CS5375 Computer Systems Organization and Architecture

Lecture 17

Guest Instructors:

Ghazanfar Ali, Ghazanfar.Ali@ttu.edu

Mert Side, Mert.Side@ttu.edu

Department of Computer Science

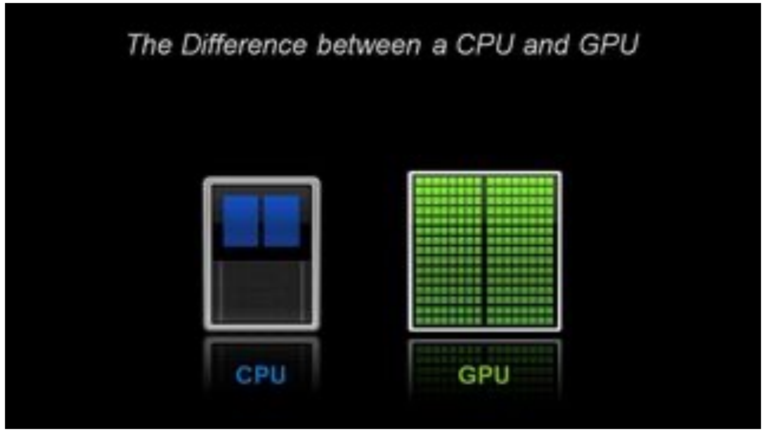
Texas Tech University

Outline

- Recap
- What Is CUDA?
- How Do You Use CUDA?
- Example: Vector Addition
  - Running on the CPU
  - Running on the GPU
  - Utilizing GPU Threads/Blocks
  - Optimizations

RECAP: What's the Difference Between a CPU and a GPU?

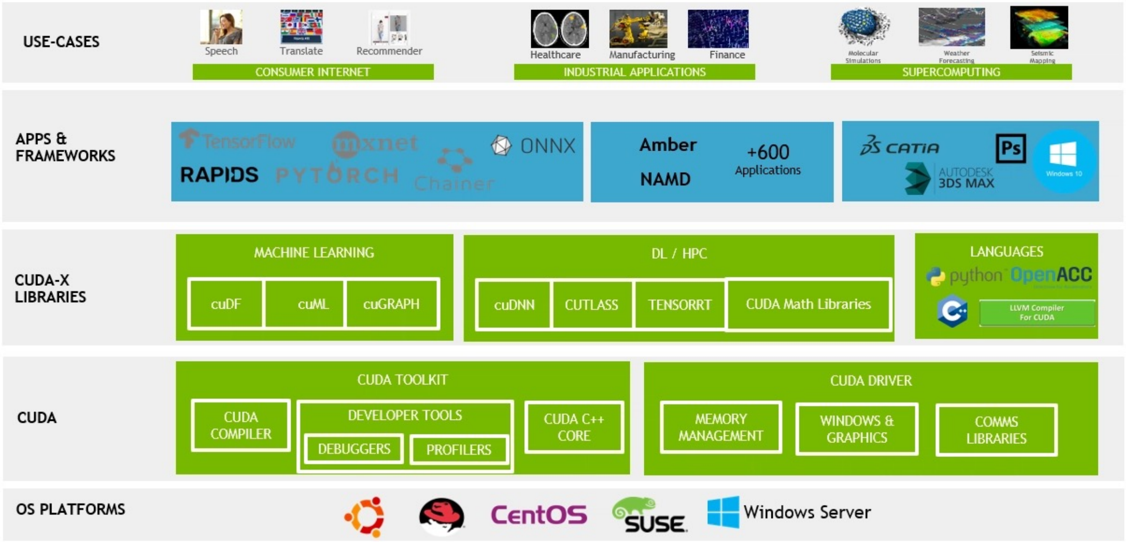
CPU	GPU
Central Processing Unit	Graphics Processing Unit
Several cores	Many cores
Low latency	High throughput
Good for serial processing	Good for parallel processing
Can do a handful of operations at once	Can do thousands of operations at once



What's the Difference Between a CPU and a GPU? – https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

What Is CUDA?

- CUDA (or Compute Unified Device Architecture) is a parallel computing platform and API.
- It allows software to use certain types of general-purpose computing on GPUs (GPGPU).
- CUDA is a software layer that gives direct access to the GPU for the execution of compute kernels.
- CUDA provides a set of programming extensions based on the C/C++ family of languages.
- If you have a basic understanding of C and understand the concept of threads and SIMD execution, then CUDA is easy to pick up.



How Do You Use CUDA?

- With CUDA, you can write programs using supported languages that includes C, C++, Python and MATLAB, by incorporating a few keywords.
- These keywords let the developer express parallelism and direct the compiler to GPU accelerators.
- The simple example shows how a standard C program can be accelerated using CUDA.

Standard C Code

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

C with CUDA extensions

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

Today's Objectives

- Today, we will write a program to add the elements of two arrays.
- We will begin by examining the code in C++ running on the CPU.
- Then, we will write the CUDA version of that code to run on the GPU.
- We will see that taking full advantage of the GPU requires some fine-tuning.
- To achieve this, we will profile different versions of the code to make it run faster!

```cpp
// Running on CPU

#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++) {
    y[i] = x[i] + y[i];
  }
}

int main(void)
{
  int N = 1<<25; // 33M elements
  //int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  /*
  for (int i = 0; i < N; i++) {
    std::cout << y[i];
  }
  */

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
~
~
~
~
~
```

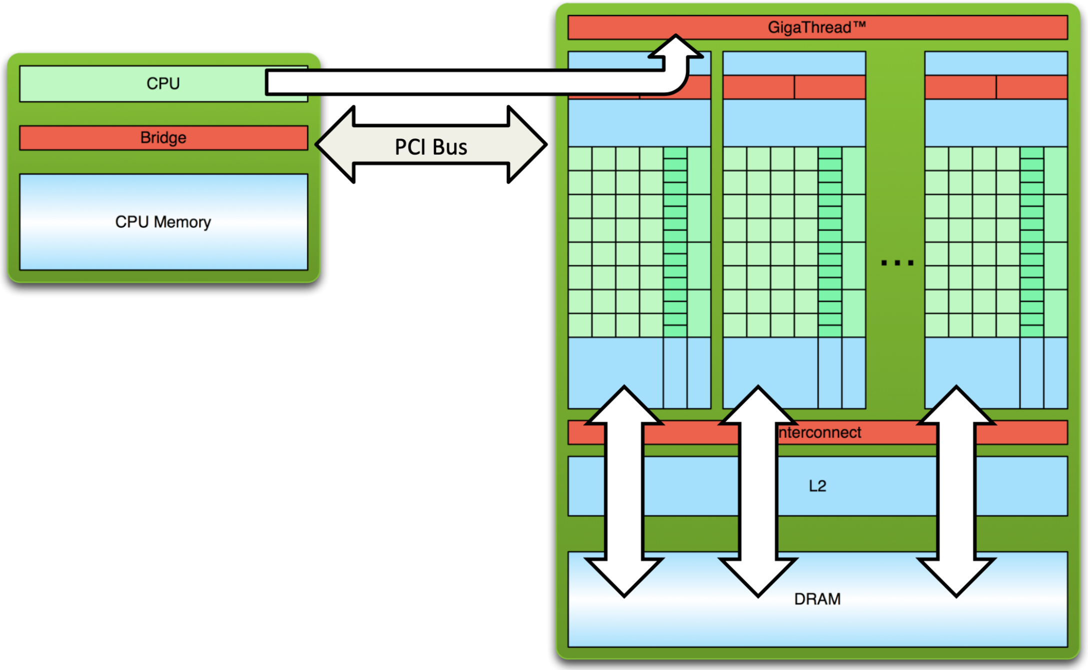
Example: Add the elements of two arrays in C++

- A simple C++ program that adds the elements of two arrays with 33 million elements each.
- Compile and run this C++ program:
  - g++ add_v0.cpp -o add_v0.exe
- As expected, it prints that there was no error in the summation and then exits.
- Takes 397ms on the CPU.
- Now, I want to get this computation running in parallel on a GPU.

The Software View

- In a bit more detail, at the top level, we have a master process which runs on the CPU and performs the following steps:

1. Initializes the card.
2. Allocates memory on the host and on the device.
3. Copies data from the host memory to device memory.
4. Launches multiple instances of the execution "kernel" on the device.
5. Copies data from the device memory to the host.
6. Repeat 3-5 as needed.
7. De-allocates all memory and terminates.



Memory Allocation in CUDA

- To compute on the GPU,
  - We need to allocate memory accessible by the GPU.
  - Unified Memory in CUDA makes this easy by providing a single memory space accessible by all GPUs and CPUs in your system.
  - To allocate data in unified memory, call `cudaMallocManaged()`, which returns a pointer that you can access from host (CPU) code or device (GPU) code.
  - To free the data, just pass the pointer to `cudaFree()`.
- Replace the calls to `new` with calls to `cudaMallocManaged()`, and replace calls to `delete []` with calls to `cudaFree`.

```
// Allocate Unified Memory

float *x, *y;

cudaMallocManaged(&x, N*sizeof(float));

cudaMallocManaged(&y, N*sizeof(float));



...



// Free memory

cudaFree(x);

cudaFree(y);
```

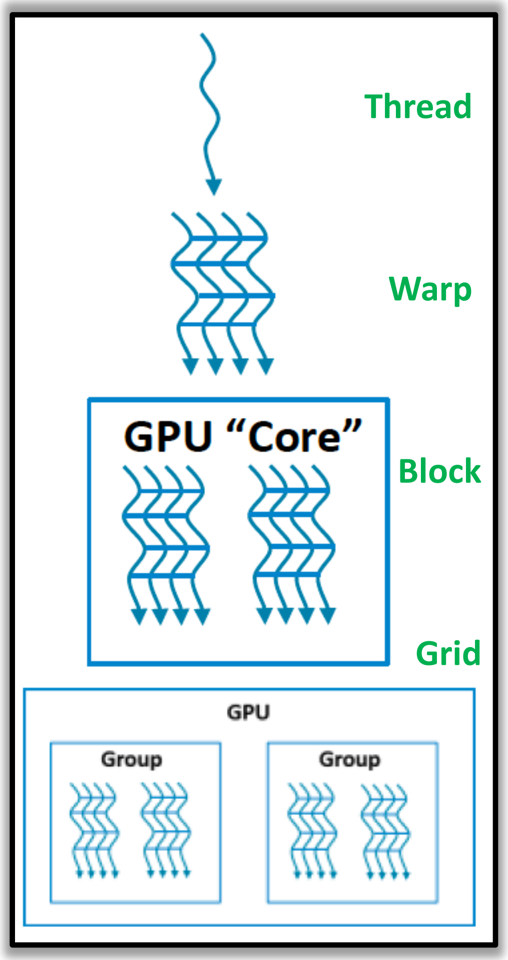
```
// 1 CUDA Thread

#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<25; // 33M elements
  //int N = 1<<20; // 1M elements
  float *x, *y;

  // Allocate Unified Memory – accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 33M elements on the GPU
  add<<<1, 1>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
~
~
~
~
~
~
```

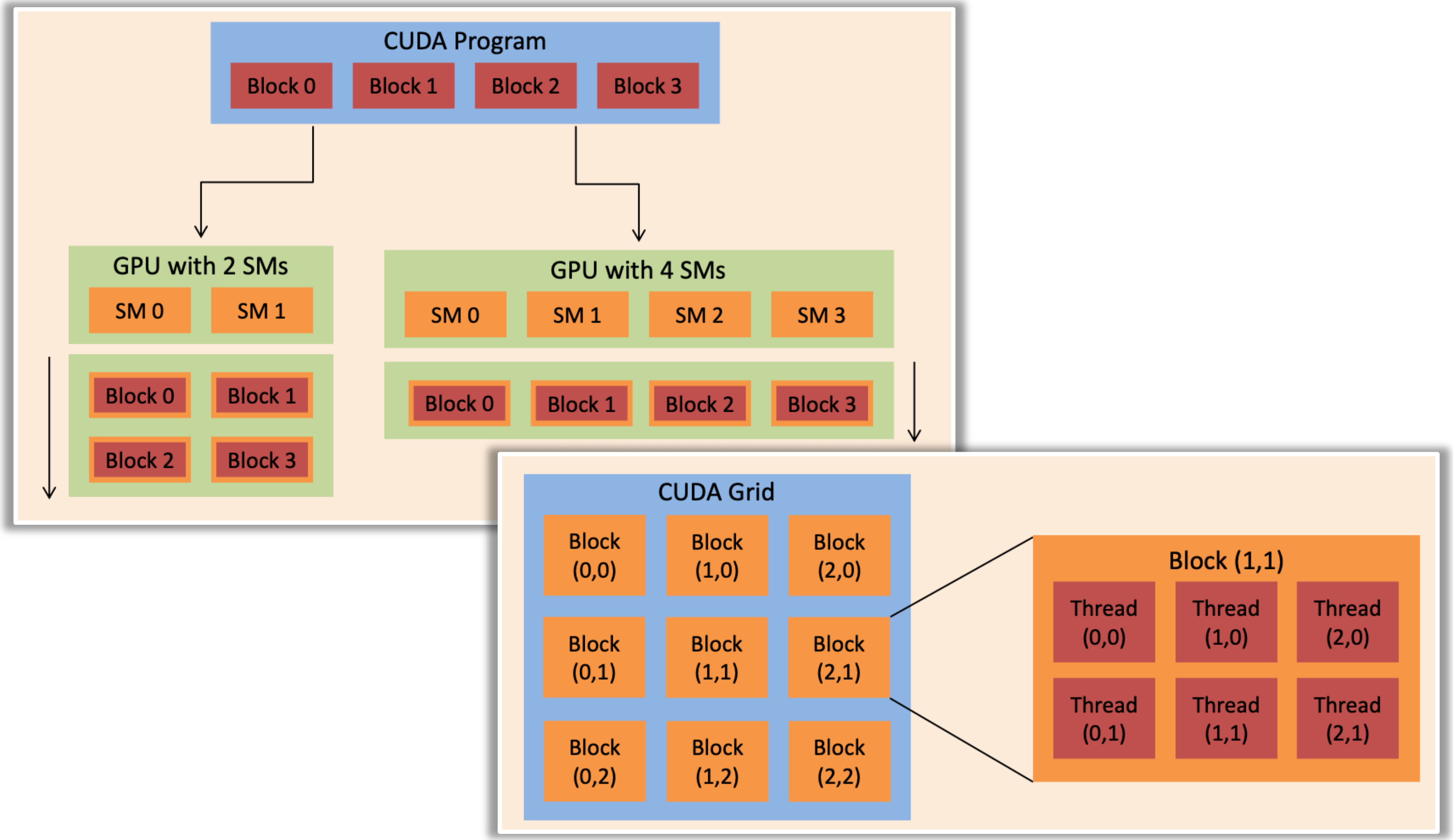
if you don't specify the global key word it will asume it as a device function not GPU

E.g.: Add the elements of two arrays in CUDA

- Easy! This launches program one GPU thread to run `add()`.
  - ➢ `nvcc add_v1.cu -o add_v1.exe`
- CPU should wait until the kernel is done!
- Let's profile it:
  - ➢ `nvprof ./add_v1.exe`
- Takes 3.32s on the GPU; that's long!?
- Now, how can we utilize more threads?

RECAP: Programming Model





Multiple Threads in CUDA

- We ran a kernel with one thread that does some computation, how do you make it parallel?
- The key is in CUDA's `<<<1, 1>>>` syntax.
- This is called the execution configuration.
- There are two parameters here, let's start by changing the second one:
  - the number of threads in a thread block.
  - CUDA GPUs run kernels using blocks of threads that are a multiple of 32 in size, so 256 threads is a reasonable size to choose.

```
add<<<1, 256>>>(N, x, y);
```

- If I run the code with only this change, it will do the computation once per thread, rather than spreading the computation across the parallel threads.

```
// 1 CUDA Block

#include <iostream>
#include <math.h>

// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<25; // 33M elements
  //int N = 1<<20; // 1M elements
  float *x, *y;

  // Allocate Unified Memory – accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 33M elements on the GPU
  add<<<1, 256>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
~
~
~
~
```

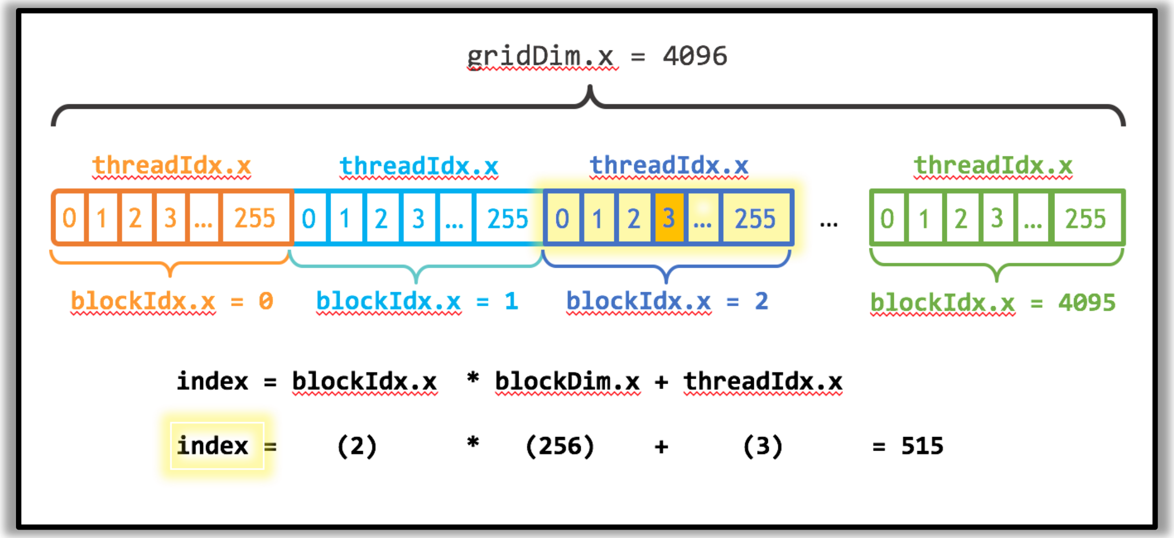
E.g.: Threads

- Modify the loop to stride through the array with parallel threads.
- Specifically, **`threadIdx.x`** contains the index of the current thread within its block, and **blockDim.x** contains the number of threads in the block.
- Takes 71.55ms on the GPU; much better!
- But what about that other parameter in the execution configuration?

Blocks in CUDA

- CUDA GPUs have many parallel processors grouped into Streaming Multiprocessors, or SMs. Each SM can run multiple concurrent thread blocks
- Together, the blocks of parallel threads make up what is known as the ***grid***.
- Since we have N elements to process, and 256 threads per block, we just need to calculate the number of blocks to get at least N threads.
- Simply divide N by the block size (being careful to round up in case N is not a multiple of `blockSize`).
- Figure illustrates the the approach to indexing into an array (one-dimensional) in CUDA using `blockDim.x`, `gridDim.x`, and `threadIdx.x`.

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```



```
// Many CUDA Blocks

#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<25; // 33M elements
  //int N = 1<<20; // 1M elements
  float *x, *y;

  // Allocate Unified Memory – accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 33M elements on the GPU
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
~
~
~
```

E.g.: Blocks

- Update the kernel code to consider the entire grid of thread blocks.
- CUDA provides `gridDim.x`, which contains the number of blocks in the grid, and `blockIdx.x`, which contains the index of the current thread block in the grid.
- The idea is that each thread gets its index by computing the offset to the beginning of its block (the block index times the block size: `blockIdx.x * blockDim.x`) and adding the thread's index within the block (`threadIdx.x`).
- Takes 62.22ms but 1152 CPU page faults.
- There are many host-to-device page faults, reducing the throughput achieved by the CUDA kernel.

Taking Advantage of the Unified Memory in CUDA

- In a real application, the GPU is likely to perform a lot more computation on data (perhaps many times) without the CPU touching it.
- The migration overhead in this simple code is caused by the fact that the CPU initializes the data, and the GPU only uses it once.
- There are a few different ways that I can eliminate or change the migration overhead to get a more accurate measurement of the vector add kernel performance.

❖ Move the data initialization to the GPU in another CUDA kernel.

❖ Prefetch the data to GPU memory before running the kernel.

- Let’s look at each of these approaches.

```
// With unified memory

#include <iostream>
#include <math.h>

// CUDA kernel to initialize elements of two arrays
__global__ void init(int n, float *x, float *y) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
}
// CUDA kernel to add elements of two arrays
__global__
void add(int n, float *x, float *y) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

int main(void) {
  int N = 1<<25; // 33M elements
  //int N = 1<<20; // 1M elements
  float *x, *y;

  // Allocate Unified Memory -- accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  // initialize x and y arrays on the host
  init<<<numBlocks, blockSize>>>(N, x, y);
  // Launch kernel on 33M elements on the GPU
  add<<<numBlocks, blockSize>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x); cudaFree(y);

  return 0;
}
```

E.g.: Unified Memory

- If we move initialization from the CPU to the GPU, the add kernel won't page fault.
- Here's a simple CUDA C++ kernel to initialize the data.
- Total CPU Page faults decreased to 384.
  - 17.73ms for `init(int, float*, float*)`
  - 498.88us for `add(int, float*, float*)`

```
// With prefetching
#include <iostream>
#include <math.h>
// CUDA kernel to initialize elements of two arrays
__global__ void init(int n, float *x, float *y) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride) {
    x[i] = 1.0f; y[i] = 2.0f;
  }
}
// CUDA kernel to add elements of two arrays
__global__
void add(int n, float *x, float *y) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

int main(void) {
  int N = 1<<25; // 33M elements
  //int N = 1<<20; // 1M elements
  float *x, *y;

  // Allocate Unified Memory -- accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));
  // Prefetch the data to the GPU
  int device = -1;
  cudaGetDevice(&device);
  cudaMemPrefetchAsync(x, N*sizeof(float), device, NULL);
  cudaMemPrefetchAsync(y, N*sizeof(float), device, NULL);

  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  // initialize x and y arrays on the host
  init<<<numBlocks, blockSize>>>(N, x, y);
  // Launch kernel on 33M elements on the GPU
  add<<<numBlocks, blockSize>>>(N, x, y);
  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x); cudaFree(y);
  return 0;
}
```

E.g.: Prefetching

- The other approach is to use Unified Memory prefetching to move the data to the GPU after initializing it.
- `cudaMemPrefetchAsync()` is for this purpose.
- Total CPU Page faults remains 384.
  - 501.47us for init(int, float*, float*)
  - 303.52us for add(int, float*, float*)

Source Code

- Code from today's lecture:

https://github.com/mertside/CS5375_GPU_Lecture

Summary

- Today, we learned how to write a program to add the elements of two arrays.
- We ran this code both on the CPU and the GPU.
- We discussed the GPU programming model and had our first steps in CUDA programming.
- We saw taking full advantage of the GPU requires some fine-tuning.
- We introduced the basics of memory allocation on the GPU and the unified memory in CUDA, along with prefetching.

Readings

- How to CUDA? GPU Accelerated Computing with C and C++:
  - https://developer.nvidia.com/how-to-cuda-c-cpp
- Introduction to CUDA:
  - https://developer.nvidia.com/blog/even-easier-introduction-cuda/
- Unified Memory with CUDA:
  - https://developer.nvidia.com/blog/unified-memory-cuda-beginners/
- How to Optimize Data Transfers in CUDA C/C++:
  - https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/
- An Efficient Matrix Transpose in CUDA using Shared Memory:
  - https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/