CS 179: GPU Computing

LECTURE 2: MORE BASICS

Recap

Can use GPU to solve highly parallelizable problems

Straightforward extension to C++

- Separate CUDA code into .cu and .cuh files and compile with nvcc to create object files (.o files)

Looked at the a[] + b[] -> c[] example

Recap

If you forgot everything, just make sure you understand that CUDA is simply an extension of other bits of code you write!!!!

- Evident in .cu/.cuh vs .cpp/.hpp distinction
- .cu/.cuh is compiled by nvcc to produce a .o file
- .cpp/.hpp is compiled by g++ and the .o file from the CUDA code is simply linked in using a "#include xxx.cuh" call
  - No different from how you link in .o files from normal C++ code

.cu/.cuh vs .cpp/.hpp

~/Documents/test/cuda_test.cuh - Sublime Text (UNREGISTERED)

test.cpp | cuda_test.cu | test.hpp | cuda_test.cuh

```
#ifndef CUDA_TEST_HPP
#define CUDA_TEST_HPP

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#ifdef __CUDACC__
#define CUDA_CALLABLE __host__ __device__
#else
#define CUDA_CALLABLE
#endif

void cudaCallAddVectorKernel(
    const uint block_count,
    const uint per_block_thread_count,
    const float *a,
    const float *b,
    float *c,
    const uint size);

#endif
```

Line 19, Column 14 | Spaces: 4 | C++

.cu/.cuh vs .cpp/.hpp

~/Documents/test/cuda_test.cu - Sublime Text (UNREGISTERED)

test.cpp | cuda_test.cu | test.hpp | cuda_test.cuh

```
#include "cuda_test.cuh"

__global__
void cudaAddVectorKernel(
    const float *a,
    const float *b,
    float *c,
    const uint size)
{
    /* get current thread's id */
    uint thread_index = blockIdx.x * blockDim.x + threadIdx.x;

    /* while this thread is dealing with a valid index */
    while (thread_index < size) {
        /* add a and b into c */
        c[thread_index] = a[thread_index] + b[thread_index];

        /* advance thread id */
        thread_index += blockDim.x * gridDim.x;
    }
}

void cudaCallAddVectorKernel(
    const uint block_count,
    const uint per_block_thread_count,
    const float *a,
    const float *b,
    float *c,
    const uint size)
{
    cudaAddVectorKernel<<<block_count, per_block_thread_count>>>(a, b, c, size);
}
```

Line 11, Column 37 | Spaces: 4 | C++

Menu | Messenger - Googl... | Terminal | ~/Documents/test... | Pictures | Wednesday March 30, 12:51:16 AM

.cu/.cuh vs .cpp/.hpp

~/Documents/test/test.cpp - Sublime Text (UNREGISTERED)

test.cpp | cuda_test.cu | test.hpp | cuda_test.cuh

```cpp
#include "test.hpp"

#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <assert.h>
#include <cmath>
#include <cuda_runtime.h>
#include "cuda_test.cuh"

using namespace std;

int main(int argc, char **argv) {
    /* setup block size and max block count */
    const uint per_block_thread_count = 1024;
    const uint max_block_count = 65535;

    /* setup host memory */
    const uint array_size = 10000000;
    float *a = new float[array_size];
    float *b = new float[array_size];
    float *c = new float[array_size];
    // fill a and b
    for (uint i = 0; i < array_size; i++) {
        a[i] = i;
        b[i] = array_size - i;
    }

    /* setup device memory */
    float *dev_a;
    float *dev_b;
    float *dev_c;
```

Line 27, Column 6 | Spaces: 4 | C++

.cu/.cuh vs .cpp/.hpp

~/Documents/test/test.cpp - Sublime Text (UNREGISTERED)

test.cpp | cuda_test.cu | test.hpp | cuda_test.cuh

```cpp

    /* setup device memory */
    float *dev_a;
    float *dev_b;
    float *dev_c;
    cudaMalloc((void**) &dev_a, array_size * sizeof(float));
    cudaMalloc((void**) &dev_b, array_size * sizeof(float));
    cudaMalloc((void**) &dev_c, array_size * sizeof(float));

    /* copy a and b into dev_a and dev_b */
    cudaMemcpy(dev_a, a, array_size * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, array_size * sizeof(float), cudaMemcpyHostToDevice);

    /* call kernel to add the two arrays into dev_c */
    uint block_count = min(max_block_count,
        (uint) ceil(array_size / (float) per_block_thread_count));
    cudaCallAddVectorKernel(
        block_count,
        per_block_thread_count,
        dev_a,
        dev_b,
        dev_c,
        array_size);

    /* copy dev_c into c */
    cudaMemcpy(c, dev_c, array_size * sizeof(float), cudaMemcpyDeviceToHost);

    /* check the output */
    for (uint i = 0; i < array_size; i++) {
        assert(c[i] == array_size);
    }

    /* free device memory */
    delete[] a;
    delete[] b;
    delete[] c;
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

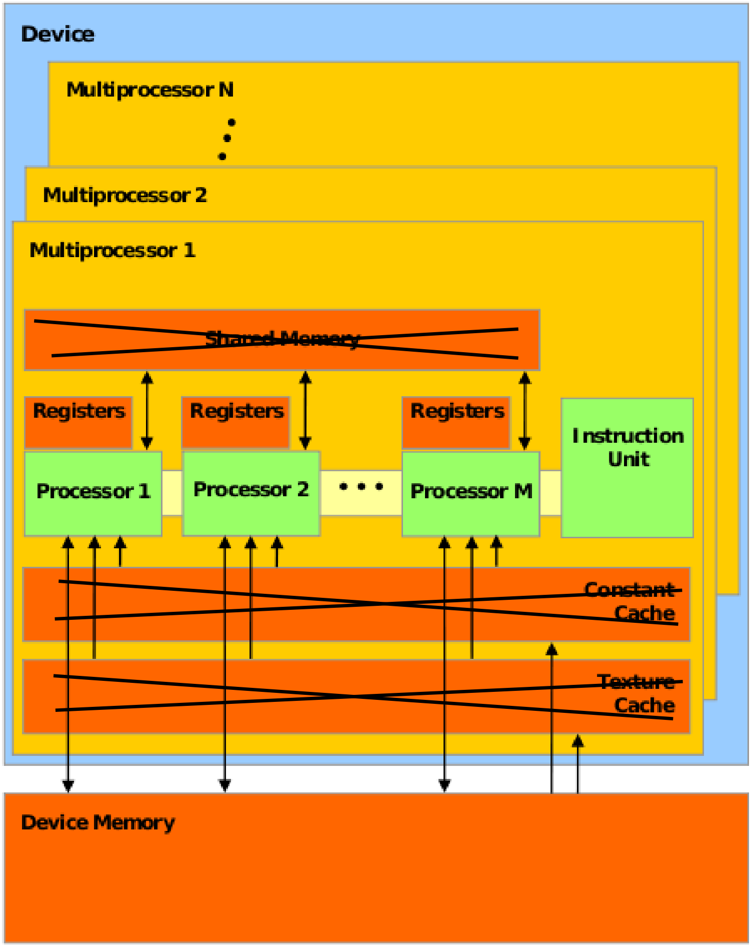
Line 27, Column 6 | Spaces: 4 | C++

Thread Organization

We will now look at how threads are organized and used in GPUs

- Keywords you MUST know to code in CUDA:
  - Thread
  - Block
  - Grid
- Keywords you MUST know to code WELL in CUDA:
  - (Streaming) Multiprocessor
  - Warp
  - Warp Divergence

Inside a GPU



The black Xs are just crossing out things you don't have to think about just yet. You'll learn about them later

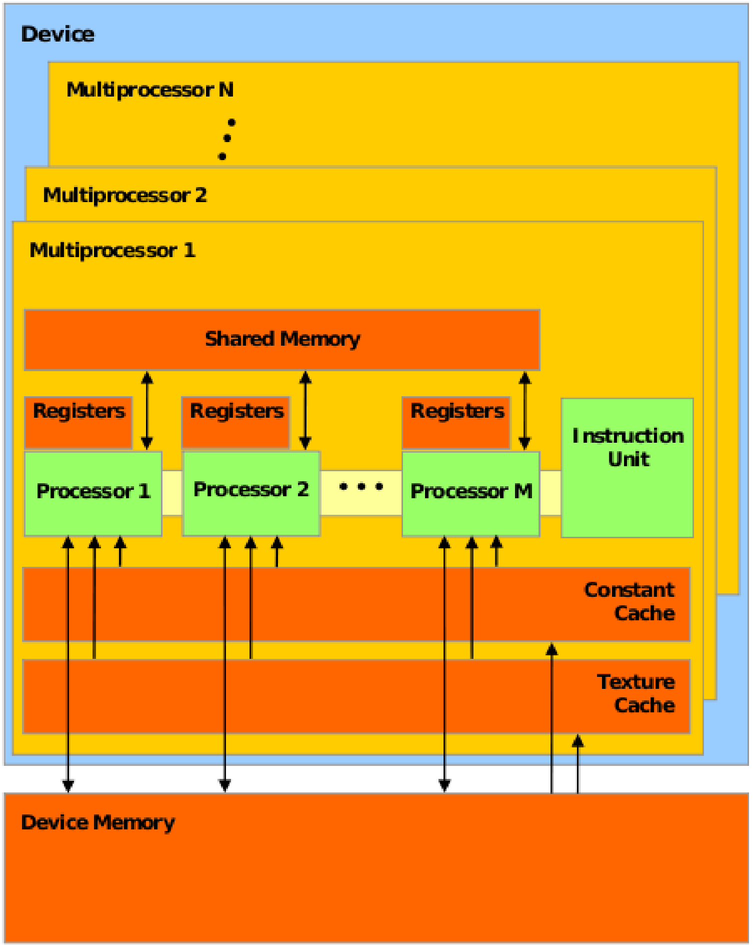
Inside a GPU

Think of **Device Memory** (we will also refer to it as **Global Memory**) as a RAM for your GPU

- Faster than getting memory from the actual RAM but still can be faster
- Will come back to this in future lectures

GPUs have many **Streaming Multiprocessors (SMs)**

- Each SM has multiple processors but only one instruction unit
- Groups of processors must run the exact same set of instructions at any given time with in a single SM



Inside a GPU

When a kernel (the thing you define in .cu files) is called, the task is divided up into threads

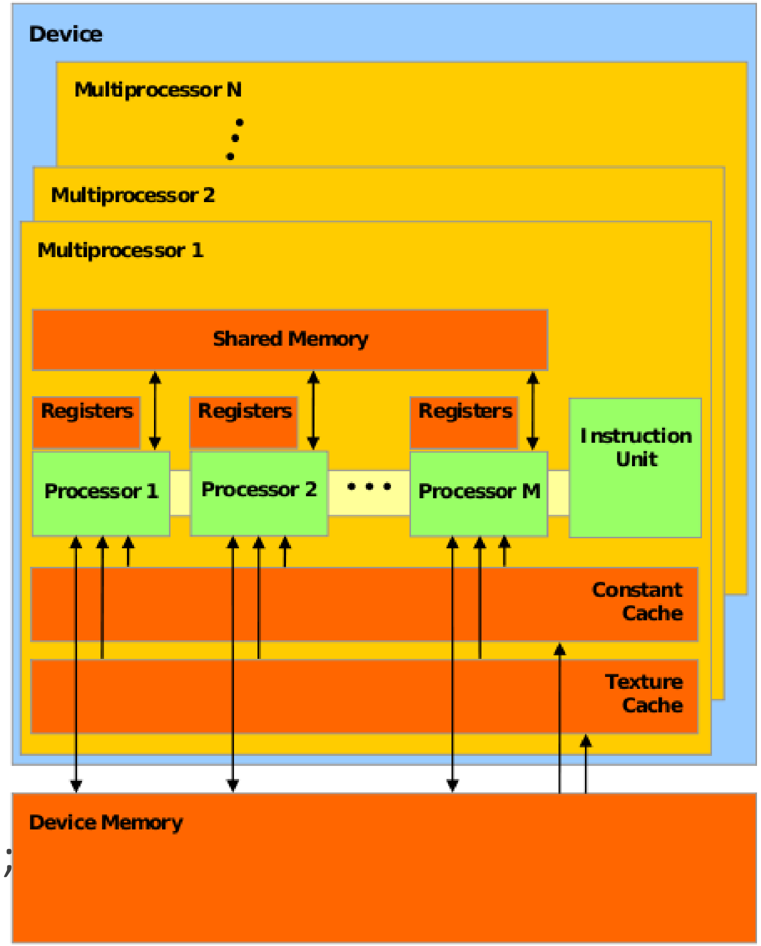
- Each thread handles a small portion of the given task

The threads are divided into a **Grid** of **Blocks**

- Both Grids and Blocks are 3 dimensional
- e.g.

```
dim3 dimBlock(8, 8, 8);
dim3 dimGrid(100, 100, 1);
Kernel<<<dimGrid, dimBlock>>>(…);
```

- However, we'll often only work with 1 dimensional grids and blocks
- e.g. Kernel<<<block_count, block_size>>>(…);

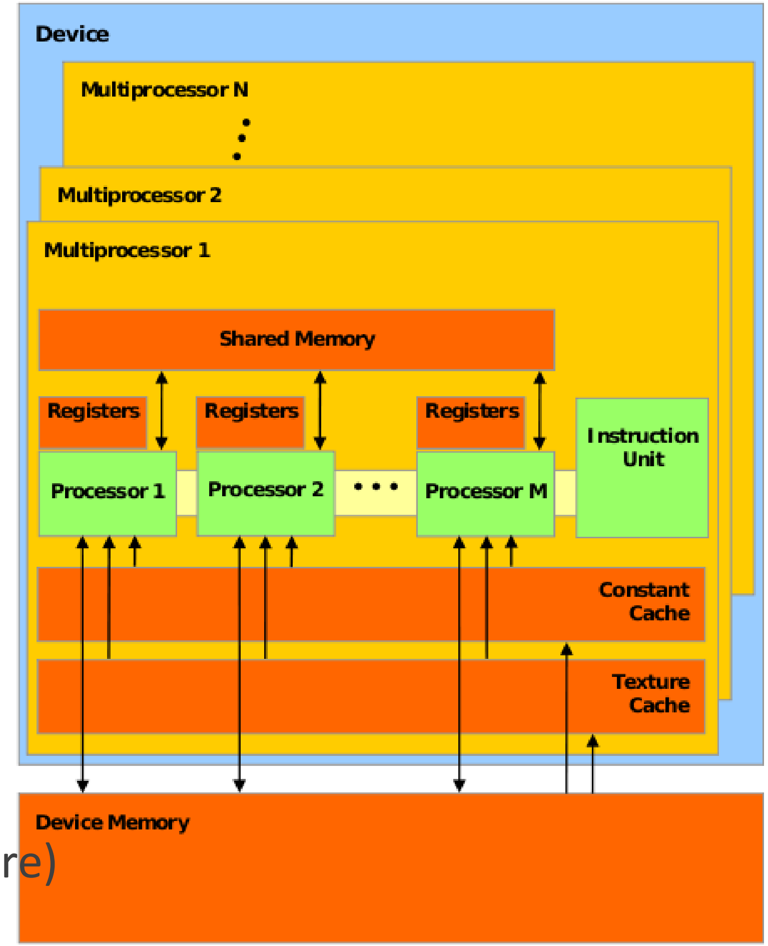


Inside a GPU

Maximum number of threads per block count is usually 512 or 1024 depending on the machine

Maximum number of blocks per grid is usually 65535

- If you go over either of these numbers your GPU will just give up or output garbage data
- Much of GPU programming is dealing with this kind of hardware limitations! Get used to it
- This limitation also means that your Kernel must compensate for the fact that you may not have enough threads to individually allocate to your data points
  - Will show how to do this later (this lecture)

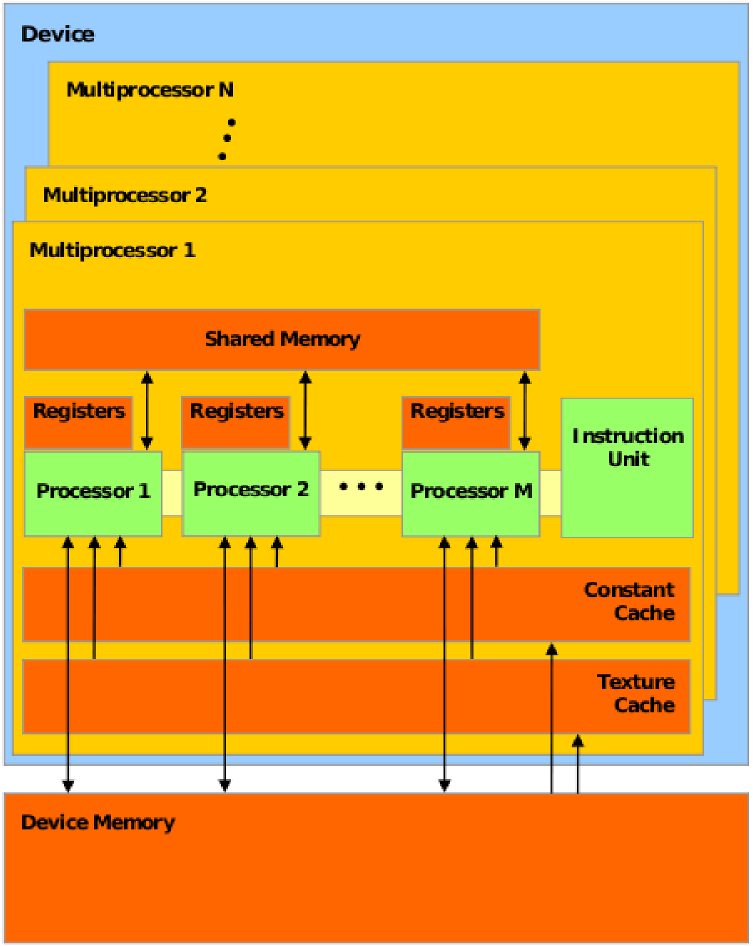


Inside a GPU

Each block is assigned to an SM

Inside the SM, the block is divided into **Warps** of threads

- Warps consist of 32 threads
- All 32 threads MUST run the exact same set of instructions at the same time
  - Due to the fact that there is only one instruction unit
- Warps are run concurrently in an SM
- If your Kernel tries to have threads do different things in a single warp (using if statements for example), the two tasks will be run sequentially
  - Called **Warp Divergence** (NOT GOOD)



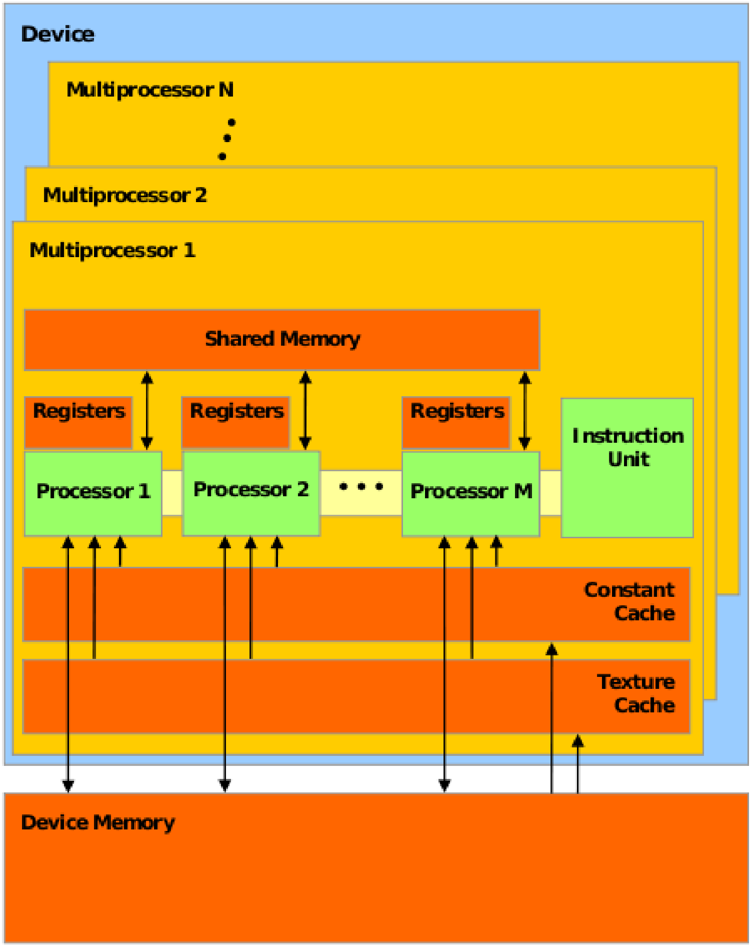
Inside a GPU (fun hardware info)

In Fermi Architecture (i.e. GPUs with Compute Capability 2.x), each SM has 32 cores

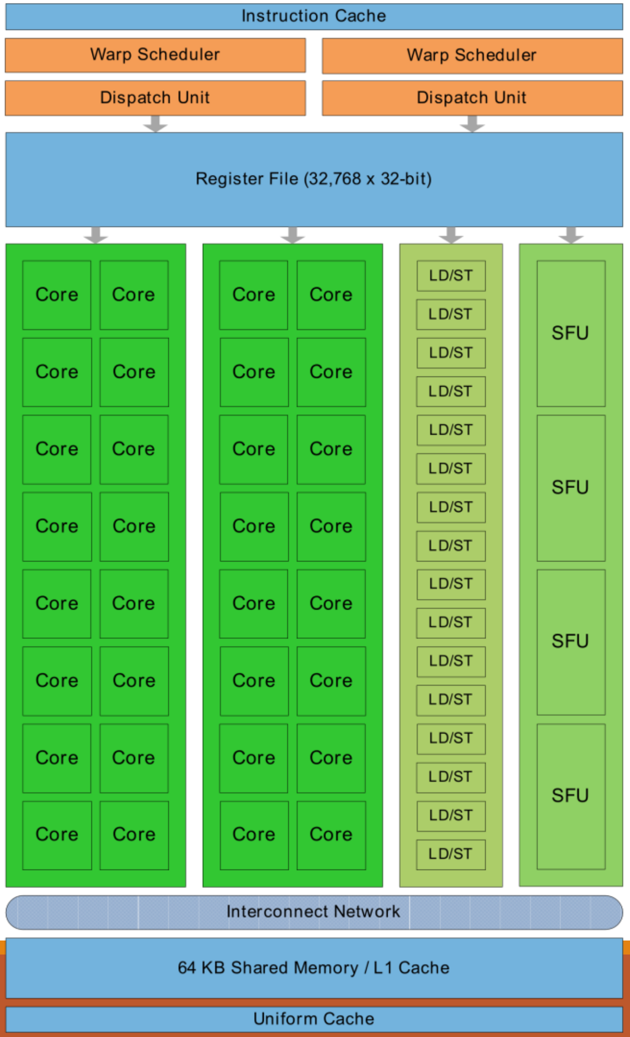
- e.g. GTX 400, 500 series
- 32 cores is not what makes each warp have 32 threads. Previous architecture also had 32 threads per warp but had less than 32 cores per SM

Halo.cms.caltech.edu has 3 GTX 570s

- This course will cover CC 2.x



Streaming Multiprocessor



A[] + B[] -> C[] (again)

~/Documents/test/test.cpp - Sublime Text (UNREGISTERED)

test.cpp | cuda_test.cu | test.hpp | cuda_test.cuh

```cpp
#include "test.hpp"

#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <assert.h>
#include <cmath>
#include <cuda_runtime.h>
#include "cuda_test.cuh"

using namespace std;

int main(int argc, char **argv) {
    /* setup block size and max block count */
    const uint per_block_thread_count = 1024;
    const uint max_block_count = 65535;

    /* setup host memory */
    const uint array_size = 10000000;
    float *a = new float[array_size];
    float *b = new float[array_size];
    float *c = new float[array_size];
    // fill a and b
    for (uint i = 0; i < array_size; i++) {
        a[i] = i;
        b[i] = array_size - i;
    }

    /* setup device memory */
    float *dev_a;
    float *dev_b;
    float *dev_c;
```

Line 27, Column 6 | Spaces: 4 | C++

A[] + B[] -> C[] (again)

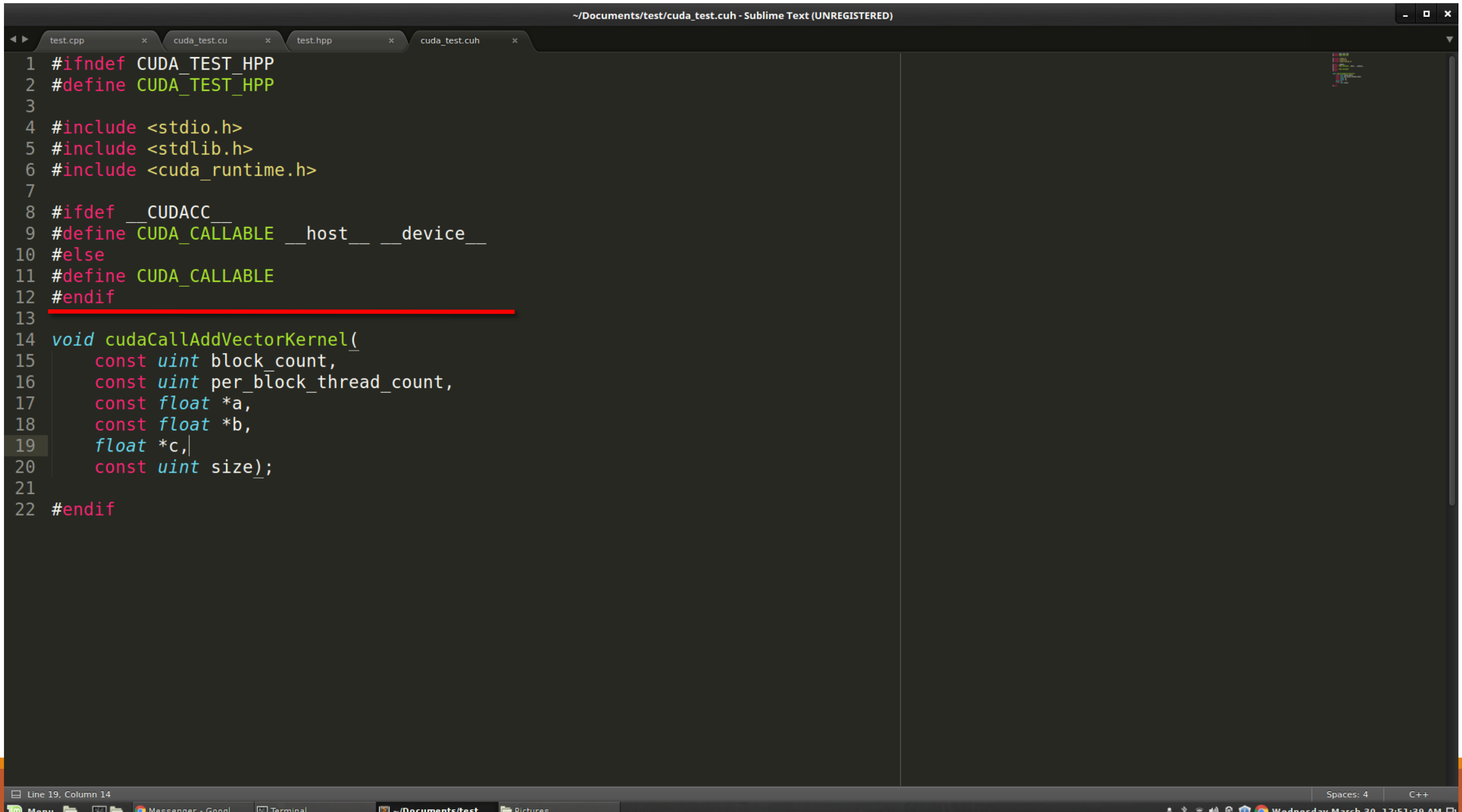
```
    /* setup device memory */
    float *dev_a;
    float *dev_b;
    float *dev_c;
    cudaMalloc((void**) &dev_a, array_size * sizeof(float));
    cudaMalloc((void**) &dev_b, array_size * sizeof(float));
    cudaMalloc((void**) &dev_c, array_size * sizeof(float));

    /* copy a and b into dev_a and dev_b */
    cudaMemcpy(dev_a, a, array_size * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, array_size * sizeof(float), cudaMemcpyHostToDevice);

    /* call kernel to add the two arrays into dev_c */
    uint block_count = min(max_block_count,
        (uint) ceil(array_size / (float) per_block_thread_count));
    cudaCallAddVectorKernel(
        block_count,
        per_block_thread_count,
        dev_a,
        dev_b,
        dev_c,
        array_size);

    /* copy dev_c into c */
    cudaMemcpy(c, dev_c, array_size * sizeof(float), cudaMemcpyDeviceToHost);

    /* check the output */
    for (uint i = 0; i < array_size; i++) {
        assert(c[i] == array_size);
    }

    /* free device memory */
    delete[] a;
    delete[] b;
    delete[] c;
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

A[] + B[] -> C[] (again)

```
#include "cuda_test.cuh"

__global__
void cudaAddVectorKernel(
    const float *a,
    const float *b,
    float *c,
    const uint size)
{
    /* get current thread's id */
    uint thread_index = blockIdx.x * blockDim.x + threadIdx.x;

    /* while this thread is dealing with a valid index */
    while (thread_index < size) {
        /* add a and b into c */
        c[thread_index] = a[thread_index] + b[thread_index];

        /* advance thread id */
        thread_index += blockDim.x * gridDim.x;
    }
}

void cudaCallAddVectorKernel(
    const uint block_count,
    const uint per_block_thread_count,
    const float *a,
    const float *b,
    float *c,
    const uint size)
{
    cudaAddVectorKernel<<<block_count, per_block_thread_count>>>(a, b, c, size);
}
```

Questions so far?

Stuff that will be useful later



Stuff that will be useful later

~/Documents/repos/cs81/src/swarm.cuh - Sublime Text (UNREGISTERED)

test.cpp | cuda_test.cu | swarm.cuh | test.hpp | cuda_test.cuh

```cpp
#ifndef SWARM_CUH
#define SWARM_CUH

#include <cstdio>
#include <cuda_runtime.h>

#include "constants.hpp"
#include "utils.hpp"

struct GPUVec3d {
    double x;
    double y;
    double z;

    CUDA_CALLABLE GPUVec3d();
    CUDA_CALLABLE GPUVec3d(double x, double y, double z);
    CUDA_CALLABLE GPUVec3d(const GPUVec3d &v);
    CUDA_CALLABLE ~GPUVec3d();

    CUDA_CALLABLE double sqNorm();

    CUDA_CALLABLE const GPUVec3d &operator=(const GPUVec3d& rhs);
    CUDA_CALLABLE bool operator==(const GPUVec3d& rhs) const;
    CUDA_CALLABLE bool operator!=(const GPUVec3d& rhs) const;
    CUDA_CALLABLE const GPUVec3d &operator+=(const GPUVec3d &rhs);
    CUDA_CALLABLE const GPUVec3d &operator-=(const GPUVec3d &rhs);
    CUDA_CALLABLE const GPUVec3d &operator*=(const double d);
    CUDA_CALLABLE const GPUVec3d &operator/=(const double d);
    CUDA_CALLABLE friend const GPUVec3d operator+(const GPUVec3d &lhs,
        const GPUVec3d &rhs);
    CUDA_CALLABLE friend const GPUVec3d operator-(const GPUVec3d &lhs,
        const GPUVec3d &rhs);
    CUDA_CALLABLE friend const GPUVec3d operator*(const GPUVec3d &lhs,
        const double d);
    CUDA_CALLABLE friend const GPUVec3d operator*(const double d,
        const GPUVec3d &rhs);
    CUDA_CALLABLE friend const GPUVec3d operator/(const GPUVec3d &lhs,
        const double d);
};

class SwarmParticle;

```

Line 1, Column 1 | Spaces: 4 | C++

Menu | Edit cs179_2015_le... | ~/Documents/rep... | Pictures | Wednesday March 30, 4:11:19 AM

Stuff that will be useful later

```
#include "cuda_test.cuh"

__global__
void cudaAddVectorKernel(
    const float *a,
    const float *b,
    float *c,
    const uint size)
{
    /* get current thread's id */
    uint thread_index = blockIdx.x * blockDim.x + threadIdx.x;

    /* while this thread is dealing with a valid index */
    while (thread_index < size) {
        /* add a and b into c */
        c[thread_index] = a[thread_index] + b[thread_index];

        /* advance thread id */
        thread_index += blockDim.x * gridDim.x;
    }
}

void cudaCallAddVectorKernel(
    const uint block_count,
    const uint per_block_thread_count,
    const float *a,
    const float *b,
    float *c,
    const uint size)
{
    cudaAddVectorKernel<<<block_count, per_block_thread_count>>>(a, b, c, size);
}
```

Next Time...

Global Memory access is not that fast

- Tends to be the bottleneck in many GPU programs
- Especially true if done stupidly
  - We'll look at what "stupidly" means

Optimize memory access by utilizing hardware specific memory access patterns

Optimize memory access by utilizing different caches that come with the GPU