

# CS 179: GPU Computing

---

LECTURE 2: INTRO TO THE SIMD  
LIFESTYLE AND GPU INTERNALS



# Recap

---

Can use GPU to solve highly parallelizable problems

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

CUDA is a straightforward extension to C++

- Separate CUDA code into `.cu` and `.cuh` files
- We compile with `nvcc`, NVIDIA's compiler for CUDA, to create object files (`.o` files)

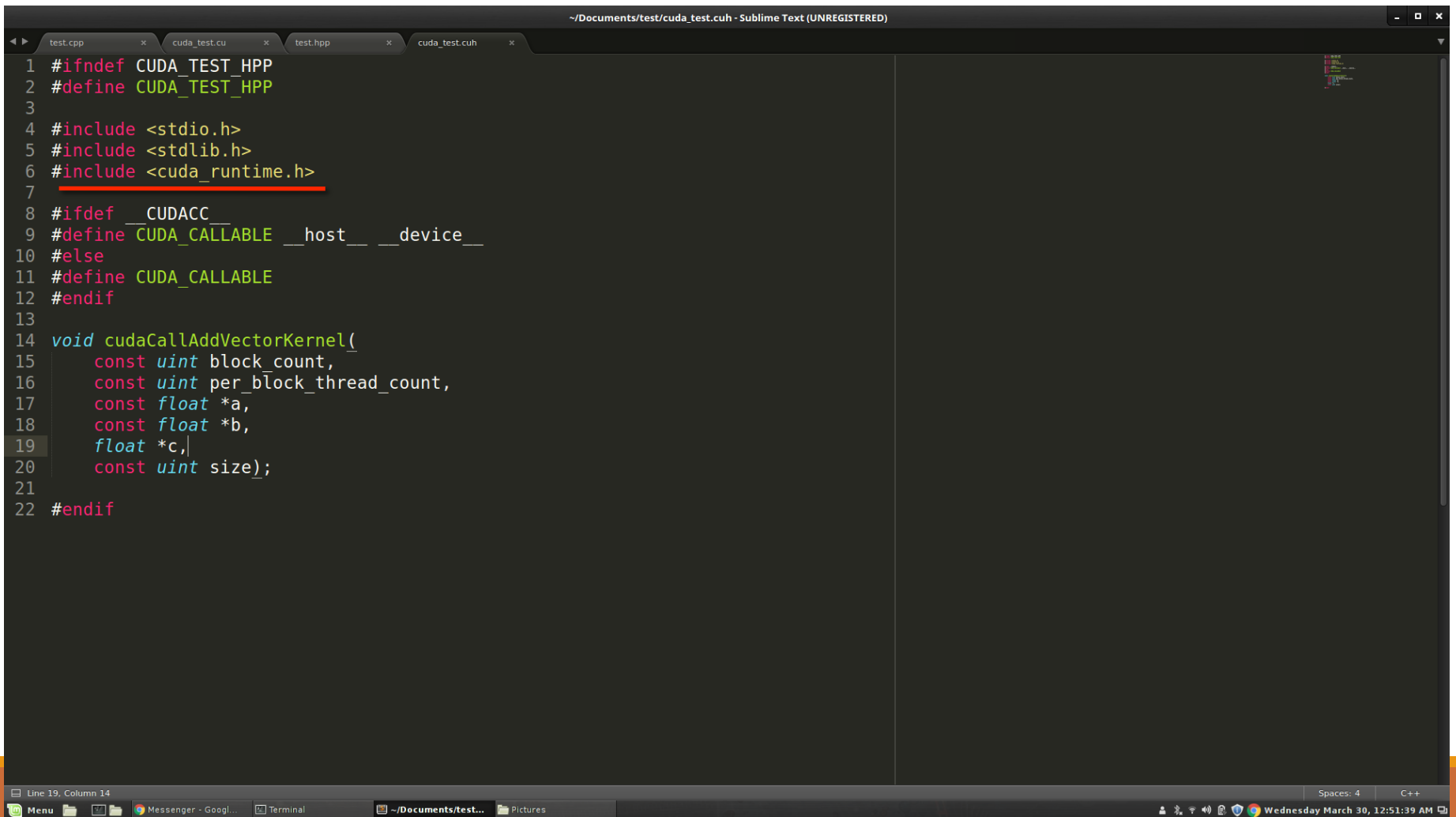
# NVCC and G++

---

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
  - Since CUDA 7.0 / 9.0 there's support by NVCC for most C++11 / C++14 language features, but make sure to read restrictions for device code
    - <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-cplusplus-language-support>
- .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

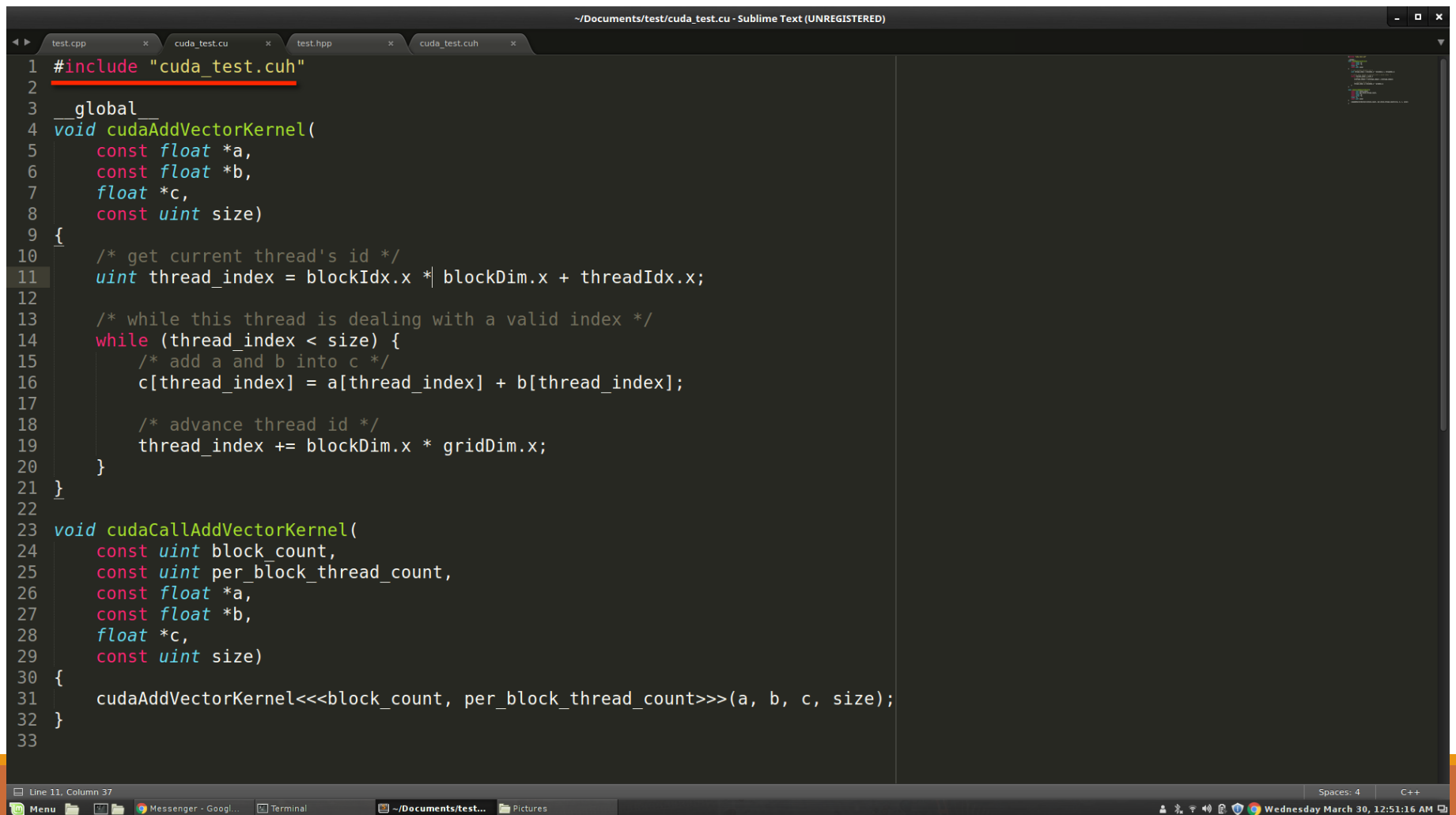


The screenshot shows a Sublime Text editor window titled "~Documents/test/cuda\_test.cuh - Sublime Text (UNREGISTERED)". The editor has four tabs open: test.cpp, cuda\_test.cu, test.hpp, and cuda\_test.cuh. The active tab is cuda\_test.cu, which contains the following code:

```
1 #ifndef CUDA_TEST_HPP
2 #define CUDA_TEST_HPP
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <cuda_runtime.h>
7
8 #ifdef __CUDACC__
9 #define CUDA_CALLABLE __host__ __device__
10 #else
11 #define CUDA_CALLABLE
12 #endif
13
14 void cudaCallAddVectorKernel(
15     const uint block_count,
16     const uint per_block_thread_count,
17     const float *a,
18     const float *b,
19     float *c,
20     const uint size);
21
22 #endif
```

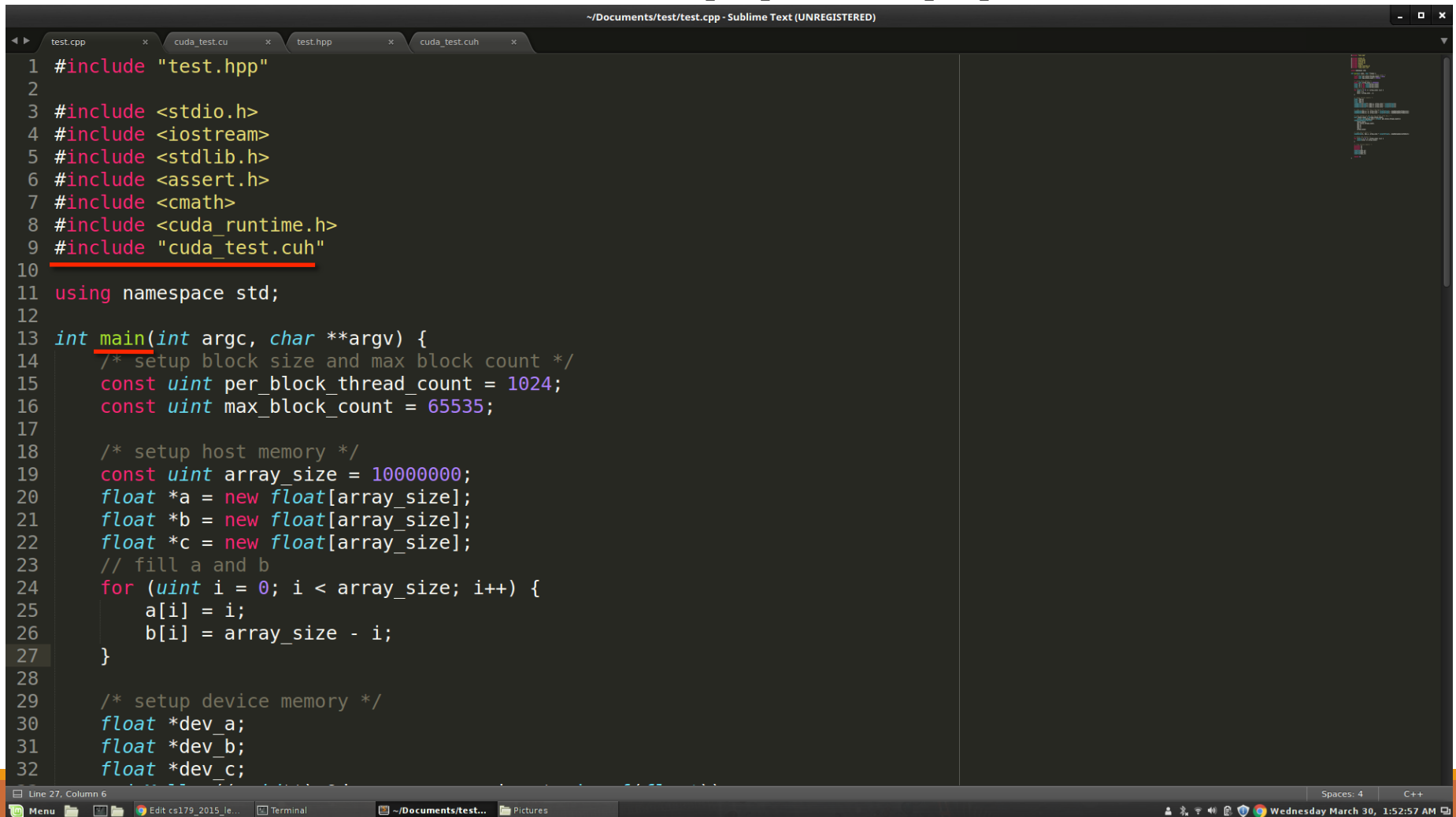
The code is syntax-highlighted in a dark theme, with keywords in blue, constants in orange, and identifiers in green. The line numbers are on the left. The status bar at the bottom shows "Line 19, Column 14", "Spaces: 4", and "C++". The system tray at the bottom right shows the date and time: "Wednesday March 30, 12:51:39 AM".

# .cu/.cuh vs .cpp/.hpp



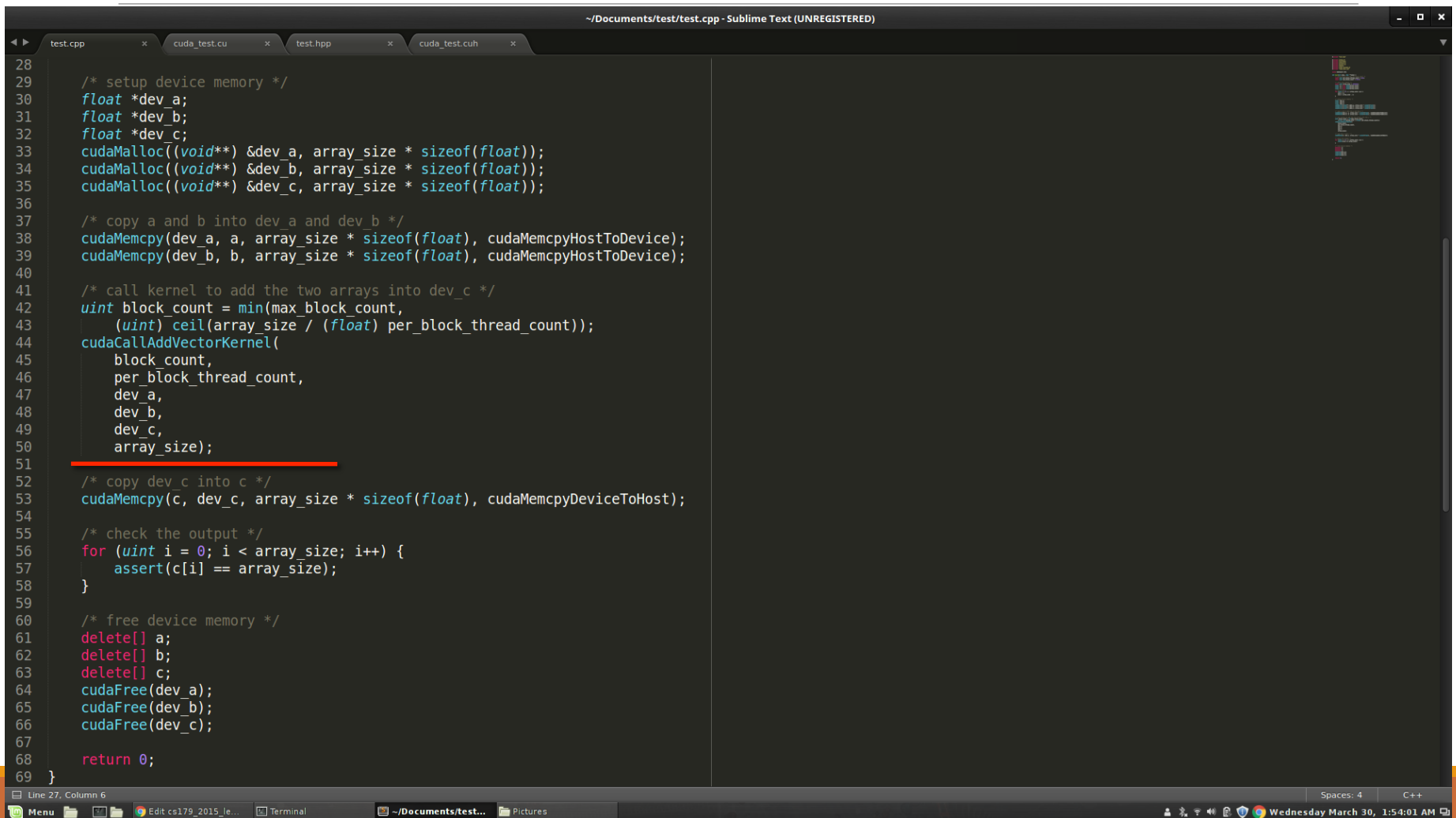
```
1 #include "cuda_test.cuh"
2
3 __global__
4 void cudaAddVectorKernel(
5     const float *a,
6     const float *b,
7     float *c,
8     const uint size)
9 {
10     /* get current thread's id */
11     uint thread_index = blockIdx.x * blockDim.x + threadIdx.x;
12
13     /* while this thread is dealing with a valid index */
14     while (thread_index < size) {
15         /* add a and b into c */
16         c[thread_index] = a[thread_index] + b[thread_index];
17
18         /* advance thread id */
19         thread_index += blockDim.x * gridDim.x;
20     }
21 }
22
23 void cudaCallAddVectorKernel(
24     const uint block_count,
25     const uint per_block_thread_count,
26     const float *a,
27     const float *b,
28     float *c,
29     const uint size)
30 {
31     cudaAddVectorKernel<<<block_count, per_block_thread_count>>>(a, b, c, size);
32 }
33
```

# .cu/.cuh vs .cpp/.hpp



```
1 #include "test.hpp"
2
3 #include <stdio.h>
4 #include <iostream>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <cmath>
8 #include <cuda_runtime.h>
9 #include "cuda_test.cuh"
10
11 using namespace std;
12
13 int main(int argc, char **argv) {
14     /* setup block size and max block count */
15     const uint per_block_thread_count = 1024;
16     const uint max_block_count = 65535;
17
18     /* setup host memory */
19     const uint array_size = 10000000;
20     float *a = new float[array_size];
21     float *b = new float[array_size];
22     float *c = new float[array_size];
23     // fill a and b
24     for (uint i = 0; i < array_size; i++) {
25         a[i] = i;
26         b[i] = array_size - i;
27     }
28
29     /* setup device memory */
30     float *dev_a;
31     float *dev_b;
32     float *dev_c;
```

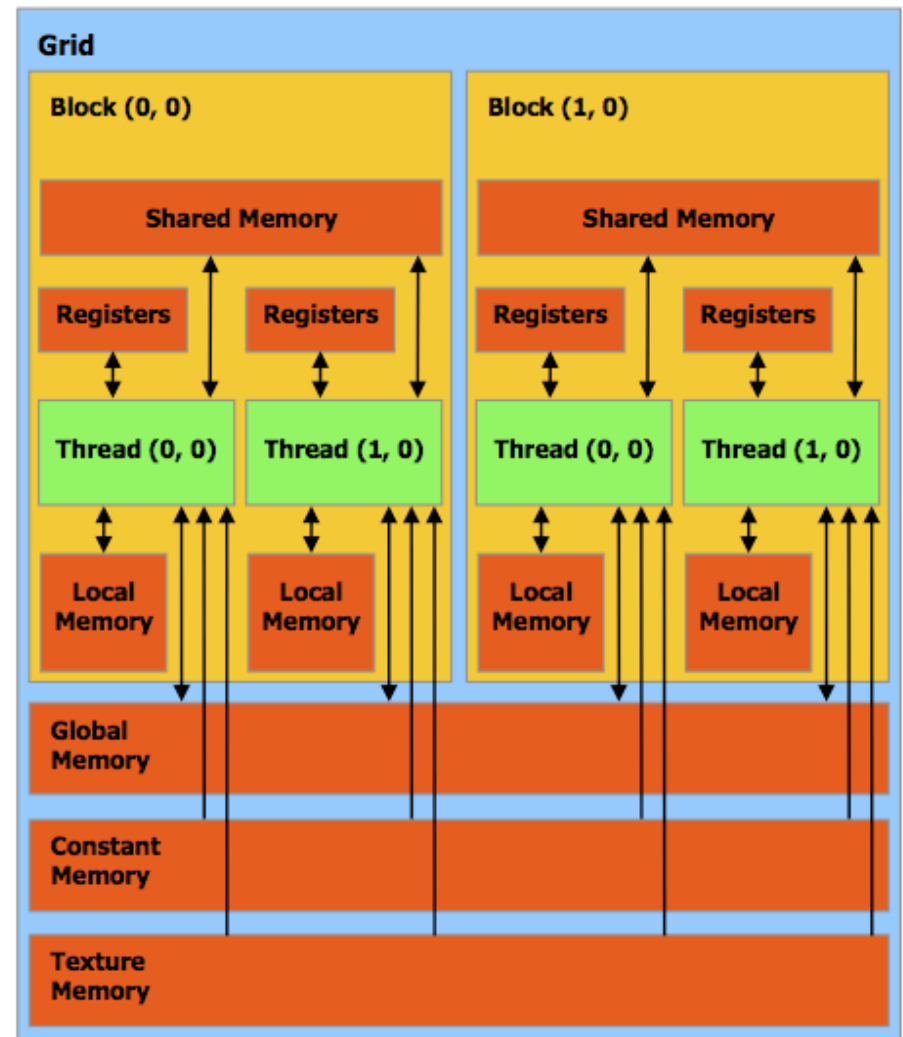
# .cu/.cuh vs .cpp/.hpp



```
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 }
```

# Thread Block Organization

- 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.
- Imagine thread organization as an array of thread indices





# Block and Grid Dimensions

---

- For many parallelizeable problems involving arrays, it's useful to think of multidimensional arrays.
  - E.g. linear algebra, physical modelling, etc, where we want to assign unique thread indices over a multidimensional object
  - So, CUDA provides built in multidimensional thread indexing capabilities with a struct called `dim3`!

# Dim3

---

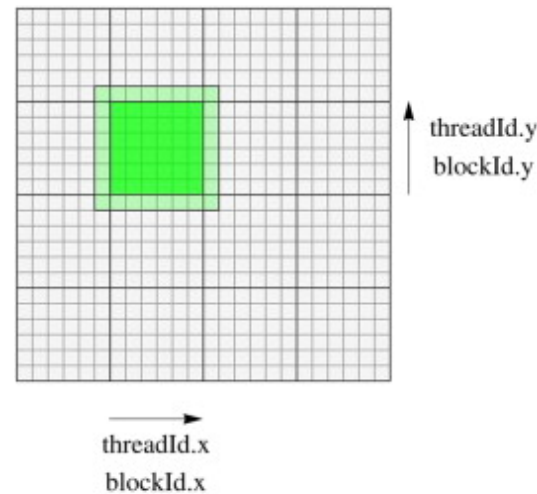
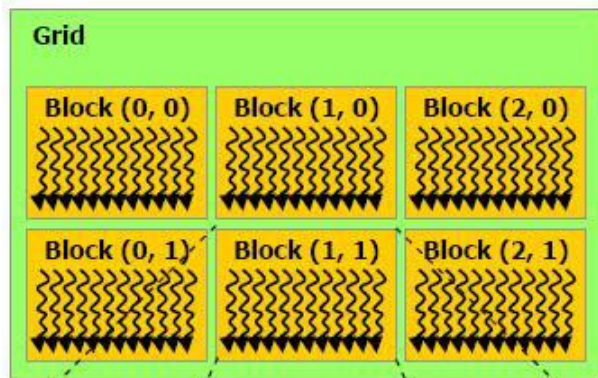
dim3 is a struct (defined in vector\_types.h) to define your Grid and Block dimensions.

```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#ifdef __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 */
};
```

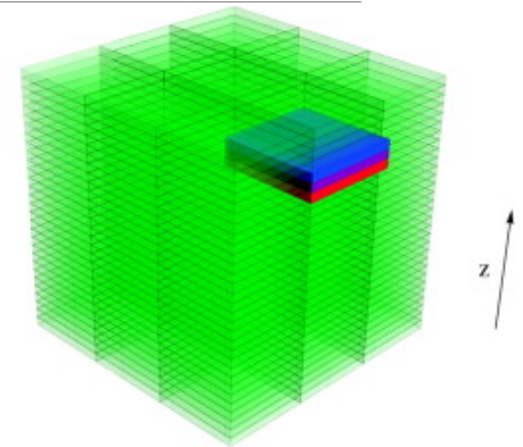
Works for dimensions 1, 2, and 3:

- 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>>>(...);

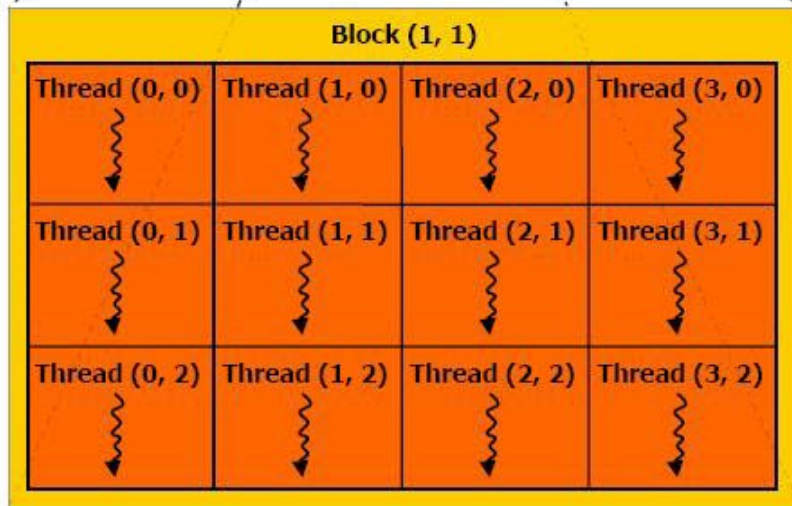
# Grid/Block/Thread Visualized



(a)



(b)



# 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/>

# SIMT (Single Instruction, Multiple Thread) Architecture

---

A looser extension of SIMD which is what CUDA's computational model uses

- Key differences:
  - Single instruction, multiple register sets
    - Wastes some registers, but mostly necessary for following two points
  - Single instruction, multiple addresses (i.e. parallel memory access!)
    - Memory access conflicts! Will discuss next week.
  - Single instruction, multiple flow paths (i.e. if statements are allowed!!!)
    - Introduces slowdowns, called 'warp-divergence.'

Good description of differences

- <https://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html>
- <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation>



# Thread blocks and Warps – Questions?



Copyright Games Workshop inc.

# Important CUDA Hardware Keywords!

---

- Streaming Multiprocessor (SM) – Each contains (usually) 128 single precision CUDA cores (which execute a thread) and their associated cache.
  - This is a standard based on your machines Compute Capability.
- Warp – A unit of up to 32 threads (all within the same block)
  - Each SM creates and manages multiple warps via the block abstraction. Assigns to each warp a Warp Scheduler to schedule the execution of instructions in each warp.
- Warp Divergence – A condition where threads within a warp need to execute different instructions in order to continue executing their kernel.
  - In order to maintain multiple flow path per instruction, threads in different ‘execution branches’ during an instruction are given no-ops.
  - 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.
  - Independent Thread Scheduling fixes this problem by maintaining an execution state per thread. See Compute Capability 7.x

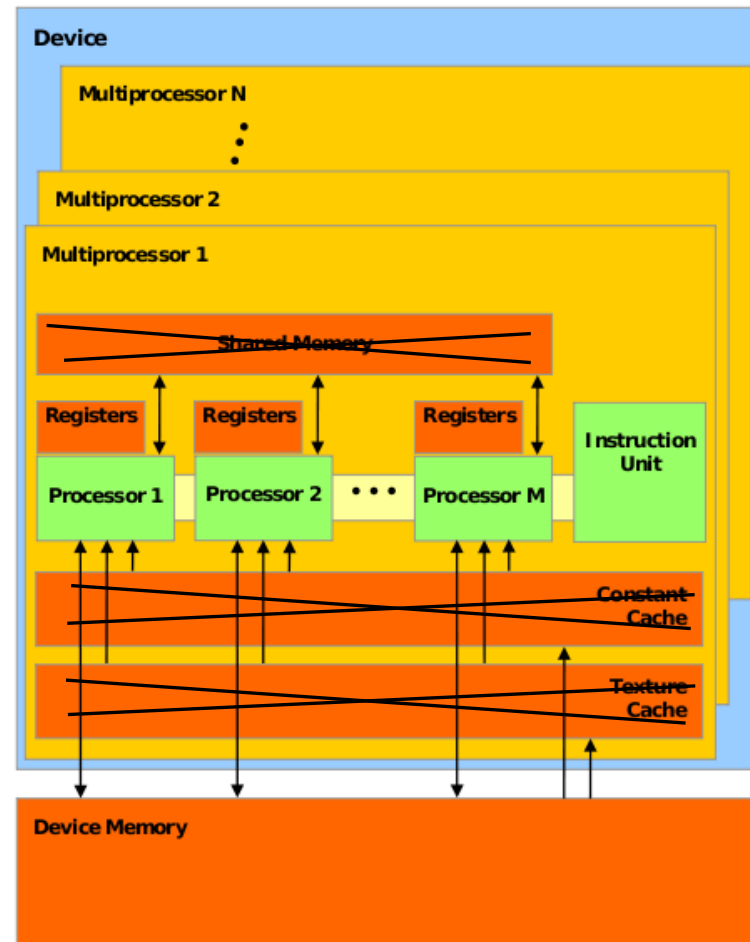


# What a modern GPU looks like



# Inside a GPU

The black Xs are just crossing out things you don't have to think about just yet. We'll cover memory next week



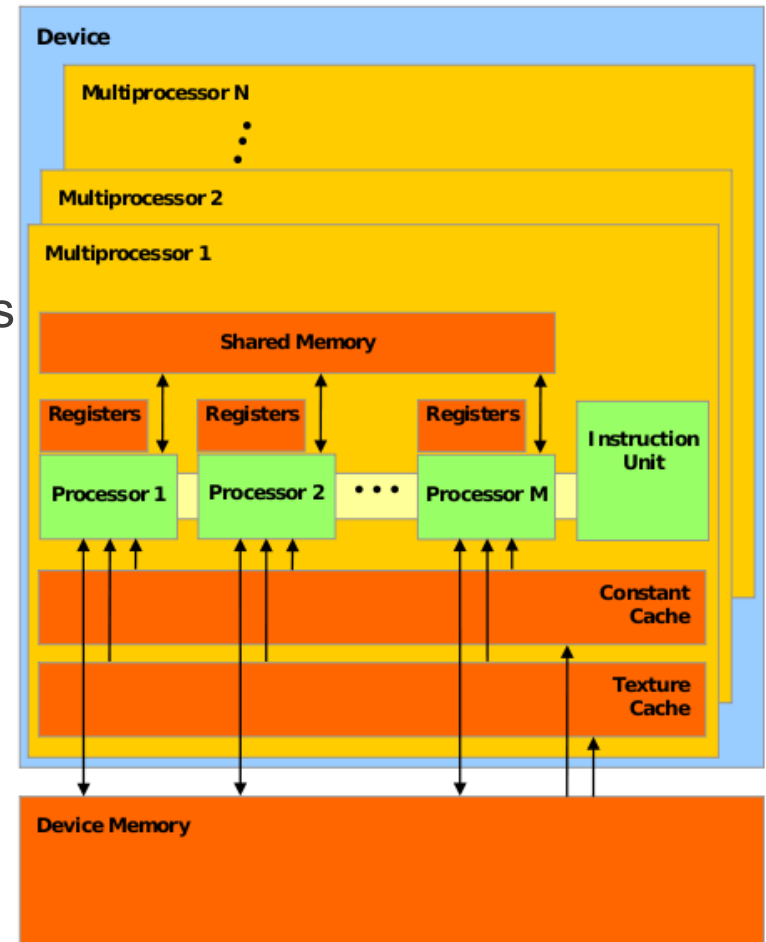
# Inside a GPU

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 have other options
- Will come back to this in future lectures

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

- Each SM has multiple processors but only one instruction unit (each thread shares program counter)
- Groups of processors must run the exact same set of instructions at any given time within a single SM



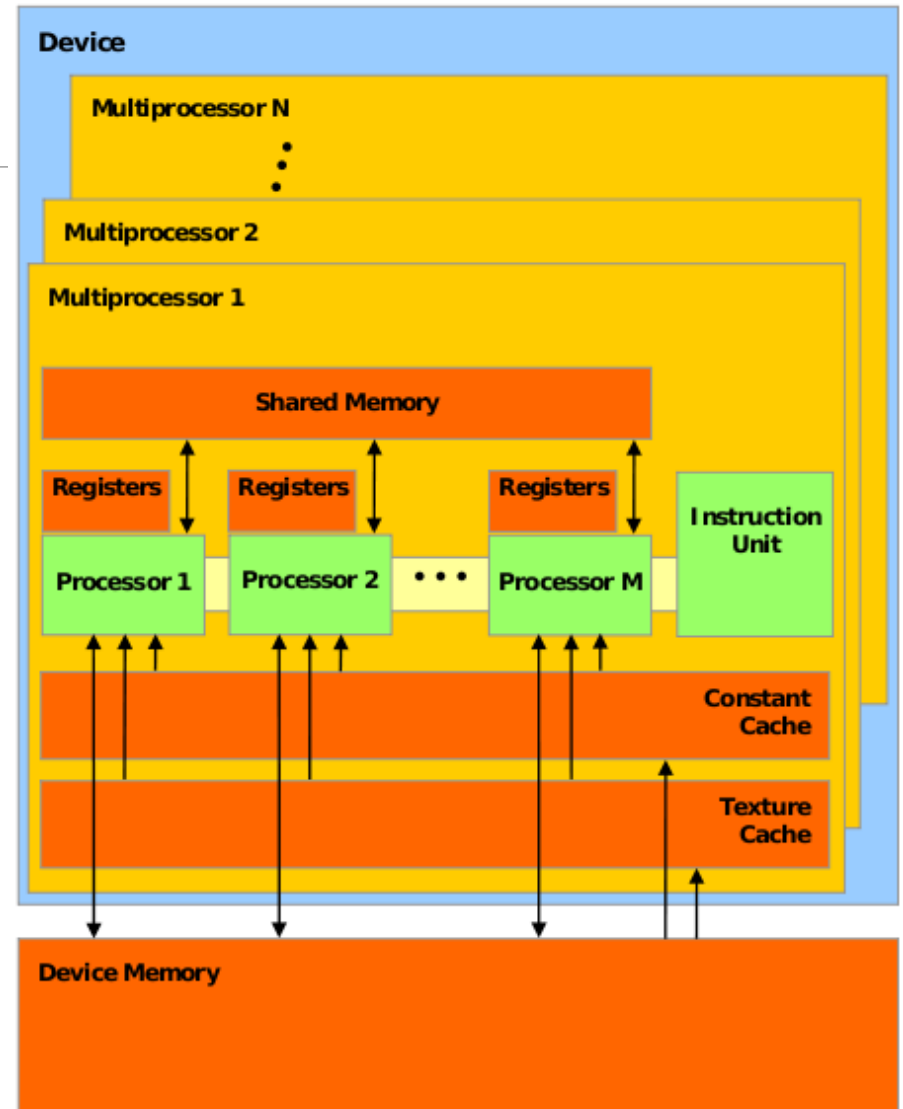
# Inside a GPU

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>>>(...);`

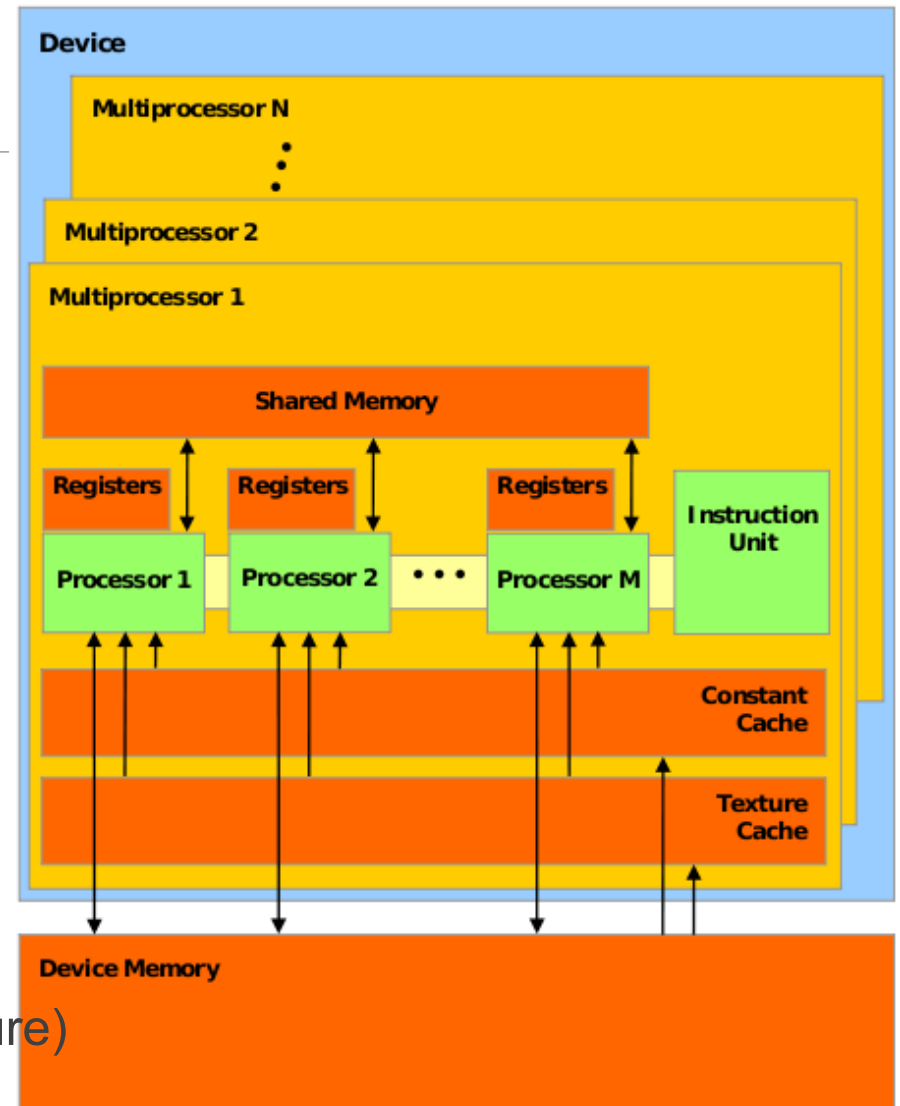


# Inside a GPU

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)

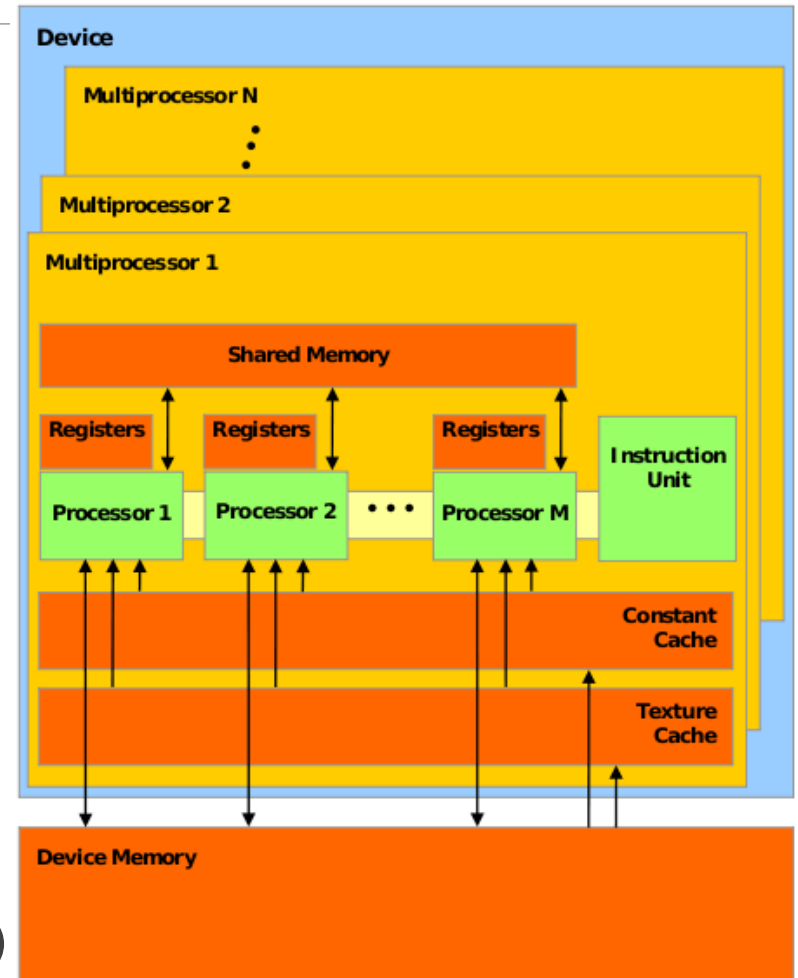


# Inside a GPU

Each block is assigned to an SM

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

- Warps consist of 32 threads
  - CUDA defined constant in `cuda_runtime.h`
- 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)

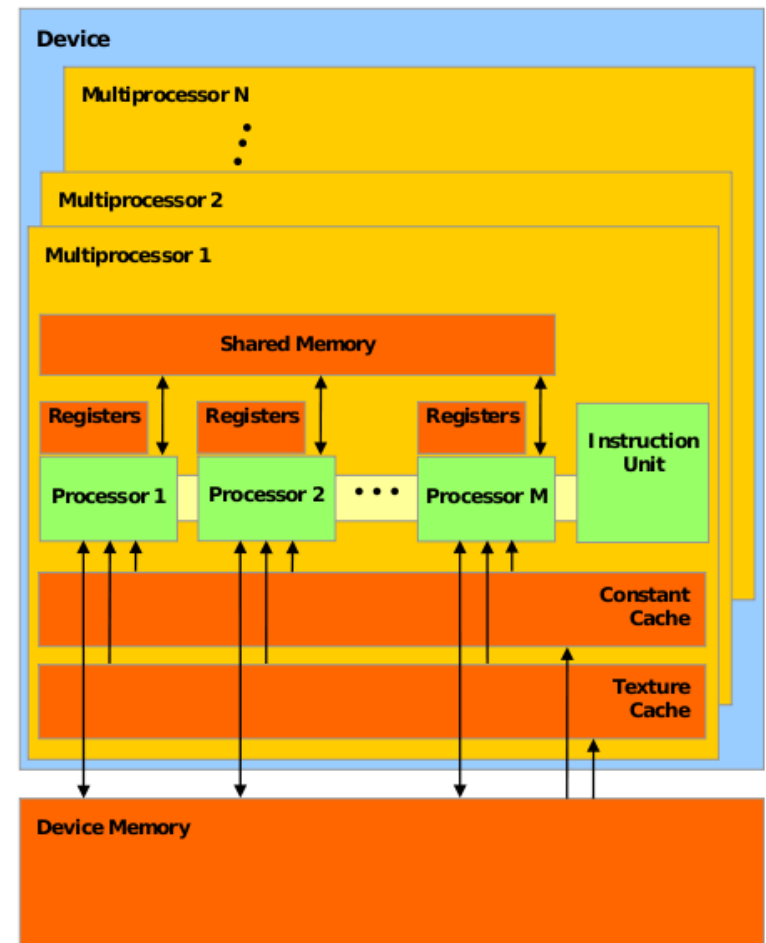




# Inside a GPU (fun hardware info)

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.
- Turing (2018) maintains 128 core standard



# 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](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](http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf)
  - “nvidia kepler whitepaper”





# $A[] + B[] \rightarrow C[]$ (again)

```
test.cpp x cuda_test.cu x test.hpp x cuda_test.cuh x
~/Documents/test/test.cpp - Sublime Text (UNREGISTERED)

1 #include "test.hpp"
2
3 #include <stdio.h>
4 #include <iostream>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <cmath>
8 #include <cuda_runtime.h>
9 #include "cuda_test.cuh"
10
11 using namespace std;
12
13 int main(int argc, char **argv) {
14     /* setup block size and max block count */
15     const uint per_block_thread_count = 1024;
16     const uint max_block_count = 65535;
17
18     /* setup host memory */
19     const uint array_size = 10000000;
20     float *a = new float[array_size];
21     float *b = new float[array_size];
22     float *c = new float[array_size];
23     // fill a and b
24     for (uint i = 0; i < array_size; i++) {
25         a[i] = i;
26         b[i] = array_size - i;
27     }
28
29     /* setup device memory */
30     float *dev_a;
31     float *dev_b;
32     float *dev_c;
```

Line 27, Column 6

Menu Edit cs179\_2015\_le... Terminal ~/Documents/test... Pictures

Spaces: 4 C++

Wednesday March 30, 1:52:57 AM

# $A[] + B[] \rightarrow C[]$ (again)

```
~/Documents/test/test.cpp - Sublime Text (UNREGISTERED)
test.cpp x cuda_test.cu x test.hpp x cuda_test.cuh x
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  
Menu Edit cs179\_2015\_le... Terminal ~/Documents/test... Pictures  
Spaces: 4 C++  
Wednesday March 30, 1:54:01 AM

# $A[] + B[] \rightarrow C[]$ (new!!)

```
~/Documents/test/cuda_test.cu - Sublime Text (UNREGISTERED)
test.cpp x cuda_test.cu x test.hpp x cuda_test.cuh x
1 #include "cuda_test.cuh"
2
3 __global__
4 void cudaAddVectorKernel(
5     const float *a,
6     const float *b,
7     float *c,
8     const uint size)
9 {
10     /* get current thread's id */
11     uint thread_index = blockIdx.x * blockDim.x + threadIdx.x;
12
13     /* while this thread is dealing with a valid index */
14     while (thread_index < size) {
15         /* add a and b into c */
16         c[thread_index] = a[thread_index] + b[thread_index];
17
18         /* advance thread id */
19         thread_index += blockDim.x * gridDim.x;
20     }
21 }
22
23 void cudaCallAddVectorKernel(
24     const uint block_count,
25     const uint per_block_thread_count,
26     const float *a,
27     const float *b,
28     float *c,
29     const uint size)
30 {
31     cudaAddVectorKernel<<<block_count, per_block_thread_count>>>(a, b, c, size);
32 }
33
```

Line 11, Column 37

Menu Messenger - Google... Terminal ~/Documents/test... Pictures

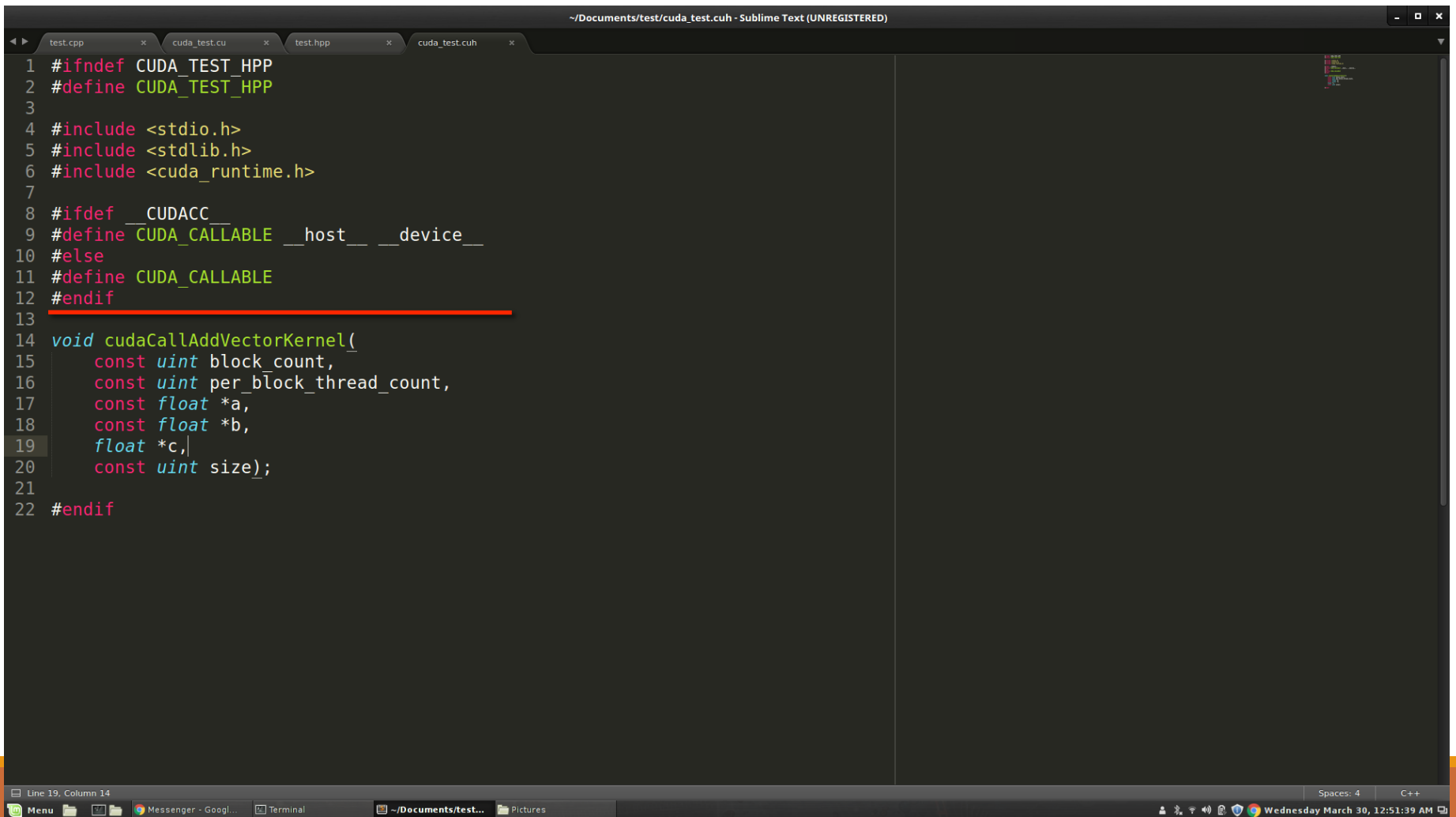
Spaces: 4 C++ Wednesday March 30, 12:51:16 AM

# Questions for hardware?

---



# Stuff that will be useful later

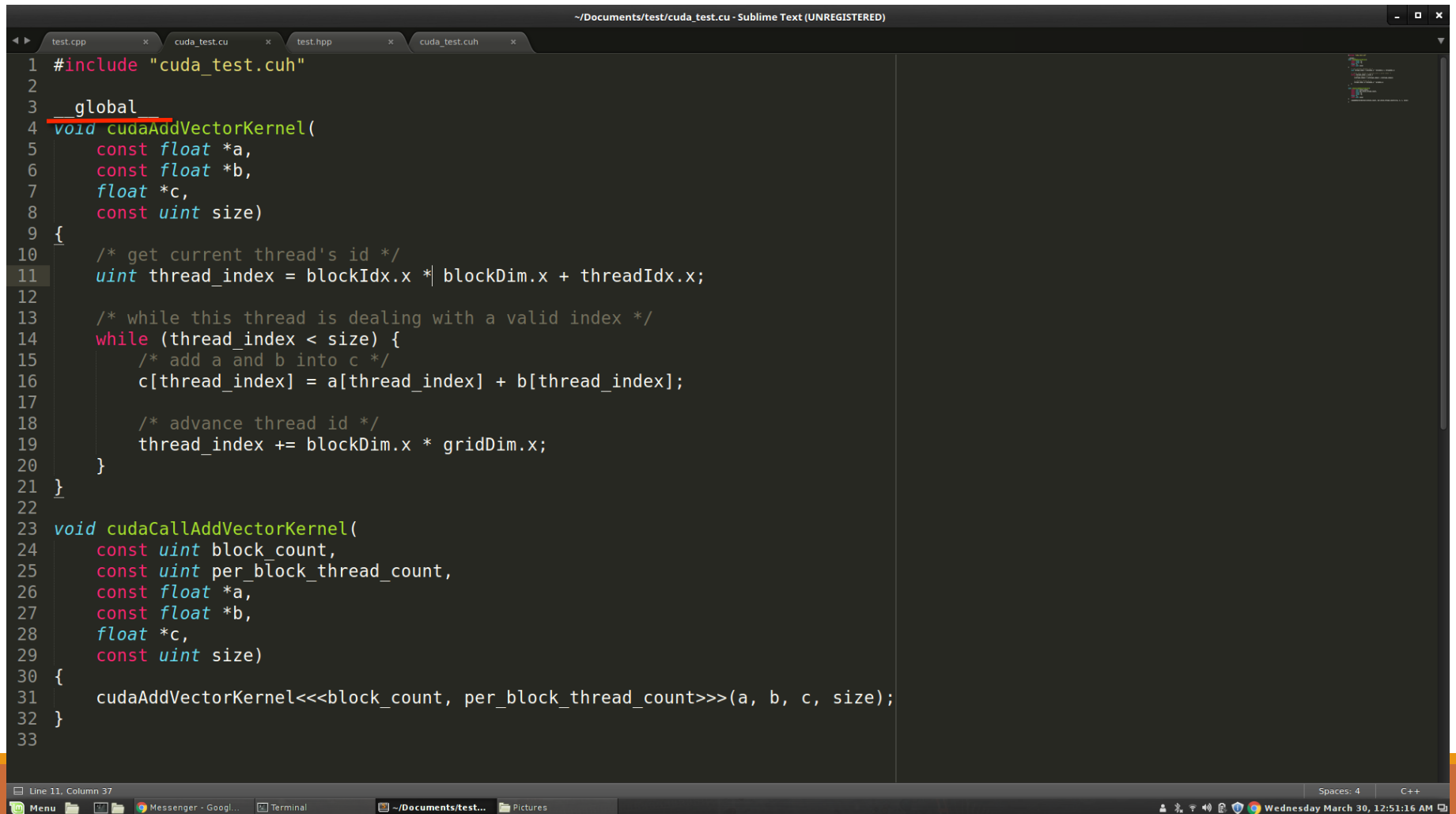


```
1 #ifndef CUDA_TEST_HPP
2 #define CUDA_TEST_HPP
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <cuda_runtime.h>
7
8 #ifdef __CUDACC__
9 #define CUDA_CALLABLE __host__ __device__
10 #else
11 #define CUDA_CALLABLE
12 #endif
13
14 void cudaCallAddVectorKernel(
15     const uint block_count,
16     const uint per_block_thread_count,
17     const float *a,
18     const float *b,
19     float *c,
20     const uint size);
21
22 #endif
```

# Stuff that will be useful later

```
~/Documents/repos/cs81/src/swarm.cuh - Sublime Text (UNREGISTERED)
test.cpp x cuda_test.cu x swarm.cuh x test.hpp x cuda_test.cuh x
1 #ifndef SWARM_CUH
2 #define SWARM_CUH
3
4 #include <stdio>
5 #include <cuda_runtime.h>
6
7 #include "constants.hpp"
8 #include "utils.hpp"
9
10 struct GPUVec3d {
11     double x;
12     double y;
13     double z;
14
15     CUDA_CALLABLE GPUVec3d();
16     CUDA_CALLABLE GPUVec3d(double x, double y, double z);
17     CUDA_CALLABLE GPUVec3d(const GPUVec3d &v);
18     CUDA_CALLABLE ~GPUVec3d();
19
20     CUDA_CALLABLE double sqNorm();
21
22     CUDA_CALLABLE const GPUVec3d &operator=(const GPUVec3d & rhs);
23     CUDA_CALLABLE bool operator==(const GPUVec3d & rhs) const;
24     CUDA_CALLABLE bool operator!=(const GPUVec3d & rhs) const;
25     CUDA_CALLABLE const GPUVec3d &operator+=(const GPUVec3d & rhs);
26     CUDA_CALLABLE const GPUVec3d &operator-=(const GPUVec3d & rhs);
27     CUDA_CALLABLE const GPUVec3d &operator*=(const double d);
28     CUDA_CALLABLE const GPUVec3d &operator/=(const double d);
29     CUDA_CALLABLE friend const GPUVec3d operator+(const GPUVec3d &lhs,
30     const GPUVec3d &rhs);
31     CUDA_CALLABLE friend const GPUVec3d operator-(const GPUVec3d &lhs,
32     const GPUVec3d &rhs);
33     CUDA_CALLABLE friend const GPUVec3d operator*(const GPUVec3d &lhs,
34     const double d);
35     CUDA_CALLABLE friend const GPUVec3d operator*(const double d,
36     const GPUVec3d &rhs);
37     CUDA_CALLABLE friend const GPUVec3d operator/(const GPUVec3d &lhs,
38     const double d);
39 };
40
41 class SwarmParticle;
42
```

# Stuff that will be useful later



```
1 #include "cuda_test.cuh"
2
3 global
4 void cudaAddVectorKernel(
5     const float *a,
6     const float *b,
7     float *c,
8     const uint size)
9 {
10     /* get current thread's id */
11     uint thread_index = blockIdx.x * blockDim.x + threadIdx.x;
12
13     /* while this thread is dealing with a valid index */
14     while (thread_index < size) {
15         /* add a and b into c */
16         c[thread_index] = a[thread_index] + b[thread_index];
17
18         /* advance thread id */
19         thread_index += blockDim.x * gridDim.x;
20     }
21 }
22
23 void cudaCallAddVectorKernel(
24     const uint block_count,
25     const uint per_block_thread_count,
26     const float *a,
27     const float *b,
28     float *c,
29     const uint size)
30 {
31     cudaAddVectorKernel<<<block_count, per_block_thread_count>>>(a, b, c, size);
32 }
33
```

The screenshot shows a Sublime Text editor window titled '~/.Documents/test/cuda\_test.cu - Sublime Text (UNREGISTERED)'. The editor has four tabs: test.cpp, cuda\_test.cu, test.hpp, and cuda\_test.cuh. The active tab is cuda\_test.cu, which contains the code shown in the pre-block. The code is a C++ file for a CUDA kernel. It includes a header file 'cuda\_test.cuh'. The kernel function 'cudaAddVectorKernel' takes four arguments: two constant float pointers 'a' and 'b', a float pointer 'c', and a constant unsigned integer 'size'. The function body calculates the thread index and iterates over the array, adding elements from 'a' and 'b' to 'c'. The host function 'cudaCallAddVectorKernel' takes the same arguments plus 'block\_count' and 'per\_block\_thread\_count', and calls the kernel with a launch configuration of three blocks. The status bar at the bottom shows 'Line 11, Column 37', 'Spaces: 4', 'C++', and the system clock 'Wednesday March 30, 12:51:16 AM'.

# 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