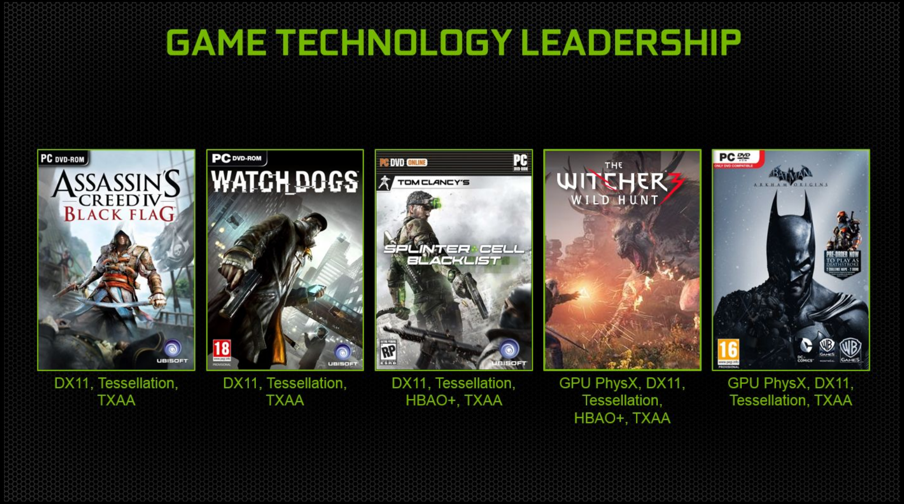
Attribution/License

- Original Materials developed by Mike Shah, Ph.D. (www.mshah.io)
- This work was initially developed for the 2020 PetaScale Blue Waters Institute
- Funding for the development of this work came from http://shodor.org/
-
    -

Introduction to CUDA

Graphics Programming Unit (GPU)

- The Graphics Programming Unit (GPU) is something many of you may be familiar with:
  - GPUs are frequently used to power games and improve their graphics!
  - But modern GPU technology has allowed us to program more than just games
    - GPUs can be used for more general-purpose applications as well!



https://cdn.wccftech.com/wp-content/uploads/2013/08/NVIDIA-Game-Technology-Leadership.png

General Purpose Graphics Programming Unit - GPGPU (1/2)

- General Purpose Graphics Programming Unit
  - This is the term given to a more rich programming interface on GPUs
- NVIDIA's CUDA API is one such Application Programming Interface (API) that allows us to write C programs on our GPU
  - https://en.wikipedia.org/wiki/CUDA#Programming_abilities
  - Before CUDA, we were very limited in how expressive of programs we could write on GPUs--again, primarily applications that handled graphics workloads

https://upload.wikimedia.org/wikipedia/en/b/b9/Nvidia_CUDA_Logo.jpg

General Purpose Graphics Programming Unit - GPGPU (2/2)

- General Purpose Graphics Programming Unit
  - This is the term given to a more rich programming interface on GPUs
- NVIDIA's CUDA API
  Programming Inter
  to write C program
  - https://en.wikipedia.org/w
  - Before CUDA, we were very limited in how expressive of programs we could write on GPUs--again, primarily applications that handled graphics workloads

Let's go ahead and look at how a CUDA program differs from C programs that you have been writing

pload.wikimedia.org/wikipedia/en/b/b9/Nvidia_C
ogo.jpg

A CPU Use Case and a GPU Use Case

CPU Program

- To understand more about how a GPU works, let's first look at solving a problem on the CPU
- We will then solve that exact same problem using the GPU

Problem: Add 1 to every item in an array (1/2)

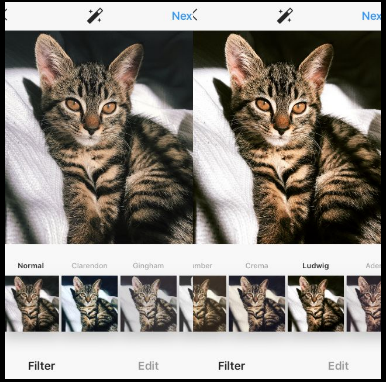
- The program we are going to write is a simple one
- We are going to 'add 1' to every element in an array

Before						
myArray	76	57	42	89	...	7
Index	0	1	2	3	...	n

After						
myArray	77	58	43	90	...	8
Index	0	1	2	3	...	n

Problem: Add 1 to every item in an array (2/2)

- While this is a simple program--it demonstrates a real world use case for something you might do in image processing--such as ‘brighten’ the pixels in an image



https://www.artsycouture.com/blog/wp-content/uploads/2020/03/ludwig.png

Before						
myArray	76	57	42	89	...	7
Index	0	1	2	3	...	n

After						
myArray	77	58	43	90	...	8
Index	0	1	2	3	...	n

cpuAddOne (in file hello_cpu.c) (1/2)

- Provided to the right, is a solution to this problem
  - We can write a function which iterates through every single element in the array, incrementing the value by 1
- Question:
  - What is the complexity of this algorithm?

```c
// Increments all values in array by 1
void cpuAddOne(int* array,int size){
    for(unsigned int i=0; i < size; i++){
        array[i]+=1;
    }
}
```

cpuAddOne (in file hello_cpu.c) (2/2)

- Provided to the right, is a solution to this problem
  - We can write a function which iterates through every single element in the array, incrementing the value by 1
- Question:
  - What is the complexity of this algorithm?
  - O(n) -- because we access every element of the array one time

```c
// Increments all values in array by 1
void cpuAddOne(int* array,int size){
    for(unsigned int i=0; i < size; i++){
        array[i]+=1;
    }
}
```

Complete CPU Program

- The entire CPU program is listed to the right
  - We start from our main() function
  - We allocate memory for our array
  - We initialize values in our array to zero
  - We call our add function
  - Then we print our result
- We can fairly easily trace through the execution of this program.

```c
// Compile with: gcc hello_cpu.cu -o hello_cpu
// Run with    : ./hello_cpu
// Include our C Standard Libraries
#include <stdio.h>
#include <stdlib.h>

// This is our problem size
#define SIZE (640*480)

// Increments all values in array by 1
void cpuAddOne(int* array,int size){
    for(unsigned int i=0; i < size; i++){
        array[i]+=1;
    }
}

int main(){
    // ====================== CPU Code ====================
    // Create an array of numbers
    int* myCPUArray = (int*)malloc(SIZE*sizeof(int));
    // Let's initialize our values to a specific value to start.
    // We can do so using an array.
    for(int i=0; i < SIZE; i++){
        myCPUArray[i] = 0;
    }
    // (For bonus points)
    // Alternatively you can set each byte to a specific value (see 'memset')

    // Now let's actually solve a problem by incrementing our array.
    cpuAddOne(myCPUArray,SIZE);

    for(int i=0; i < SIZE; i++){
        printf("myCPUArray[%i]=%i\n",i,myCPUArray[i]);
    }

    free(myCPUArray);

    return 0;
}
```

Complete CPU Program

- The entire CPU program is listed to the right
  - We start from our r
    function
  - We allocate memor
    array
  - We initialize values
    to zero
  - We call our add function
  - Then we print our result
- We can fairly easily trace through the execution of this program.

Let’s now take a look at CUDA, and run our first General Purpose GPU Program

```
// Compile with: gcc hello_cpu.cu -o hello_cpu
// Run with    : ./hello_cpu
// Include our C Standard Libraries
#include <stdio.h>
#include <stdlib.h>

// This is our problem size
#define SIZE (640*480)

// Increments all values in array by 1
void cpuAddOne(int* array,int size){
    for(unsigned int i=0; i < size; i++){
        array[i]+=1;
    }

U Code ====================
s
oc(SIZE*sizeof(int));
es to a specific value to start.
ay.

    // (For bonus points)
    // Alternatively you can set each byte to a specific value (see 'memset')

    // Now let's actually solve a problem by incrementing our array.
    cpuAddOne(myCPUArray,SIZE);

    for(int i=0; i < SIZE; i++){
        printf("myCPUArray[%i]=%i\n",i,myCPUArray[i]);
    }

    free(myCPUArray);

    return 0;
}
```

GPU Program (in file hello_gpu.cu) (1/2)

- To the right is the GPU code that will do the exact same thing as before!
- Don’t worry, we will walk through the code in more detail.

```
__global__ void gpuAddOne(int* array, int size){
    // We need some  unique variable identifying which thread accesses
    // a piece of data. So below we are going to figure out which
    // thread (i.e. index) is going to operate on which piece of data.
    //
    // This gives us full coverage of every id that we want to access.
    // Figure out which block we are, what dimension, and which thread.
    // You can think of these values like a phone number if you like.
    // The block is the area code for example.
    int currentThread = (blockIdx.x * blockDim.x) + threadIdx.x;
    // Do a bounds check to make sure we are not accessing
    // out of bounds memory.
    // (This can occur if we don't have a perfect number within 32.
    if(currentThread < size){
        array[currentThread] += 1;
    }
}
```

GPU Program (in file hello_

- Here is a full view of the code, so we can see a full sample

```
// Compile with: nvcc hello_gpu.cu -o hello_gpu
// Run with    : ./hello_gpu
#include <stdio.h>
#include <stdlib.h>

// This is our problem size
#define SIZE (640*480)

__global__ void gpuAddOne(int* array, int size){
    int currentThread = (blockIdx.x * blockDim.x) + threadIdx.x;
    if(currentThread < size){
        array[currentThread] += 1;
    }
}
int main(){
    // ====================== CPU Code ====================
    // Create an array of numbers
    int* myCPUArray = (int*)malloc(SIZE*sizeof(int));
    // Let's initialize our values to a specific value to start.
    // We can do so using an array.
    for(int i=0; i < SIZE; i++){
        myCPUArray[i] = 0;
    }
    // ====================== CPU Code ====================
    // ====================== GPU Code ====================
    int* myGPUArray = (int*)malloc(SIZE*sizeof(int));
    cudaMalloc(&myGPUArray, SIZE*sizeof(int));
    cudaMemcpy(myGPUArray, myCPUArray, SIZE*sizeof(int), cudaMemcpyHostToDevice)
    int NUM_THREADS = 256;
    int NUM_BLOCKS = (int)ceil(SIZE/ (float)NUM_THREADS);
    gpuAddOne<<<NUM_BLOCKS,NUM_THREADS>>>(myGPUArray,SIZE);
    cudaMemcpy(myCPUArray, myGPUArray, SIZE*sizeof(int), cudaMemcpyDeviceToHost)

    for(int i=0; i < SIZE; i++){
        printf("myCPUArray [%i]=%i\n",i,myCPUArray[i]);
    }

    cudaFree(myGPUArray);
    // ====================== GPU Code ====================
    // Free our CPU Memory
    free(myCPUArray);
    return 0;
}
```

GPU Program (in file hello_gpu.cu) (2/2)

- Question to audience:
  - What is the O(n) of this code?
    - ?!

```
__global__ void gpuAddOne(int* array, int size){
    // We need some  unique variable identifying which thread accesses
    // a piece of data. So below we are going to figure out which
    // thread (i.e. index) is going to operate on which piece of data.
    //
    // This gives us full coverage of every id that we want to access.
    // Figure out which block we are, what dimension, and which thread.
    // You can think of these values like a phone number if you like.
    // The block is the area code for example.
    int currentThread = (blockIdx.x * blockDim.x) + threadIdx.x;
    // Do a bounds check to make sure we are not accessing
    // out of bounds memory.
    // (This can occur if we don't have a perfect number within 32.
    if(currentThread < size){
        array[currentThread] += 1;
    }
}
```

GPU Program (in file hello_gpu.cu) (2/2)

- Question to audience:
    - What is the O(n) of this code?
        - Answer is closer to O(1) -- and we'll need to look at our GPU to understand why!

```
__global__ void gpuAddOne(int* array, int size){
    // We need some  unique variable identifying which thread accesses
    // a piece of data. So below we are going to figure out which
    // thread (i.e. index) is going to operate on which piece of data.
    //
    // This gives us full coverage of every id that we want to access.
    // Figure out which block we are, what dimension, and which thread.
    // You can think of these values like a phone number if you like.
    // The block is the area code for example.
    int currentThread = (blockIdx.x * blockDim.x) + threadIdx.x;
    // Do a bounds check to make sure we are not accessing
    // out of bounds memory.
    // (This can occur if we don't have a perfect number within 32.
    if(currentThread < size){
        array[currentThread] += 1;
    }
}
```

GPU Program (in file hello_gpu.cu) (2/2)

- Question to audience:
  - What is the O(n) of this code?
    - Answer is clos
      and we'll nee
      our GPU to ur
      why!

Let's understand a bit more about 'why'

```
__global__ void gpuAddOne(int* array, int size){
    // We need some  unique variable identifying which thread accesses
    // a piece of data. So below we are going to figure out which
    // thread (i.e. index) is going to operate on which piece of data.
    //
                      rage of every id that we want to access.
                      we are, what dimension, and which thread.
                      values like a phone number if you like.
                      ode for example.
                      Idx.x * blockDim.x) + threadIdx.x;
                      ke sure we are not accessing

                      on't have a perfect number within 32.

                      = 1;

}
```

GPU Architecture

GPU Architecture (1/2)

- The first thing we need to understand in our use case, is that one code is written on the CPU, and one for the GPU
- GPU's (right) are structured differently than our CPUs (left)
- Notice the little 'green' boxes in the GPU--there are many more of these compute units--and those units are meant to execute code in parallel in individual **threads.**

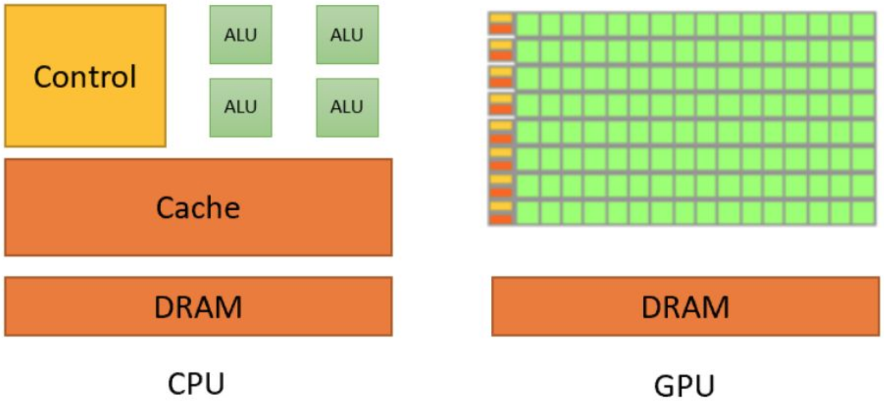


Figure 2: GPUs devote more transistors to compute data processing.

https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/

GPU Architecture (2/2)

- The first thing we need to understand in our use case, is that one code is written ... and one for the GP...
- GPU's (right) are st... differently than ou...
- Notice the little 'gr... the GPU--there are many more of these compute units--and those units are meant to execute code in parallel in individual **threads.**

So what is a thread, and how do we execute code in these 'boxes'

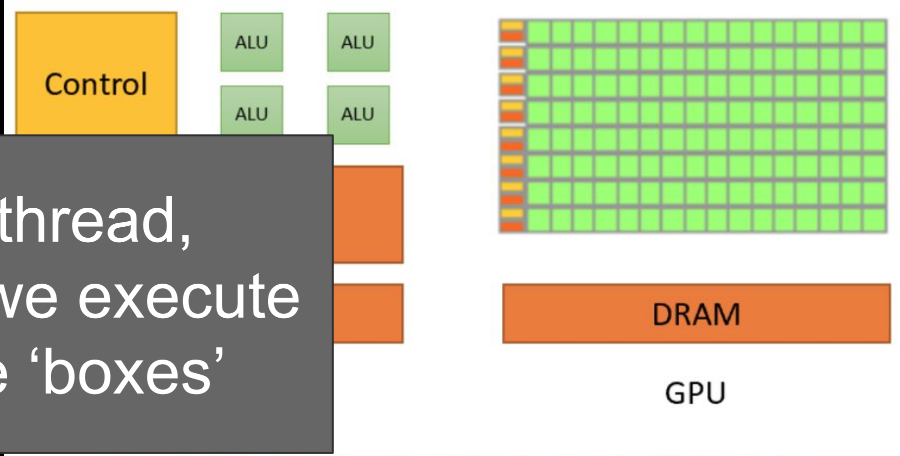


Figure 2: GPUs devote more transistors to compute data processing.

https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/

Threads

- Threads are traditionally referred to as ‘lightweight’ processes
- They are used to execute a part of your program concurrently while your program runs.
- In CUDA, we execute many threads (100s or 1000s) in parallel in order to get a speedup in execution.
  - So if we look at our example to the right
  - We can see we a ‘single thread’ that is going to execute this tiny gpuAddOne program on our GPU

```
__global__ void gpuAddOne(int* array, int size){
    // We need some  unique variable identifying which thread accesses
    // a piece of data. So below we are going to figure out which
    // thread (i.e. index) is going to operate on which piece of data.
    //
    // This gives us full coverage of every id that we want to access.
    // Figure out which block we are, what dimension, and which thread.
    // You can think of these values like a phone number if you like.
    // The block is the area code for example.
    int currentThread = (blockIdx.x * blockDim.x) + threadIdx.x;
    // Do a bounds check to make sure we are not accessing
    // out of bounds memory.
    // (This can occur if we don't have a perfect number within 32.
    if(currentThread < size){
        array[currentThread] += 1;
    }
}
```

Threads Continued - Single Instruction Multiple Thread (SIMT)

- In CUDA we call this model SIMT, for single instruction multiple thread
- We are going to have many (100s or 1000s) of threads executing on each of our GPUs execution units.

```
__global__ void gpuAddOne(int* array, int size){
    // We need some  unique variable identifying which thread accesses
    // a piece of data. So below we are going to figure out which
    // thread (i.e. index) is going to operate on which piece of data.
    //
    // This gives us full coverage of every id that we want to access.
    // Figure out which block we are, what dimension, and which thread.
    // You can think of these values like a phone number if you like.
    // The block is the area code for example.
    int currentThread = (blockIdx.x * blockDim.x) + threadIdx.x;
    // Do a bounds check to make sure we are not accessing
    // out of bounds memory.
    // (This can occur if we don't have a perfect number within 32.
    if(currentThread < size){
        array[currentThread] += 1;
    }
}
```

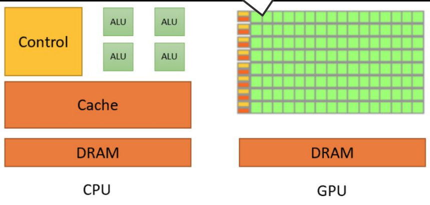


Figure 2: GPUs devote more transistors to compute data processing.

Threads Continued - Single Instruction Multiple Thread (SIMT)

- So in CUDA, the lowest granularity of execution is a 'thread'.
  - A thread executes a series of instructions (i.e. a small program).
- Groups of threads (with consecutive thread indexes) are divided into warps which execute on a single CUDA core.
  - The lowest schedulable entity is known as a 'warp'

```
__global__ void gpuAddOne(int* array, int size){
    // We need some  unique variable identifying which thread accesses
    // a piece of data. So below we are going to figure out which
    // thread (i.e. index) is going to operate on which piece of data.
    //
    // This gives us full coverage of every id that we want to access.
    // Figure out which block we are, what dimension, and which thread.
    // You can think of these values like a phone number if you like.
    // The block is the area code for example.
    int currentThread = (blockIdx.x * blockDim.x) + threadIdx.x;
    // Do a bounds check to make sure we are not accessing
    // out of bounds memory.
    // (This can occur if we don't have a perfect number within 32.
    if(currentThread < size){
        array[currentThread] += 1;
    }
}
```

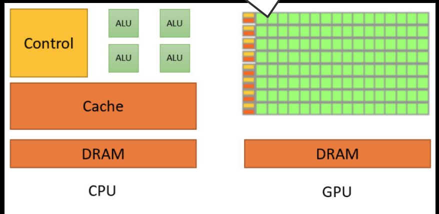


Figure 2: GPUs devote more transistors to compute data processing.

Threads Continued - Single Instruction Multiple Thread (SIMT)

- Thread blocks are the lowest programmable entity.
  - A single shader core is assigned to each thread.
  - They are organized in a 3-D block.
- Thread blocks are then further assigned to a GPU.

```
__global__ void gpuAddOne(int* array, int size){
    // We need some  unique variable identifying which thread accesses
    // a piece of data. So below we are going to figure out which
    // thread (i.e. index) is going to operate on which piece of data.
    //
    // This gives us full coverage of every id that we want to access.
    // Figure out which block we are, what dimension, and which thread.
    // You can think of these values like a phone number if you like.
    // The block is the area code for example.
    int currentThread = (blockIdx.x * blockDim.x) + threadIdx.x;
    // Do a bounds check to make sure we are not accessing
    // out of bounds memory.
    // (This can occur if we don't have a perfect number within 32.
    if(currentThread < size){
        array[currentThread] += 1;
    }
}
```

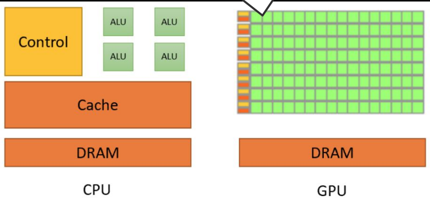


Figure 2: GPUs devote more transistors to compute data processing.

Further Resources

- CUDA Programming Model
    - https://cvw.cac.cornell.edu/GPU/simt_warp