CS 179: GPU Computing

LECTURE 2: INTRO TO THE SIMD LIFESTYLE AND GPU INTERNALS

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)

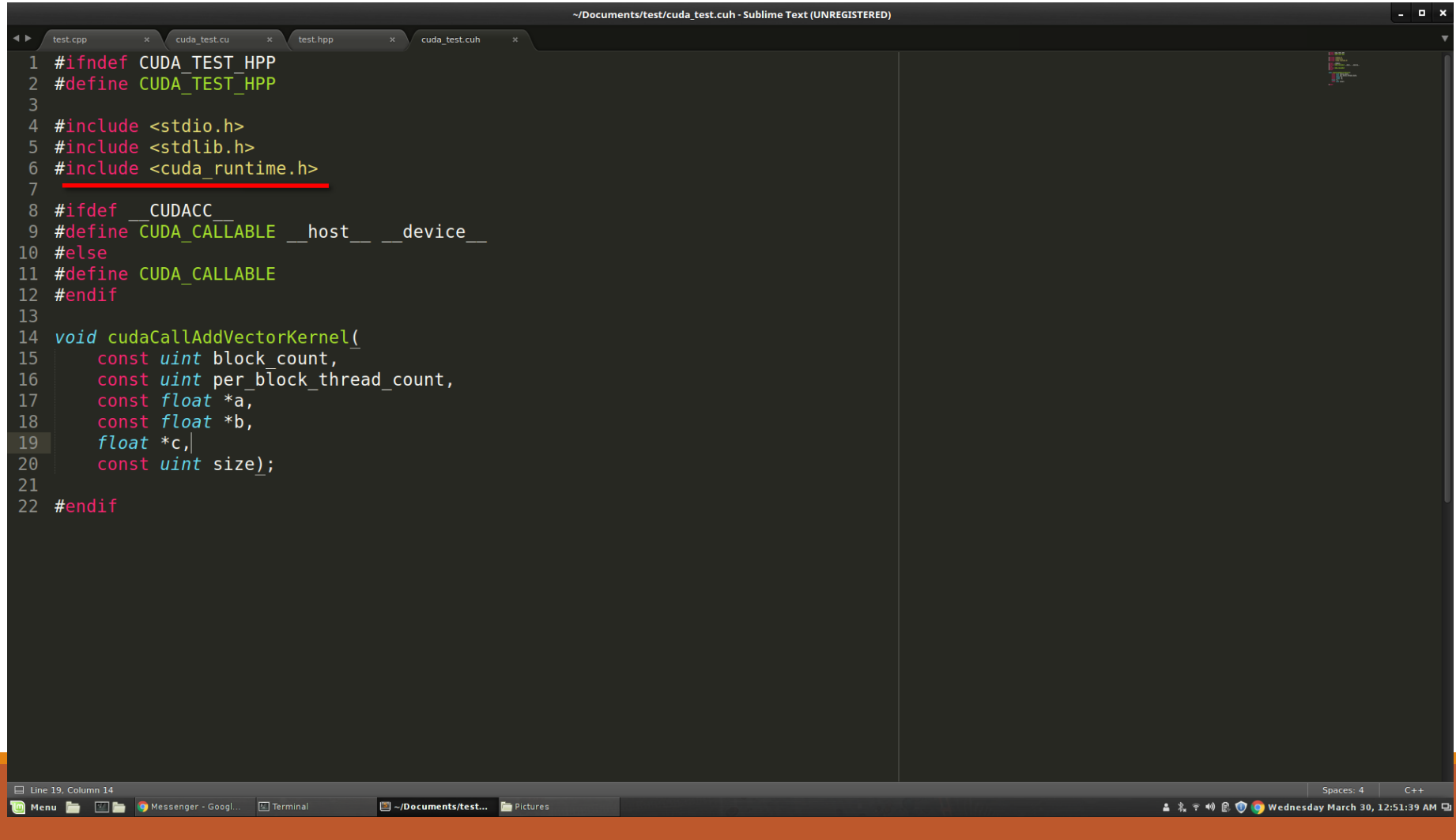
Looked at the a[] + b[] -> c[] example

Recap

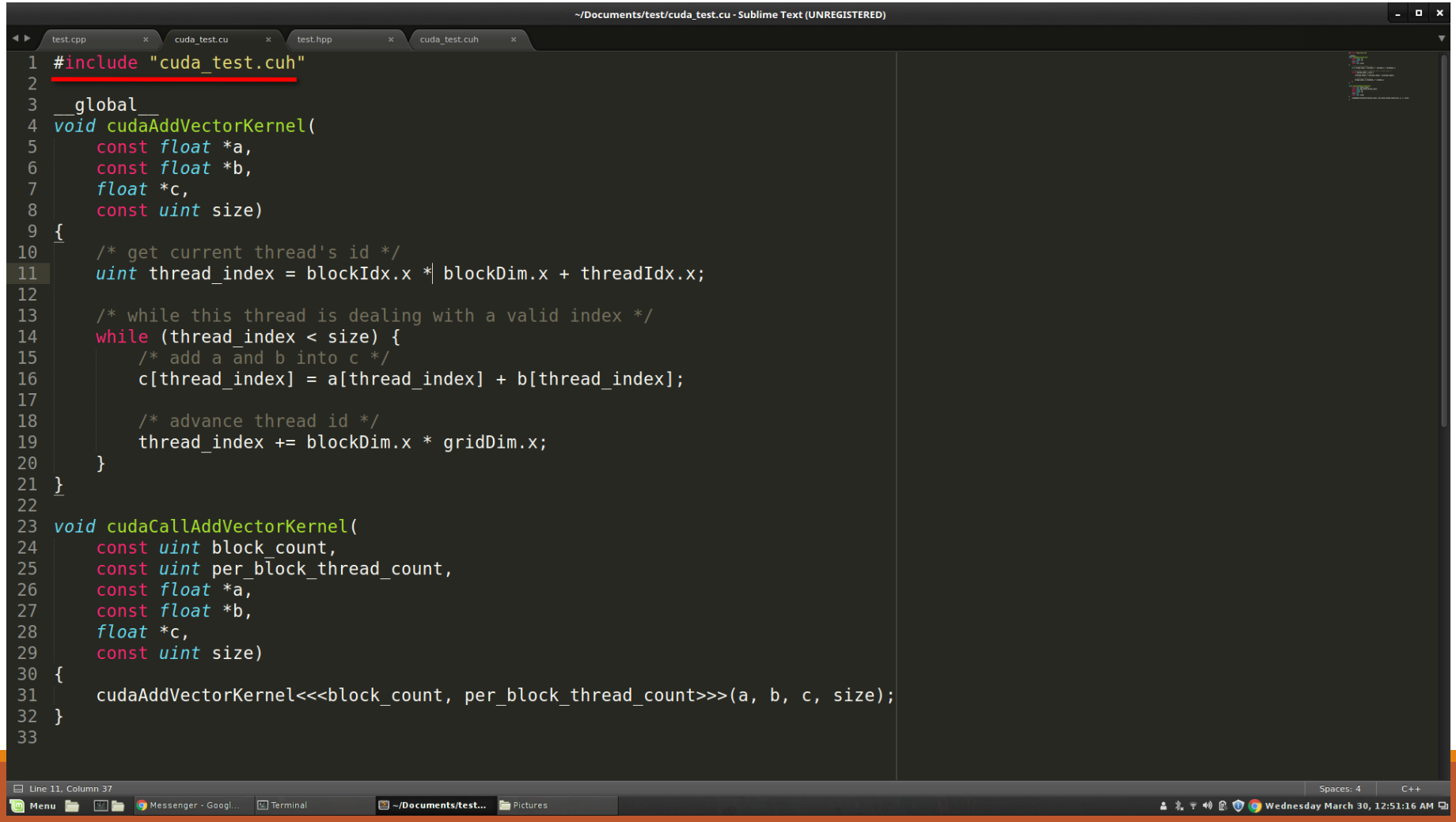
If you forgot everything, just make sure you understand that CUDA is simply an extension of other bits of code you write!!!!

- 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

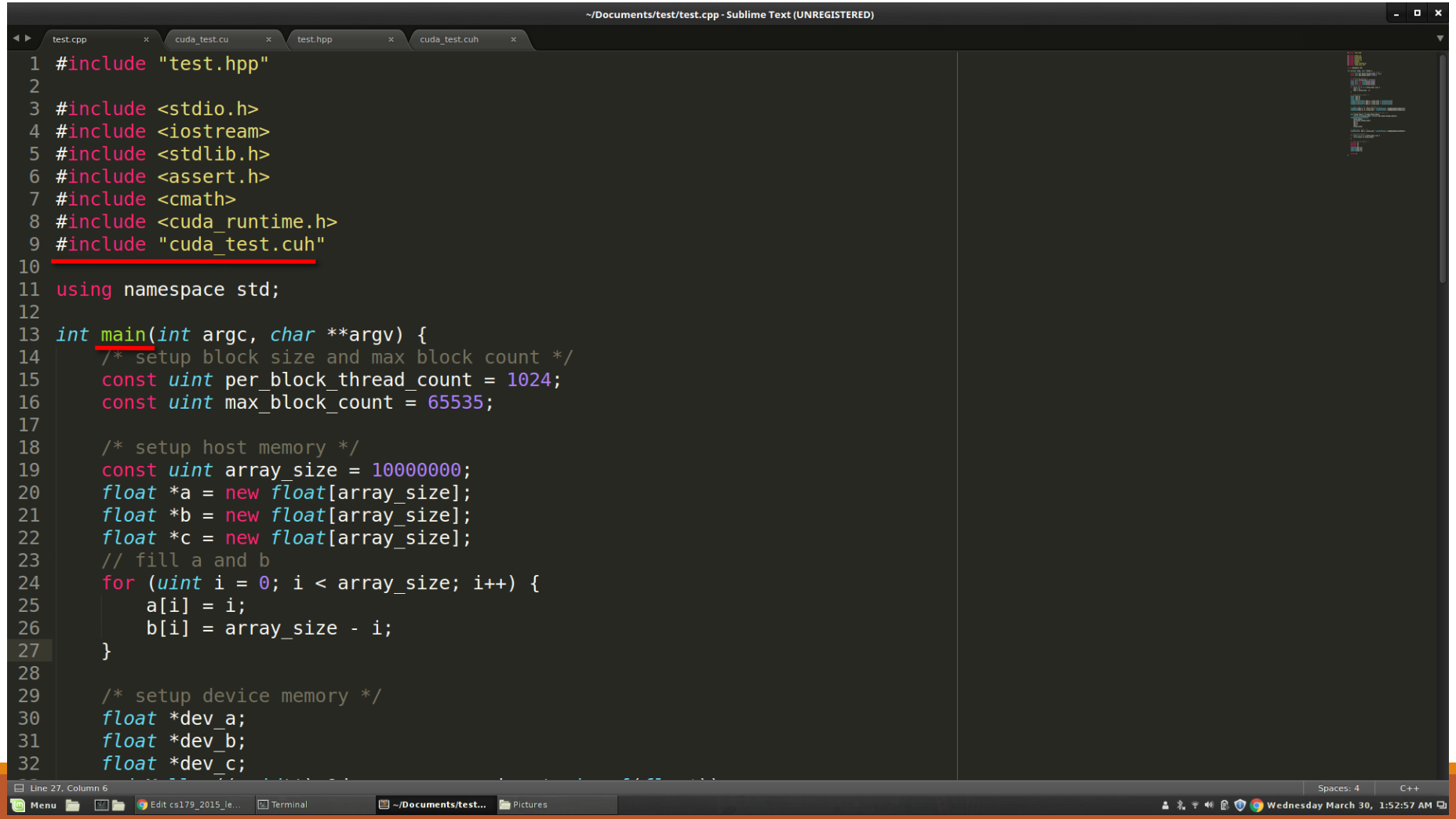
.cu/.cuh vs .cpp/.hpp



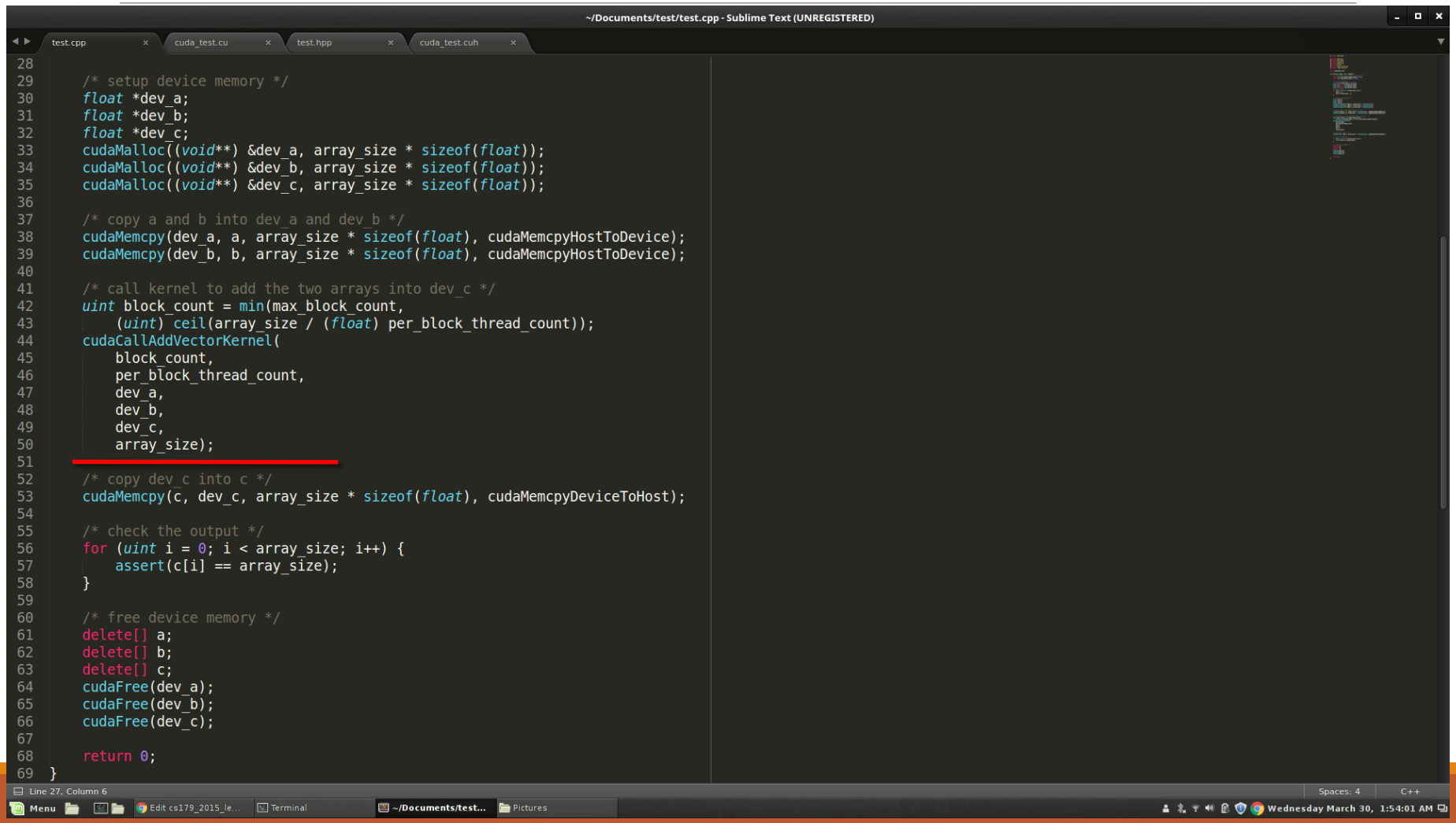
.cu/.cuh vs .cpp/.hpp



.cu/.cuh vs .cpp/.hpp

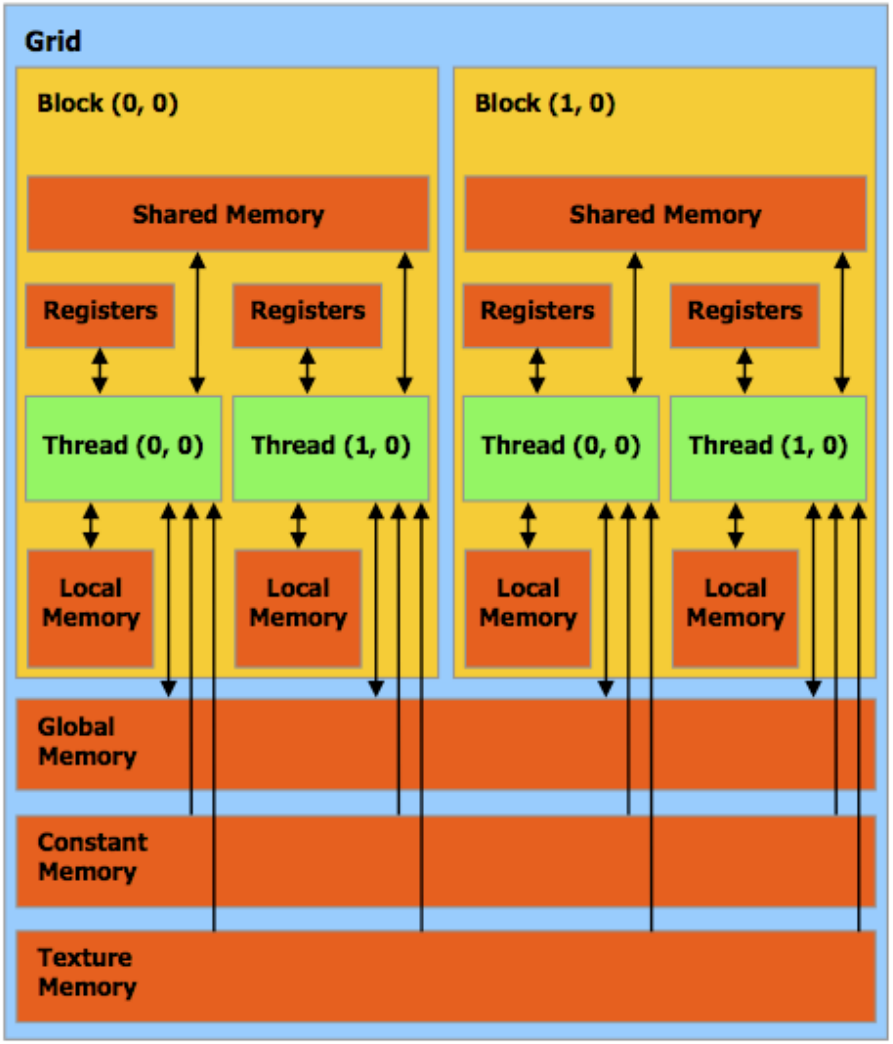


.cu/.cuh vs .cpp/.hpp



Thread Block Organization Keywords

- Keywords you MUST know to code in CUDA:
  - Thread - Distributed by the CUDA runtime (threadIdx)
  - Block - A user defined group of 1 to ~512 threads (blockIdx)
  - Grid - A group of one or more blocks. A grid is created for each CUDA kernel function called



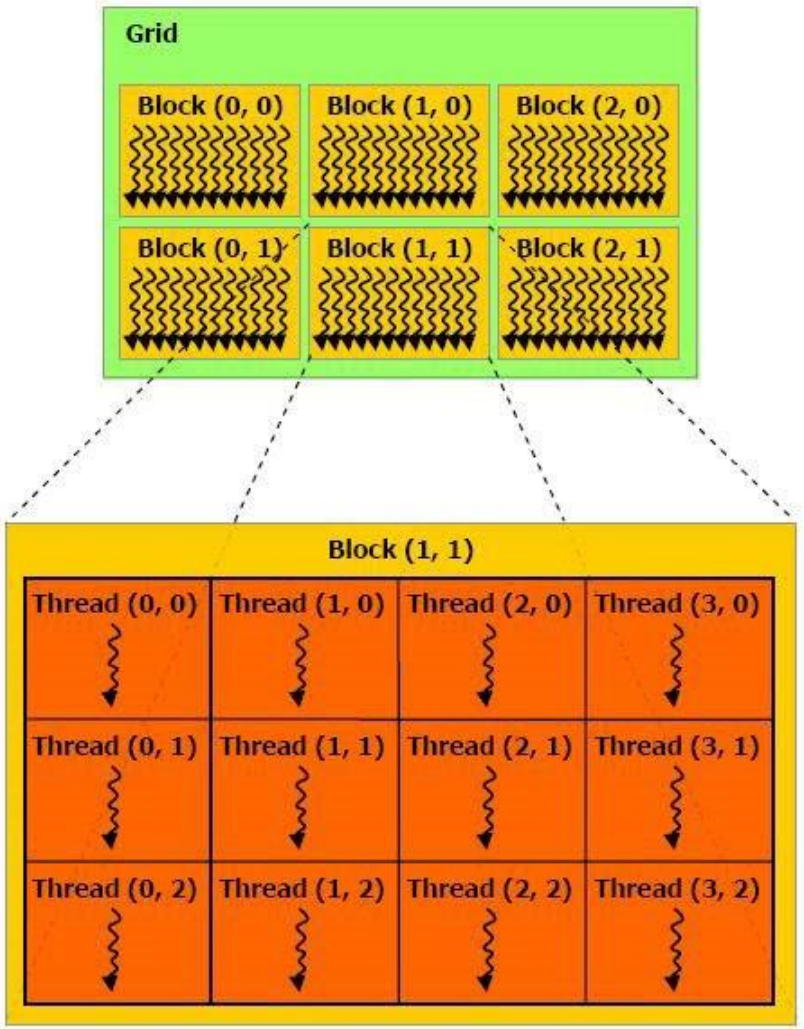
Block and Grid Dimensions

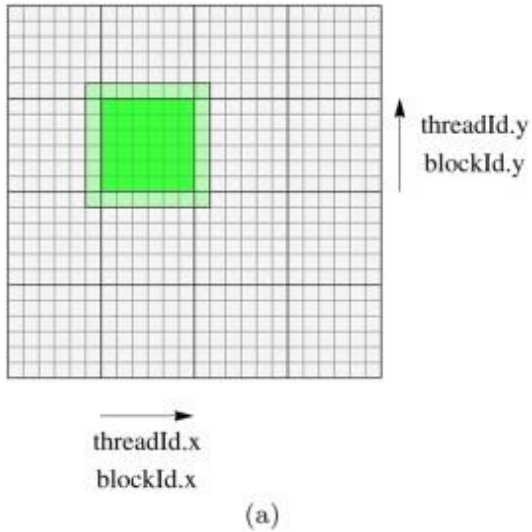
You can use a struct (defined in vector_types.h) called dim3 to define your Grid and Block dimensions.

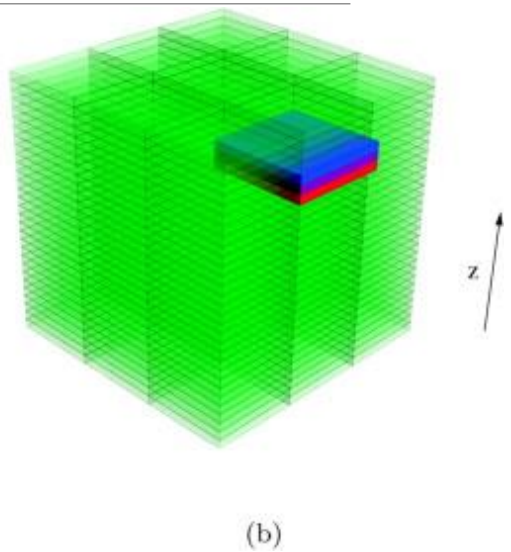
```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#if defined(__cplusplus)
    __host__ __device__ dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
    __host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
    __host__ __device__ operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
#endif /* __cplusplus */
};
```

- dim3 grid(256); // defines a grid of 256 x 1 x 1 blocks
- dim3 block(512, 512); // defines a block of 512 x 512 x 1 threads
- foo<<<grid, block>>>(...);

Grid/Block/Thread Visualized







Single Instruction, Multiple Data (SIMD)

- SIMD describes a class of instructions which perform the same operation on multiple registers simultaneously.
- Example: Add some scalar to 3 registers, storing the output for each addition in those registers.
  - Used to increase the brightness of a pixel
- CPUs also have SIMD instructions and are very important for applications that need to do a lot of number crunching
  - Video codecs like x264/x265 make extensive use of SIMD instructions to speed up video encoding and decoding.

SIMD continued

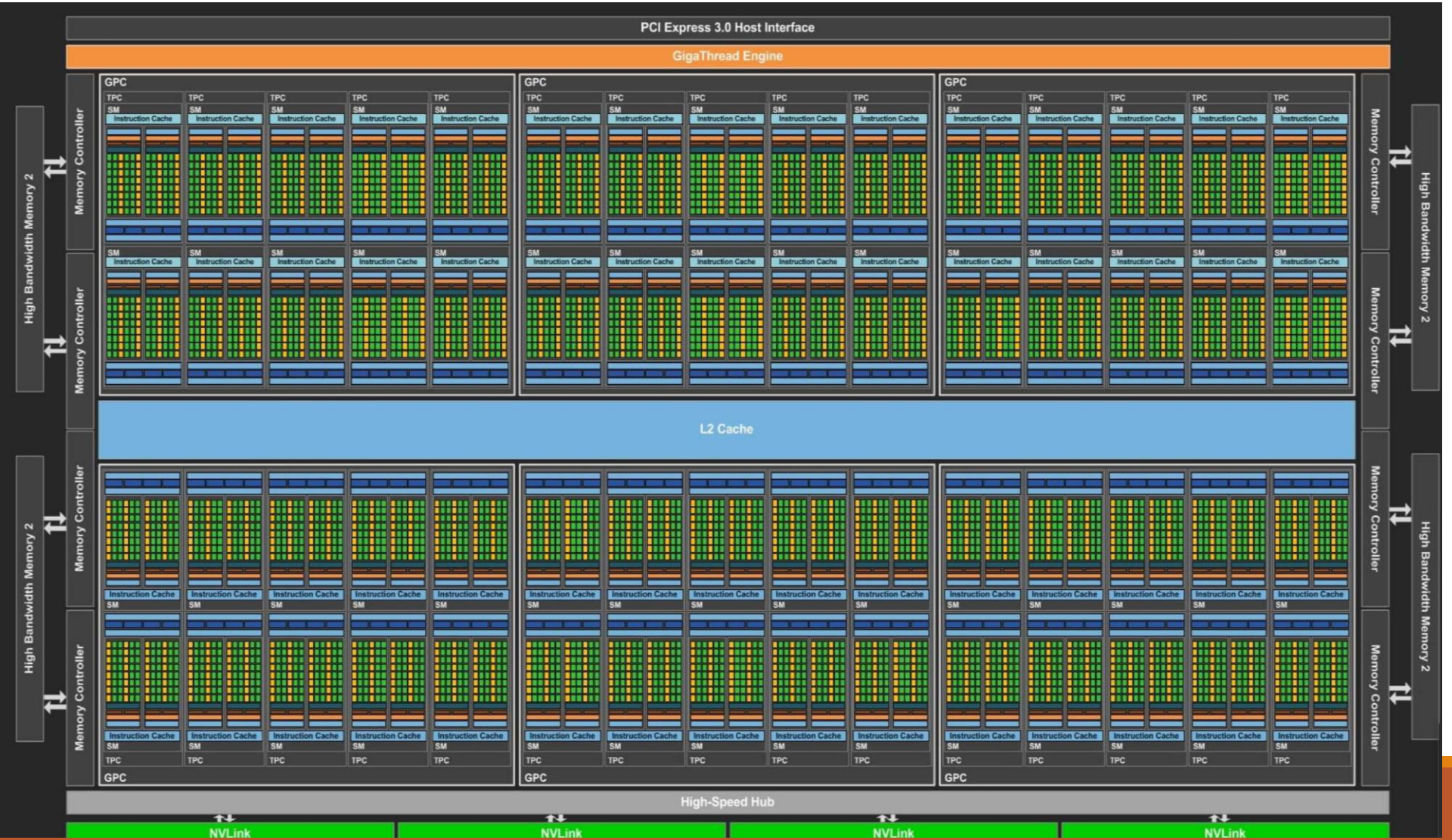
- Converting an algorithm to use SIMD is usually called “Vectorizing”
  - Not every algorithm can benefit from this or even be vectorized at all, e.x. Parsing.
  - Using SIMD instructions is not always beneficial though.
    - Even using the SIMD hardware requires additional power, and thus waste heat.
    - If the gains are small it probably isn’t worth the additional complexity.
  - Optimizing compilers like GCC and LLVM are still being trained to be able to vectorize code usefully, though there has been many exciting developments on this front in the last 2 years and is an active area of study.
    - https://polly.llvm.org/

Thread blocks and Warps visualized

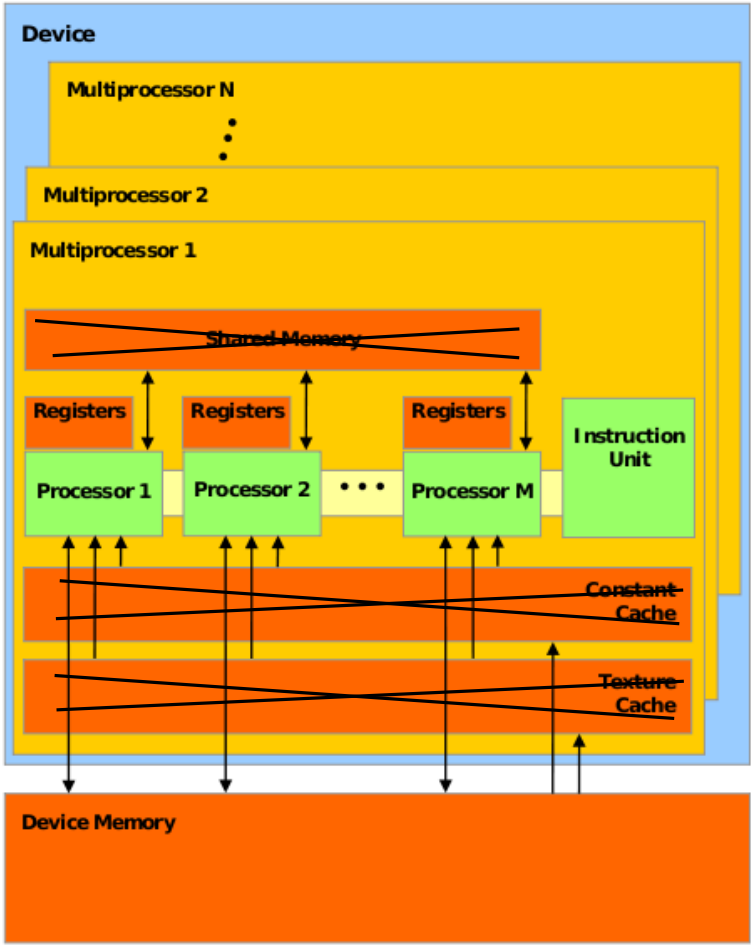
Keywords you MUST know to code WELL in CUDA

- Streaming Multiprocessor – Each contains ~128 CUDA cores (which execute a thread) and their associated cache.
- Warp - A scheduling unit of up to 32 threads (all within the same block)
- Warp Divergence – A condition where threads within a warp need to execute different instructions in order to continue executing their kernel.
  - Causes threads to execute sequentially, in most cases ruining parallel performance
  - As of the Kepler (2012) architecture each Warp can have at most 2 branches, starting with Volta (2017) this condition has been nearly eliminated. For this class assume your code must only branch at most twice as we are not yet allocating Volta GPUs to this class.

What a modern GPU looks like



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

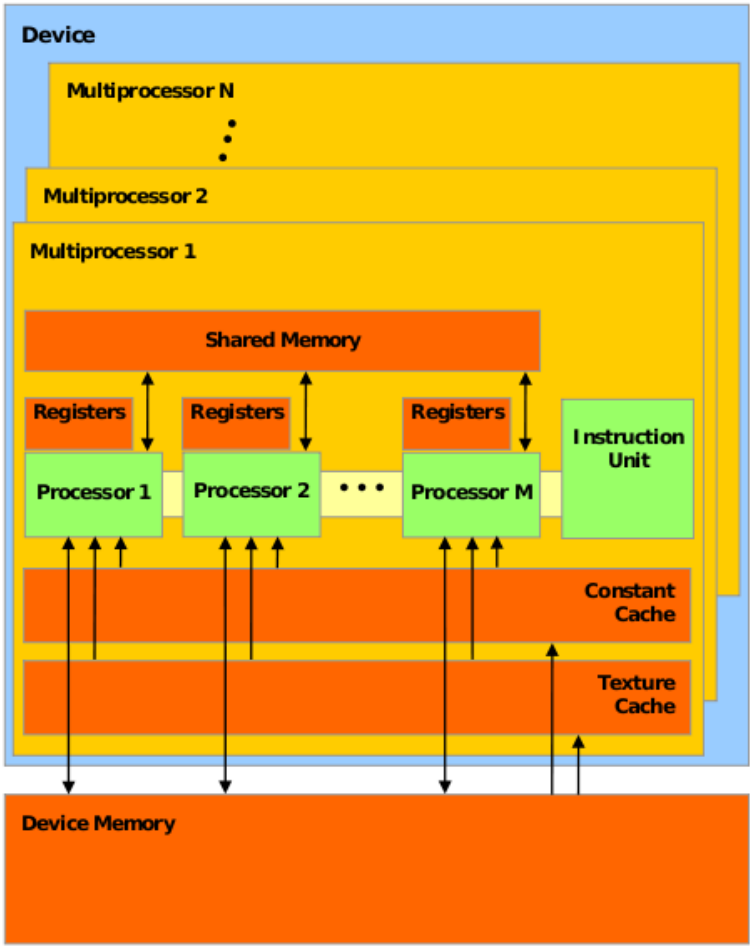
Inside a GPU

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

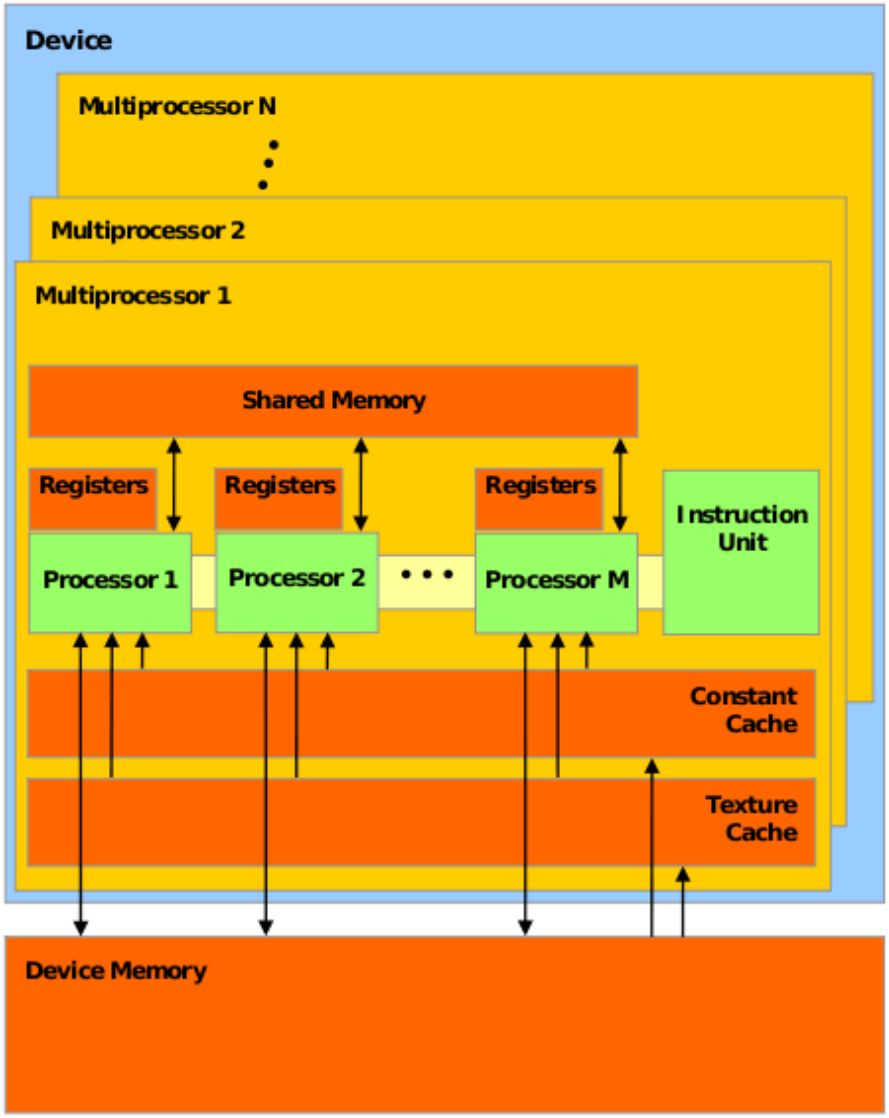
- Each thread handles a small portion of the given task

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>>>(…);

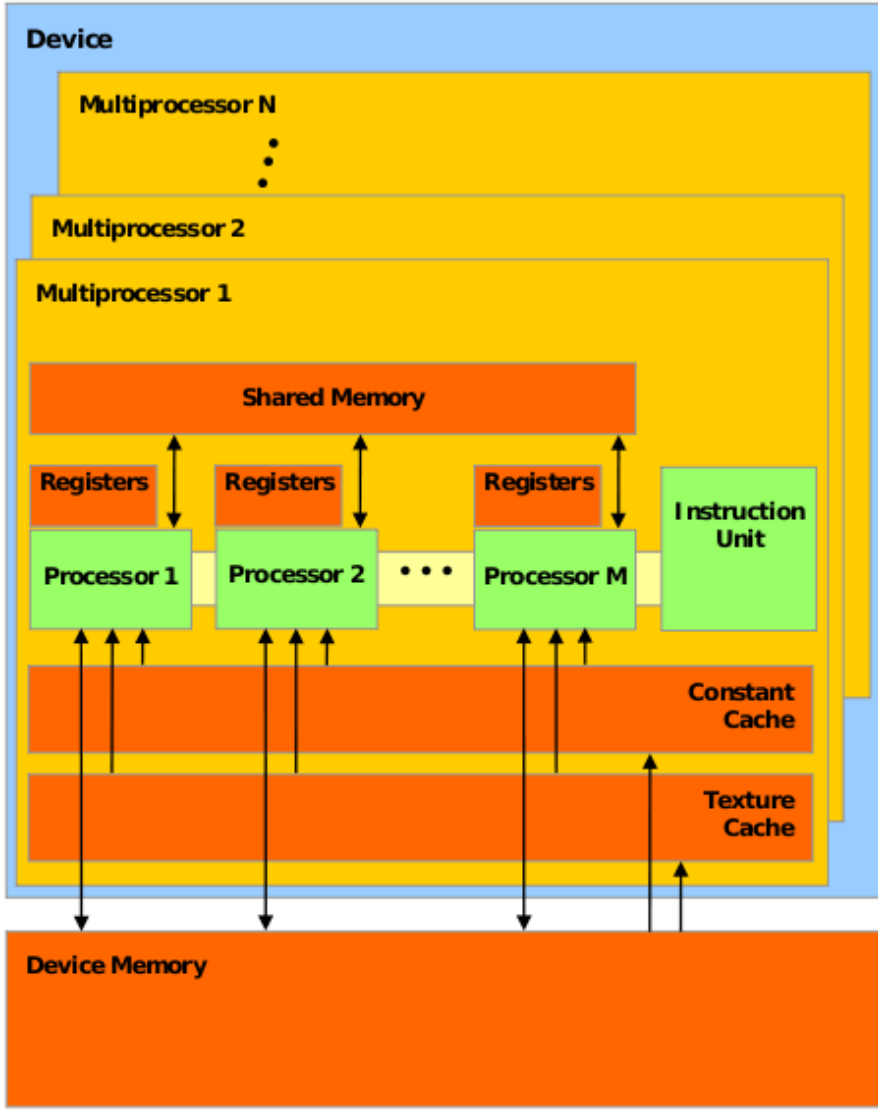


Inside a GPU

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)

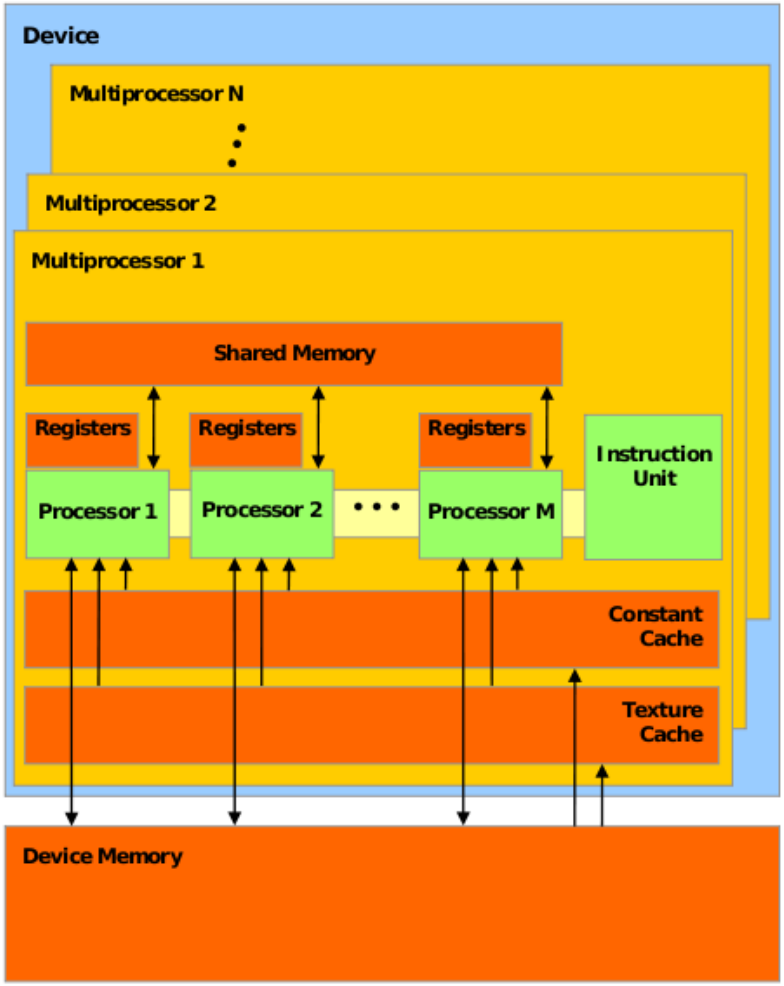


Inside a GPU

Each block is assigned to an SM

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

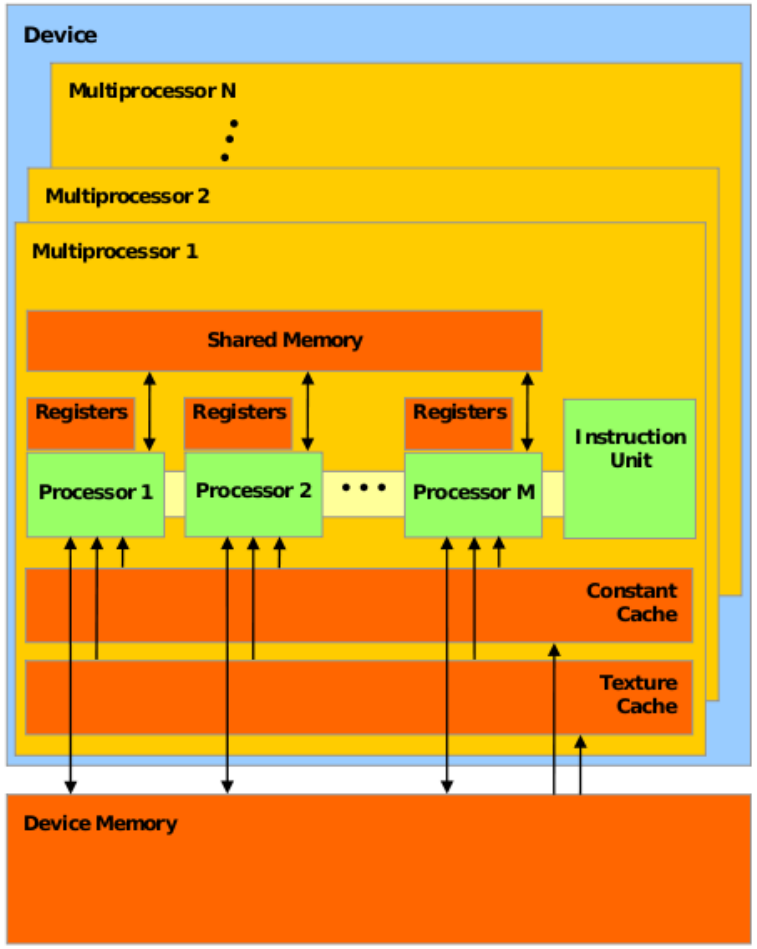
- Warps consist of 32 threads
- All 32 threads MUST run the exact same set of instructions at the same time
  - Due to the fact that there is only one instruction unit
- Warps are run concurrently in an SM
- If your Kernel tries to have threads do different things in a single warp (using if statements 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, later architectures have more.

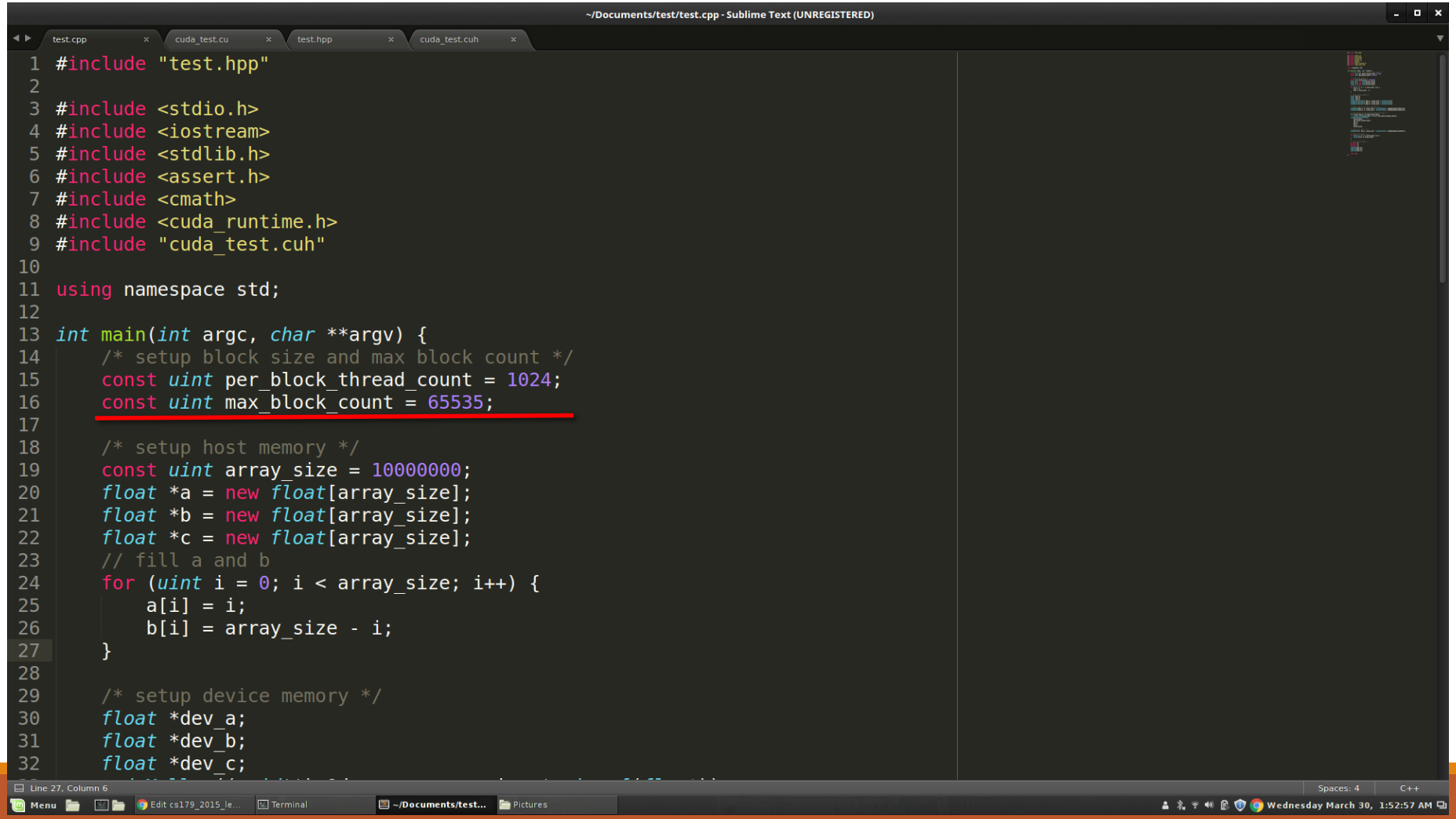
- e.g. GTX 400, 500 series
- 32 cores is not what makes each warp have 32 threads. Previous architecture also had 32 threads per warp but had less than 32 cores per SM
- Some early Pascal (2016) GPUs (GP100) had 64 cores per SM, but later chips in that generation (GP104) had 128 core model.



Streaming Multiprocessor

- Shown here is a Pascal GP104 GPU Streaming Multiprocessor that can be found in a GTX1080 graphics card.
- The exact amount of Cache and Shared Memory differ between GPU models, and even more so between different architectures.
  - Whitepapers with exact information can be gotten from Nvidia (use Google)
  - https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf
  - http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf
  - “nvidia kepler whitepaper”

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


~/Documents/test/test.cpp - Sublime Text (UNREGISTERED)
test.cpp
cuda_test.cu
test.hpp
cuda_test.cuh
#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++
Menu
Edit cs179_2015_le...
Terminal
~/Documents/test...
Pictures
Wednesday March 30, 1:52:57 AM

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

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

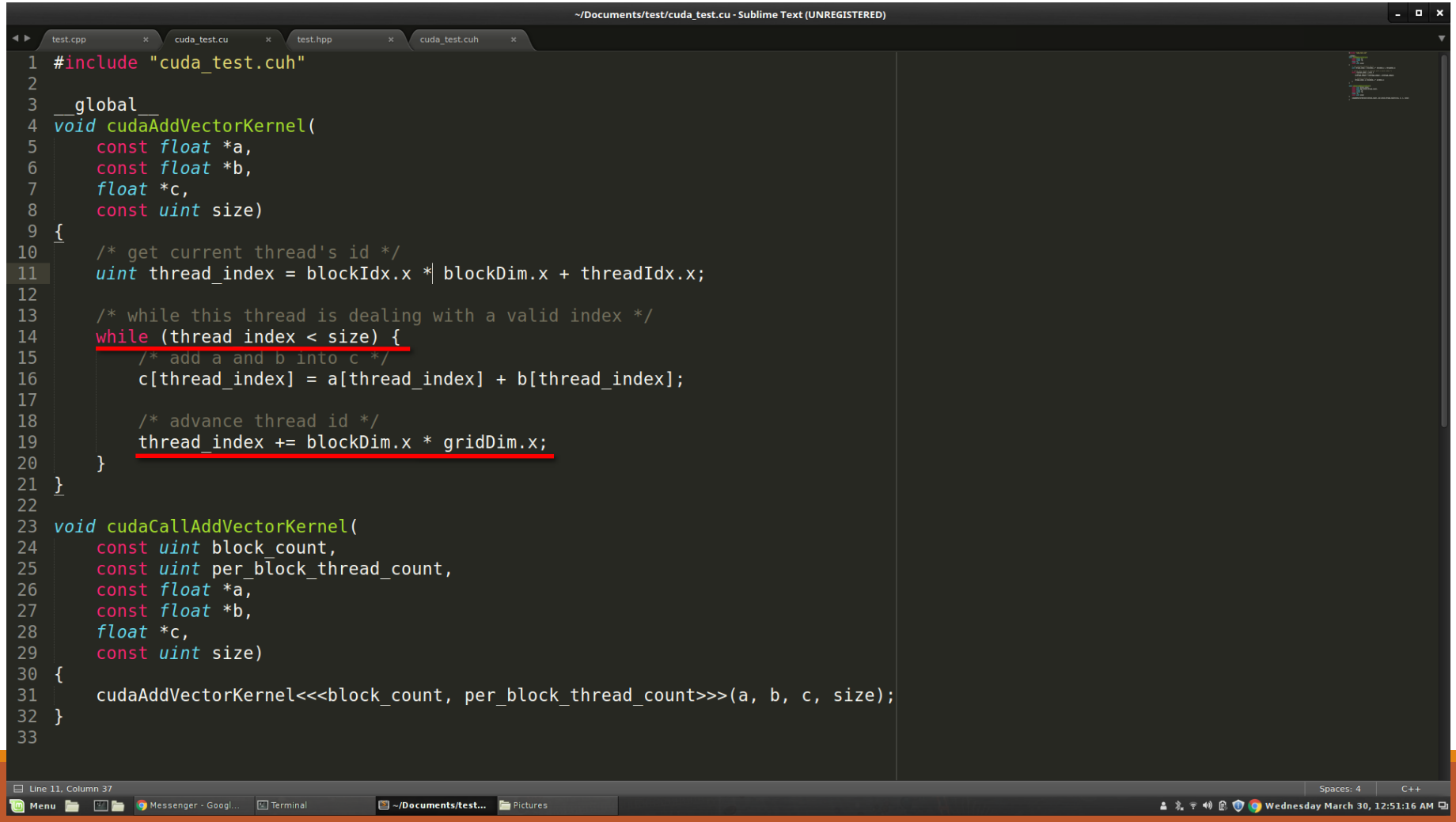
test.cpp | cuda_test.cu | test.hpp | cuda_test.cuh

```
28
29      /* setup device memory */
30      float *dev_a;
31      float *dev_b;
32      float *dev_c;
33      cudaMalloc((void**) &dev_a, array_size * sizeof(float));
34      cudaMalloc((void**) &dev_b, array_size * sizeof(float));
35      cudaMalloc((void**) &dev_c, array_size * sizeof(float));
36
37      /* copy a and b into dev_a and dev_b */
38      cudaMemcpy(dev_a, a, array_size * sizeof(float), cudaMemcpyHostToDevice);
39      cudaMemcpy(dev_b, b, array_size * sizeof(float), cudaMemcpyHostToDevice);
40
41      /* call kernel to add the two arrays into dev_c */
42      uint block_count = min(max_block_count,
43          (uint) ceil(array_size / (float) per_block_thread_count));
44      cudaCallAddVectorKernel(
45          block_count,
46          per_block_thread_count,
47          dev_a,
48          dev_b,
49          dev_c,
50          array_size);
51
52      /* copy dev_c into c */
53      cudaMemcpy(c, dev_c, array_size * sizeof(float), cudaMemcpyDeviceToHost);
54
55      /* check the output */
56      for (uint i = 0; i < array_size; i++) {
57          assert(c[i] == array_size);
58      }
59
60      /* free device memory */
61      delete[] a;
62      delete[] b;
63      delete[] c;
64      cudaFree(dev_a);
65      cudaFree(dev_b);
66      cudaFree(dev_c);
67
68      return 0;
69  }
```

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

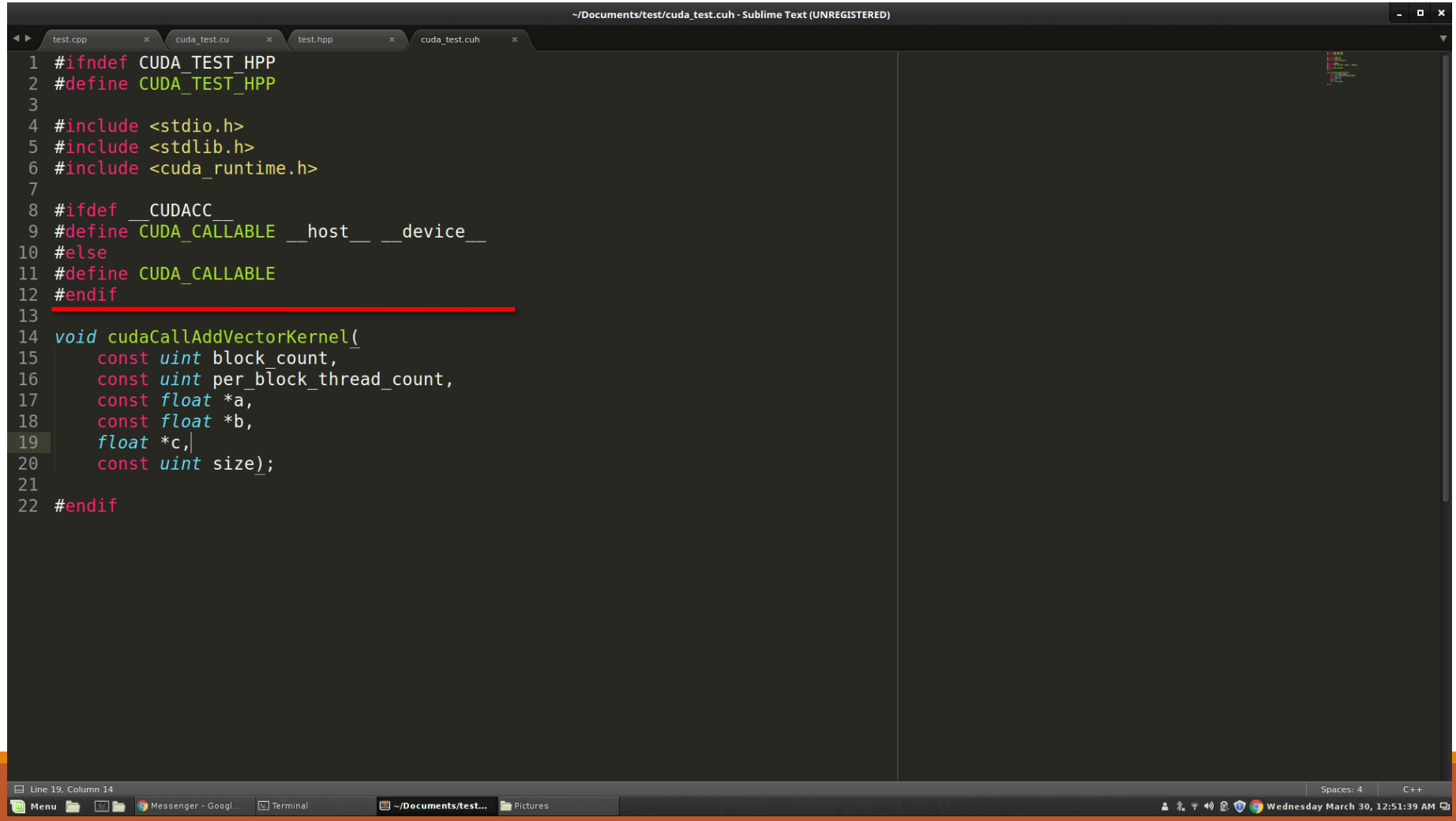
Menu | Edit cs179_2015_le... | Terminal | ~/Documents/test... | Pictures | Wednesday March 30, 1:54:01 AM

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

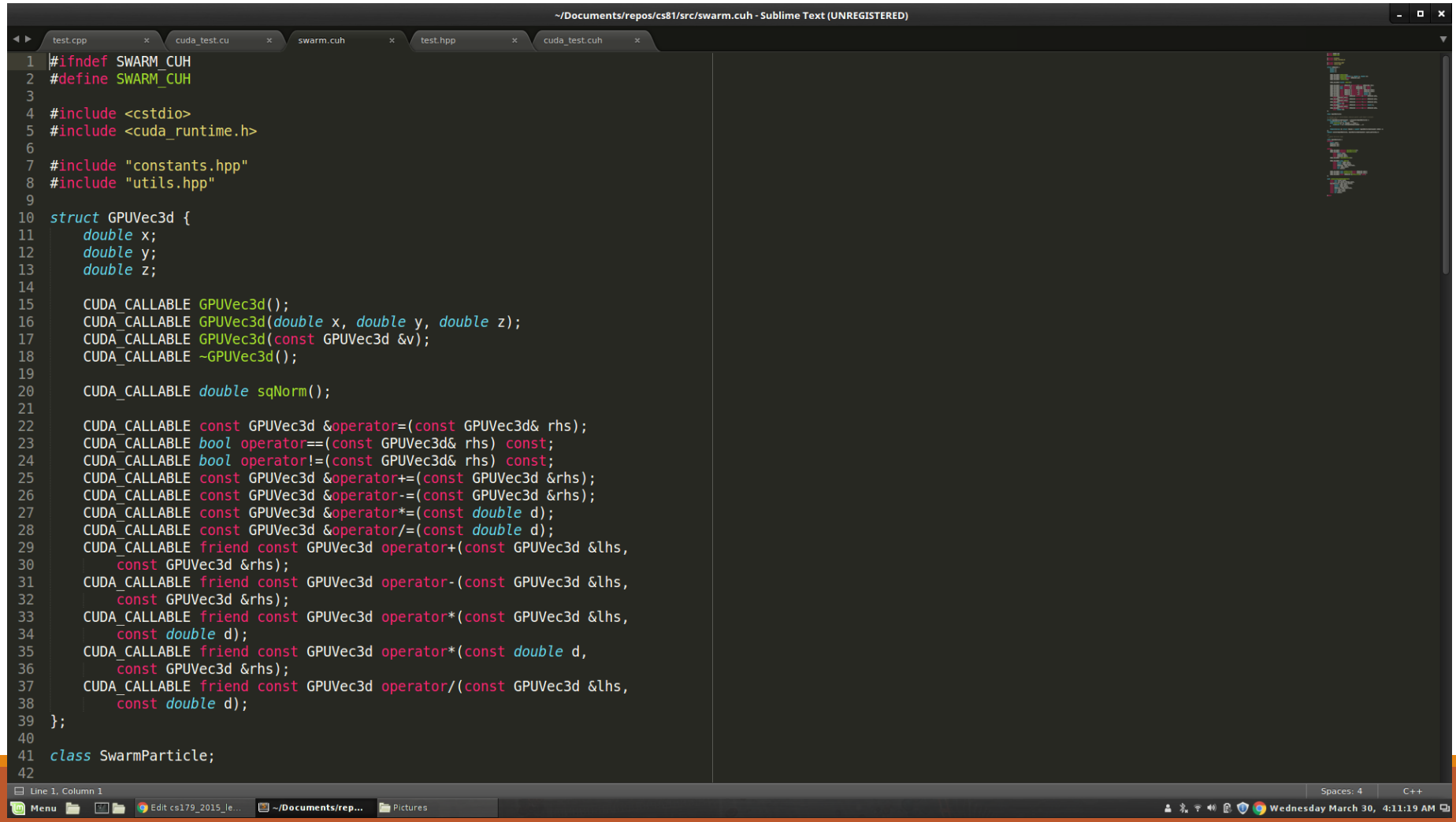

~/Documents/test/cuda_test.cu - Sublime Text (UNREGISTERED)
test.cpp
cuda_test.cu
test.hpp
cuda_test.cuh
#include "cuda_test.cuh"
__global__
void cudaAddVectorKernel(
const float *a,
const float *b,
float *c,
const uint size)
{
/* get current thread's id */
uint thread_index = blockIdx.x * blockDim.x + 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

Questions so far?

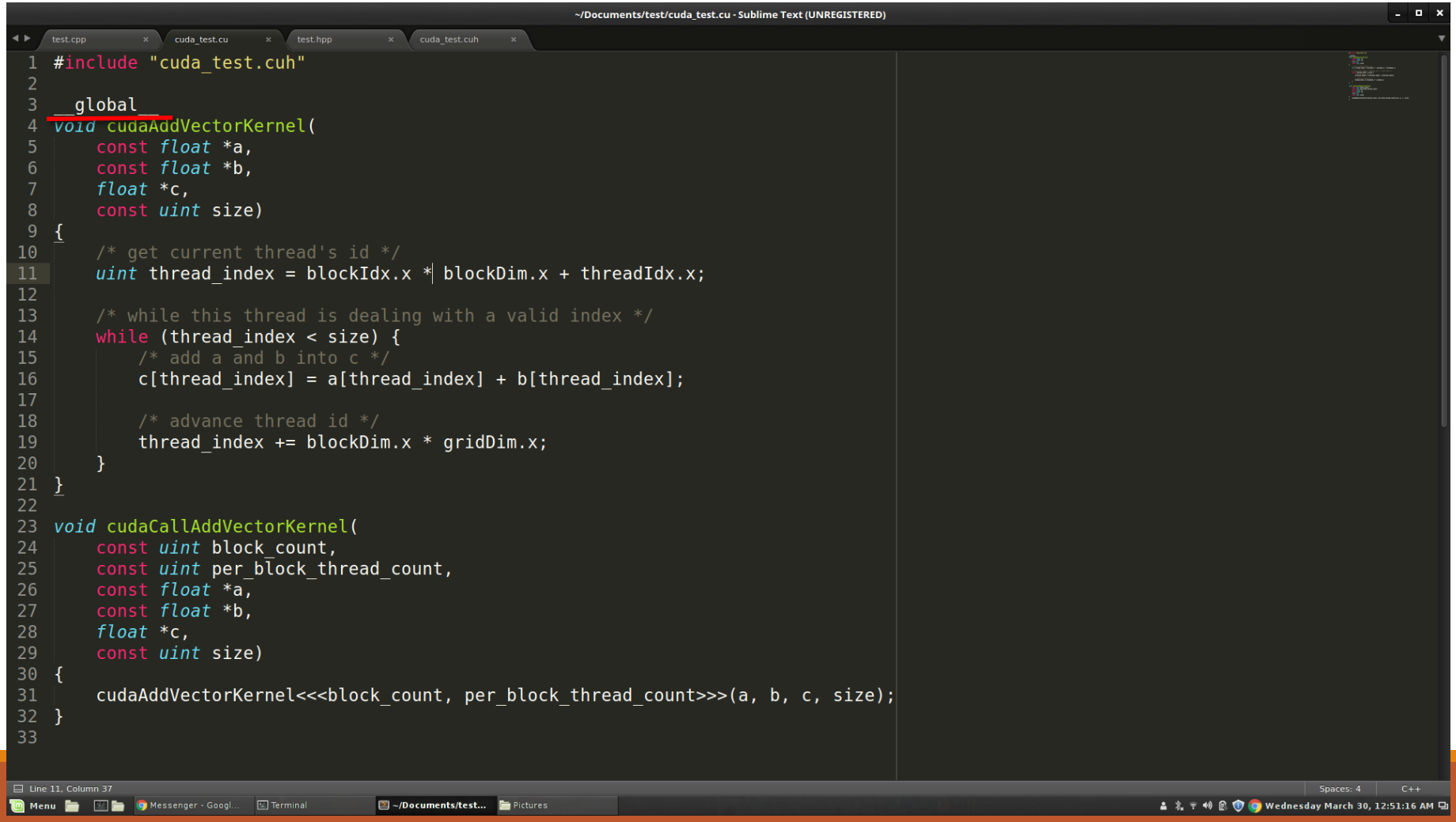
Stuff that will be useful later



Stuff that will be useful later



Stuff that will be useful later



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