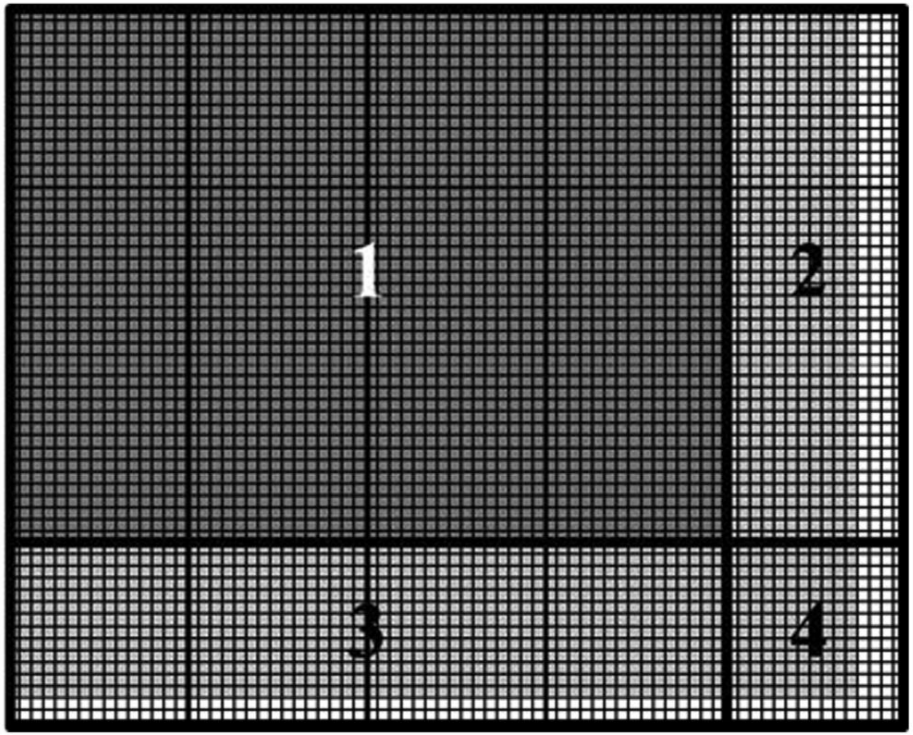
Invoking the Grayscale Conversion Kernel

- We need to launch enough threads to cover the entirety of the input image
- For an image with a width of *m* and a height of *n*, we can launch the below kernel
- We will have *16 * ceil(m/16)* threads in the x-direction to cover the width *m*
- We will have 16 * ceil(n/16) threads in the y-direction to cover the height *n*
- We do not need any z-dimension (our thread-decomposition is 2-d)

```
dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1);
dim3 dimBlock(16, 16, 1);
colorToGrayscaleConversion<<<dimGrid,dimBlock>>>
                                (Pin_d, Pout_d, m, n);
```

Excess threads

- We *always* have to handle the possibility of excess threads
- More often than not, the problem size will not be divisible by the number of threads in each block
- You have to ***explicitly*** account for this in your programs
- In the grayscale conversion example, a given block of threads can have all threads execute (1), some threads in only the x-direction compute nothing (2), some threads in the y-direction compute nothing (3) and some threads in both directions compute nothing (4)

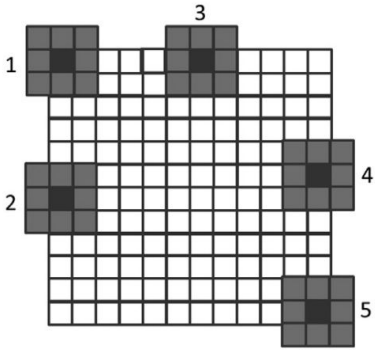


Lecture Overview

- CUDA Background
- Vector Addition with CUDA
- **Multidimensional Grids**
  - Gray Scale Conversion
  - **Image Blurring**

Image Blurring

- Image blurring involves computing the average of all surrounding pixels
- When the blur size is 1, then all pixels within radius 1 of the pixel are included in the average (this produces the example at the bottom right)
- When the pixel is at the edge of image, the average is computing by only considering actual pixels - points beyond the edge of the image are ignored



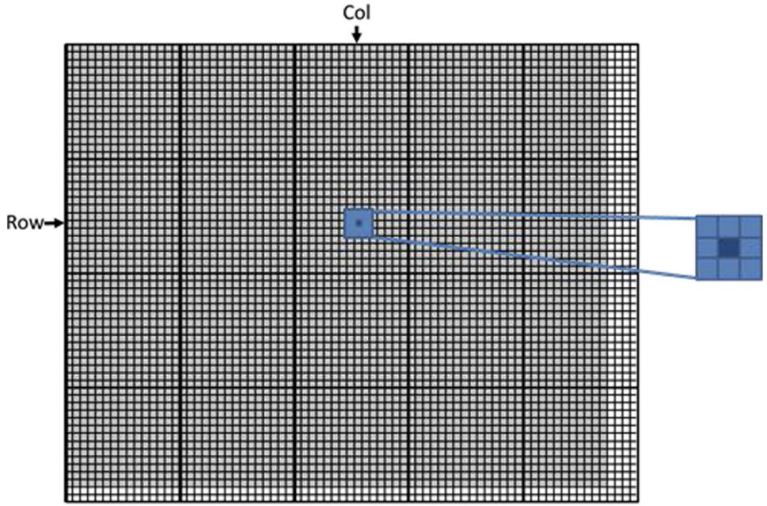


Image Blur Kernel

- As before - we compute the thread location
- If this thread's location is off the edge of the image, we skip computation
- We get the average over all surrounding locations
  - We calculate the index of a given pixel location
  - If this index is outside the boundaries of the image, we ignore it
  - We keep track of the number of nearby pixels & a running sum of the pixel values
- We compute the average pixel value

```
__global__
void blurKernel(unsigned char *in, unsigned char *out, int w, int h){
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  if(col < w && row < h) {
    int pixVal = 0;
    int pixels = 0;
        // Get average of the surrounding BLUR_SIZE x BLUR_SIZE box
    for(int blurRow=-BLUR_SIZE; blurRow<BLUR_SIZE+1; ++blurRow){
       for(int blurCol=-BLUR_SIZE; blurCol<BLUR_SIZE+1; ++blurCol){
          int curRow = row + blurRow;
          int curCol = col + blurCol;
                // Verify we have a valid image pixel
           if(curRow>=0 && curRow<h && curCol>=0 && curCol<w) {
               pixVal += in[curRow*w + curCol];
               ++pixels; // Keep track of number of pixels in the avg
            }
           }
        }
        // Write our new pixel value out
        out[row*w + col] = (unsigned char)(pixVal/pixels);
    }
}
```

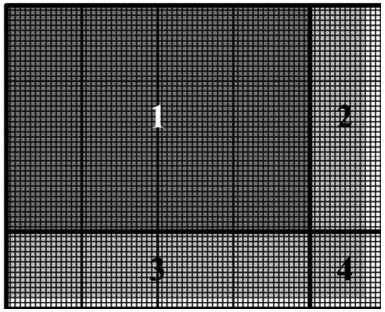
Image Blur Kernel

- **As before - we compute the thread location**
- If this thread's location is off the edge of the image, we skip computation
- We get the average over all surrounding locations
  - We calculate the index of a given pixel location
  - If this index is outside the boundaries of the image, we ignore it
  - We keep track of the number of nearby pixels & a running sum of the pixel values
- We compute the average pixel value

```
__global__
void blurKernel(unsigned char *in, unsigned char *out, int w, int h){
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  if(col < w && row < h) {
    int pixVal = 0;
    int pixels = 0;
        // Get average of the surrounding BLUR_SIZE x BLUR_SIZE box
    for(int blurRow=-BLUR_SIZE; blurRow<BLUR_SIZE+1; ++blurRow){
       for(int blurCol=-BLUR_SIZE; blurCol<BLUR_SIZE+1; ++blurCol){
          int curRow = row + blurRow;
          int curCol = col + blurCol;
                // Verify we have a valid image pixel
           if(curRow>=0 && curRow<h && curCol>=0 && curCol<w) {
               pixVal += in[curRow*w + curCol];
               ++pixels; // Keep track of number of pixels in the avg
            }
           }
        }
        // Write our new pixel value out
        out[row*w + col] = (unsigned char)(pixVal/pixels);
    }
}
```

Image Blur Kernel

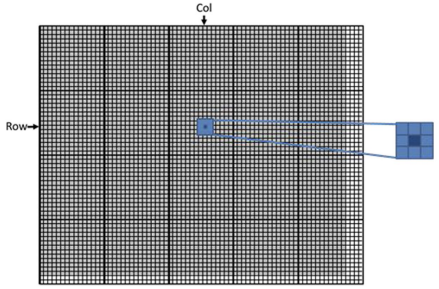
- As before - we compute the thread location
- **If this thread's location is off the edge of the image, we skip computation**
- We get the average over all surrounding locations
  - We calculate the index of a given pixel location
  - If this index is outside the boundaries of the image, we ignore it
  - We keep track of the number of nearby pixels & a running sum of the pixel values
- We compute the average pixel value




```
__global__
void blurKernel(unsigned char *in, unsigned char *out, int w, int h){
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  if(col < w && row < h) {
    int pixVal = 0;
    int pixels = 0;
        // Get average of the surrounding BLUR_SIZE x BLUR_SIZE box
    for(int blurRow=-BLUR_SIZE; blurRow<BLUR_SIZE+1; ++blurRow){
       for(int blurCol=-BLUR_SIZE; blurCol<BLUR_SIZE+1; ++blurCol){
          int curRow = row + blurRow;
          int curCol = col + blurCol;
                // Verify we have a valid image pixel
           if(curRow>=0 && curRow<h && curCol>=0 && curCol<w) {
               pixVal += in[curRow*w + curCol];
               ++pixels; // Keep track of number of pixels in the avg
            }
           }
        }
        // Write our new pixel value out
        out[row*w + col] = (unsigned char)(pixVal/pixels);
    }
}
```

Image Blur Kernel

- As before - we compute the thread location
- If this thread's location is off the edge of the image, we skip computation
- **We get the average over all surrounding locations**
  - We calculate the index of a given pixel location
  - If this index is outside the boundaries of the image, we ignore it
  - We keep track of the number of nearby pixels & a running sum of the pixel values
- We compute the average pixel value




```
__global__
void blurKernel(unsigned char *in, unsigned char *out, int w, int h){
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  if(col < w && row < h) {
    int pixVal = 0;
    int pixels = 0;
        // Get average of the surrounding BLUR_SIZE x BLUR_SIZE box
    for(int blurRow=-BLUR_SIZE; blurRow<BLUR_SIZE+1; ++blurRow){
       for(int blurCol=-BLUR_SIZE; blurCol<BLUR_SIZE+1; ++blurCol){
          int curRow = row + blurRow;
          int curCol = col + blurCol;
                // Verify we have a valid image pixel
           if(curRow>=0 && curRow<h && curCol>=0 && curCol<w) {
               pixVal += in[curRow*w + curCol];
               ++pixels; // Keep track of number of pixels in the avg
            }
          }
        }
        // Write our new pixel value out
        out[row*w + col] = (unsigned char)(pixVal/pixels);
    }
}
```

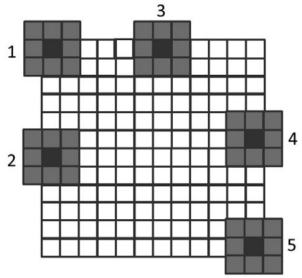
Image Blur Kernel

- As before - we compute the thread location
- If this thread's location is off the edge of the image, we skip computation
- We get the average over all surrounding locations
  - **We calculate the index of a given pixel location**
  - If this index is outside the boundaries of the image, we ignore it
  - We keep track of the number of nearby pixels & a running sum of the pixel values
- We compute the average pixel value

```
__global__
void blurKernel(unsigned char *in, unsigned char *out, int w, int h){
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  if(col < w && row < h) {
    int pixVal = 0;
    int pixels = 0;
        // Get average of the surrounding BLUR_SIZE x BLUR_SIZE box
    for(int blurRow=-BLUR_SIZE; blurRow<BLUR_SIZE+1; ++blurRow){
       for(int blurCol=-BLUR_SIZE; blurCol<BLUR_SIZE+1; ++blurCol){
          int curRow = row + blurRow;
          int curCol = col + blurCol;
                // Verify we have a valid image pixel
          if(curRow>=0 && curRow<h && curCol>=0 && curCol<w) {
              pixVal += in[curRow*w + curCol];
              ++pixels; // Keep track of number of pixels in the avg
            }
          }
        }
        // Write our new pixel value out
        out[row*w + col] = (unsigned char)(pixVal/pixels);
    }
}
```

Image Blur Kernel

- As before - we compute the thread location
- If this thread's location is off the edge of the image, we skip computation
- We get the average over all surrounding locations
  - We calculate the index of a given pixel location
  - **If this index is outside the boundaries of the image, we ignore it**
  - We keep track of the number of nearby pixels & a running sum of the pixel values
- We compute the average pixel value




```
__global__
void blurKernel(unsigned char *in, unsigned char *out, int w, int h){
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  if(col < w && row < h) {
    int pixVal = 0;
    int pixels = 0;
        // Get average of the surrounding BLUR_SIZE x BLUR_SIZE box
    for(int blurRow=-BLUR_SIZE; blurRow<BLUR_SIZE+1; ++blurRow){
       for(int blurCol=-BLUR_SIZE; blurCol<BLUR_SIZE+1; ++blurCol){
          int curRow = row + blurRow;
          int curCol = col + blurCol;
                // Verify we have a valid image pixel
          if(curRow>=0 && curRow<h && curCol>=0 && curCol<w) {
               pixVal += in[curRow*w + curCol];
               ++pixels; // Keep track of number of pixels in the avg
            }
          }
        }
        // Write our new pixel value out
        out[row*w + col] = (unsigned char)(pixVal/pixels);
    }
}
```

Image Blur Kernel

- As before - we compute the thread location
- If this thread's location is off the edge of the image, we skip computation
- We get the average over all surrounding locations
  - We calculate the index of a given pixel location
  - If this index is outside the boundaries of the image, we ignore it
  - **We keep track of the number of nearby pixels & a running sum of the pixel values**
- We compute the average pixel value

```
__global__
void blurKernel(unsigned char *in, unsigned char *out, int w, int h){
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  if(col < w && row < h) {
    int pixVal = 0;
    int pixels = 0;
        // Get average of the surrounding BLUR_SIZE x BLUR_SIZE box
    for(int blurRow=-BLUR_SIZE; blurRow<BLUR_SIZE+1; ++blurRow){
       for(int blurCol=-BLUR_SIZE; blurCol<BLUR_SIZE+1; ++blurCol){
          int curRow = row + blurRow;
          int curCol = col + blurCol;
                // Verify we have a valid image pixel
           if(curRow>=0 && curRow<h && curCol>=0 && curCol<w) {
              pixVal += in[curRow*w + curCol];
              ++pixels; // Keep track of number of pixels in the avg
            }
           }
        }
        // Write our new pixel value out
        out[row*w + col] = (unsigned char)(pixVal/pixels);
    }
}
```

Image Blur Kernel

- As before - we compute the thread location
- If this thread's location is off the edge of the image, we skip computation
- We get the average over all surrounding locations
  - We calculate the index of a given pixel location
  - If this index is outside the boundaries of the image, we ignore it
  - We keep track of the number of nearby pixels & a running sum of the pixel values
- **We compute the average pixel value**

```
__global__
void blurKernel(unsigned char *in, unsigned char *out, int w, int h){
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  if(col < w && row < h) {
    int pixVal = 0;
    int pixels = 0;
        // Get average of the surrounding BLUR_SIZE x BLUR_SIZE box
    for(int blurRow=-BLUR_SIZE; blurRow<BLUR_SIZE+1; ++blurRow){
       for(int blurCol=-BLUR_SIZE; blurCol<BLUR_SIZE+1; ++blurCol){
          int curRow = row + blurRow;
          int curCol = col + blurCol;
                // Verify we have a valid image pixel
           if(curRow>=0 && curRow<h && curCol>=0 && curCol<w) {
               pixVal += in[curRow*w + curCol];
               ++pixels; // Keep track of number of pixels in the avg
             }
           }
        }
       // Write our new pixel value out
       out[row*w + col] = (unsigned char)(pixVal/pixels);
    }
}
```