## 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

🔟 Menu 🔚

S Messenger - Googl...

~/Documents/test...

E Pictures

<pre>ty trace to the state of t</pre>		~/Documents/test/cuda_test.cuh - Sublime Text (UNREGISTERED)	- <b>D</b> ×
<pre># define cond_tcit_irr # define cond_tcit_irr # define cond_tcit_irr # define cond_tcit_irr # define cond_tcit_able # const uint block count, const float *a, const float *b, float *c, const uint size); # define cond_tcit_able # define cond_tcit_able</pre>	▲ ► test.cpp × cuda_test.cu × test.hpp ×	cuda_test.cuh x	
<pre>4 #include <stdio.h> 5 #include <stdio.h> 6 #include <stdib.h> 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7</stdib.h></stdio.h></stdio.h></pre>	<pre>2 #define CUDA_TEST_HPP</pre>		
<pre>9 #define CUDA_CALLABLEhostdevice</pre>	4 #include <stdio.h> 5 #include <stdlib.h></stdlib.h></stdio.h>		
<pre>14 void cudaCallAddVectorKernel( 15</pre>	9 #define CUDA_CALLABLEhost 10 #else 11 #define CUDA_CALLABLE 12 #endif	_device	
	<pre>14 void cudaCallAddVectorKernel( 15</pre>	count,	
□ Line 19. Column 14			
☐ Line 19, Column 14 C++			
	□ Line 19, Column 14		Spaces: 4 C++

🔺 🐁 후 🐠 🖹 💮 🧔 Wednesday March 30, 12:51:39 AM 😐

<u>.cu/.cuh</u> vs .cpp/.hpp

	_	~/Documents/test/cuda_test.cu - Sublime Text (UNREGISTERED)	_ <b>_</b> X
< Þ /	test.cpp	× cuda_test.cu × test.hpp × cuda_test.cuh ×	τ.
1	#in	clude "cuda_test.cuh"	
2	a.	lobal	
		d cudaAddVectorKernel(	
5	VUI	const float *a,	
6		const float *b,	
7		float *c,	
8		const uint size)	
9	<u>{</u>		
10	-	/* get current thread's id */	
11		<pre>uint thread index = blockIdx.x * blockDim.x + threadIdx.x;</pre>	
12			
13		/* while this thread is dealing with a valid index */	
14		while (thread index < size) {	
15		/* add a and b into c */	
16		<pre>c[thread_index] = a[thread_index] + b[thread_index];</pre>	
17			
18		/* advance thread id */	L. C.
19		thread_index += blockDim.x * gridDim.x;	
20		}	
21	<u>}</u>		
22			
23	VOI	d cudaCallAddVectorKernel(	
24		<pre>const uint block_count,</pre>	
25		<pre>const uint per_block_thread_count,</pre>	
26		const float *a,	
27 28		<pre>const float *b, float *c</pre>	
28		float *c, const uint size)	
30	5		
31	1	<pre>cudaAddVectorKernel&lt;&lt;<block block="" count="" count,="" per="" thread="">&gt;&gt;(a, b, c, size);</block></pre>	
32	1	-cuduxuveetorkernet< count, per_btock_thread_count>>>(a, b, c, size),	
33	1		
- 55			

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

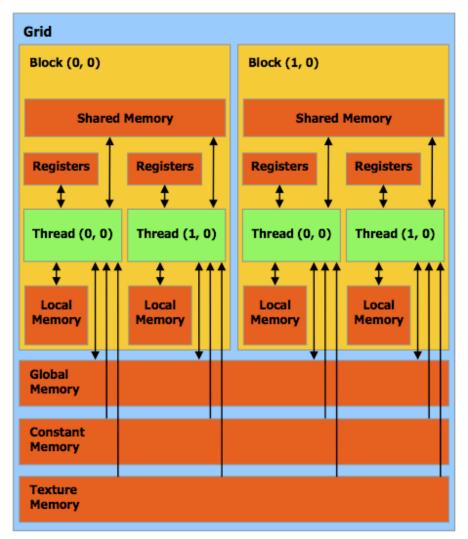
	~/Documents/test.test.cpp - Sublime Text (UNREGISTERED)	- • ×
▲ ▶ test.cpp × cuda_test.cu	x test.hpp x cuda_test.cuh x	
1 #include "test.hp	p"	
2		
3 #include <stdio.h< td=""><td>&gt;</td><td></td></stdio.h<>	>	
4 #include <iostrea< td=""><td>m&gt;</td><td>the second secon</td></iostrea<>	m>	the second secon
5 #include <stdlib.< td=""><td>h&gt;</td><td></td></stdlib.<>	h>	
6 #include <assert.< td=""><td>h&gt;</td><td></td></assert.<>	h>	
7 <pre>#include <cmath></cmath></pre>		
8 #include <cuda_ru< td=""><td>ntime.h&gt;</td><td></td></cuda_ru<>	ntime.h>	
9 <b>#include</b> "cuda_te	st.cuh"	
10		
11 using namespace s	td;	, i i i i i i i i i i i i i i i i i i i
12		
13 int main(int argo		
	k size and max block count */	
	r_block_thread_count = 1024;	
	x_block_count = 65535;	
17		
18 /* setup host		
	ray_size = 10000000;	
	w float[array_size];	
	w float[array_size];	
	w float[array_size];	
23 // fill a and		
	0; i < array_size; i++) {	
25 a[i] = i;		
	ray_size - i;	
27 }		
28		
29 /* setup devi		
30 <i>float</i> *dev_a;		
<pre>31 float *dev_b;</pre>		
32    float *dev_c;		
☐ Line 27, Column 6	I Terminal III ~/Documents/test III Pictures	Spaces: 4 C++ ▲ ≵. ♥ 40 🗭 🏠 Wednesday March 30, 1:52:57 AM 🗗

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

	~/Documents/test/test.cp	p - Sublime Text (UNREGISTERED)	- • ×
< Þ /	test.cpp x cuda_test.cu x test.hpp x cuda_test.cuh x		•
28		Training Training Training Training Training Training Training Training	
29			-
30	float *dev_a;		
31	float *dev_b;	1000-100 2020-00 2020-00	ine maran
32	float *dev_c;		
33	cudaMalloc(( <i>void</i> **) &dev_a, array_size * sizeof( <i>float</i> ));		
34	cudaMalloc(( <i>void</i> **) &dev_b, array_size * sizeof( <i>float</i> ));		
35	cudaMalloc(( <i>void</i> **) &dev_c, array_size * sizeof( <i>float</i> ));		
36			
37	/* copy a and b into dev_a and dev_b */		
38	<pre>cudaMemcpy(dev_a, a, array_size * sizeof(float), cudaMemcpyHostToDevice); cudaMemcpy(dev_b, a, array_size * sizeof(float), cudaMemcpyHostToDevice);</pre>		Î
39	<pre>cudaMemcpy(dev_b, b, array_size * sizeof(float), cudaMemcpyHostToDevice);</pre>		
40 41			
41 42	<pre>/* call kernel to add the two arrays into dev_c */ uint block count = min(max block count,</pre>		
42	(uint) ceil(array size / (float) per block thread count));		
43	cudaCallAddVectorKernel(		
45	block count,		
46	per block thread count,		
47	dev a,		
48	dev b,		
49	dev c,		
50	array size);		
51			
52			
53	<pre>cudaMemcpy(c, dev_c, array_size * sizeof(float), cudaMemcpyDeviceToHost);</pre>		I
54			
55	/* check the output */		
56 57	<pre>for (uint i = 0; i &lt; array_size; i++) {     const([]]     const([]] </pre>		
58	<pre>assert(c[i] == array_size);</pre>		
59	ſ		
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			
	27. Column 6 u 🛅 🔄 🚰 👩 Edit cs179_2015_le 🔣 Terminal 🔤 ~/Documents/test 🔚 Pictures	Spaces: 4 1 🦹 🦹 🕆 🕫 🔞 🔞 Wednesday March 3	
Men Men		🚆 🐐 र 🕸 🔄 🖤 🥥 Wednesday March 🤅	30, 1:54:01 AM 밀

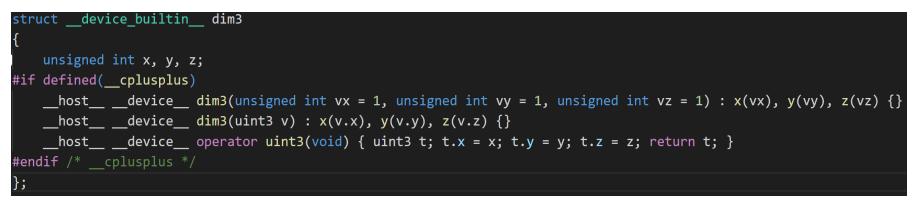
### Thread Block Organization Keywords

- 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



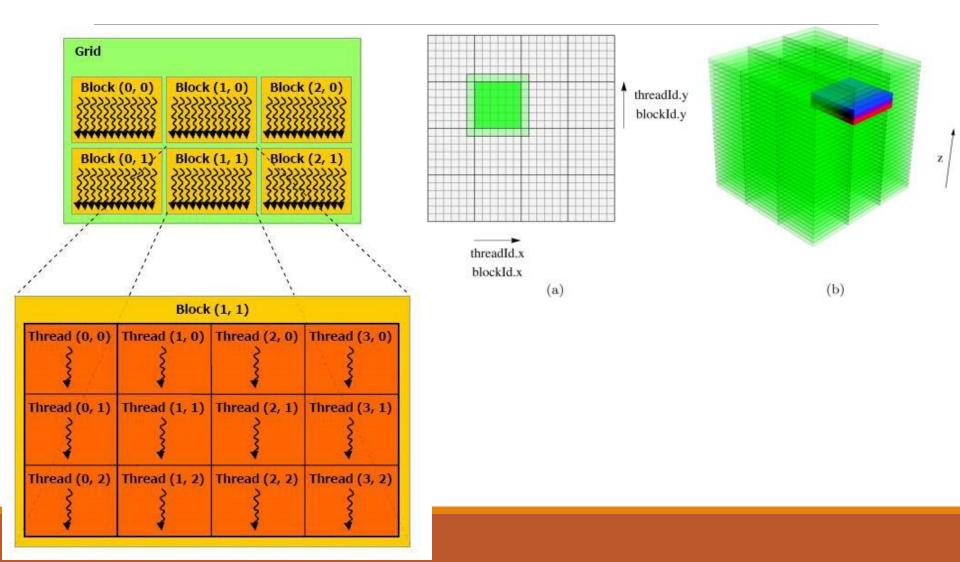
### Block and Grid Dimensions

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



- 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.

<u>https://polly.llvm.org/</u>

### Thread blocks and Warps visualized

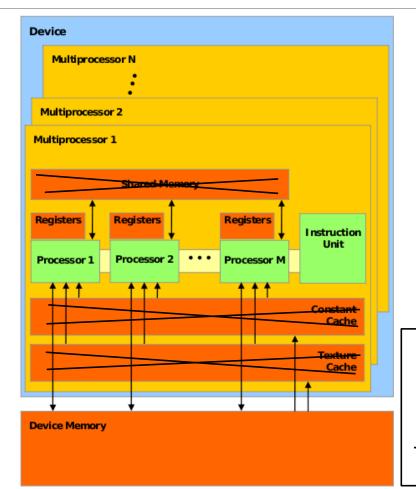


# Keywords you MUST know to code WELL in CUDA

- 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





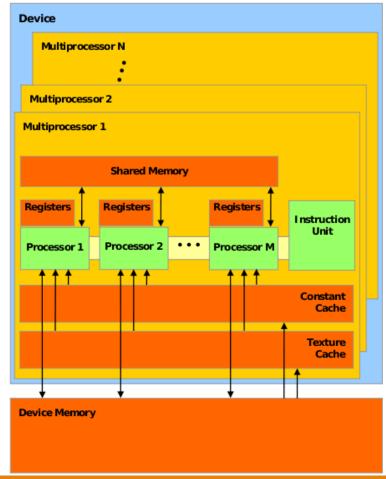
The black Xs are just crossing out things you don't have to think about just yet. You'll learn about them later

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



- 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

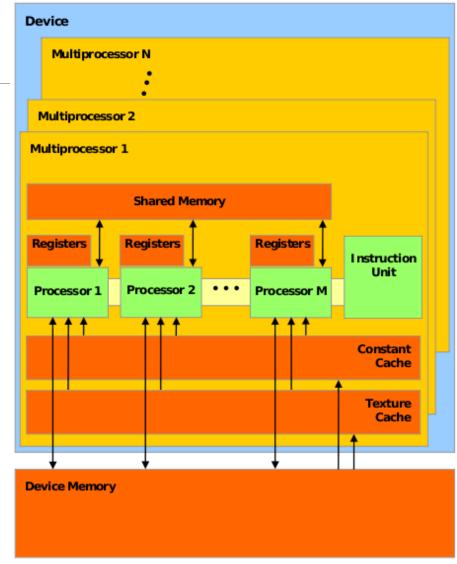
• e.g.

dim3 dimBlock(8, 8, 8);

dim3 dimGrid(100, 100, 1);

Kernel<<<dimGrid, dimBlock>>>(...);

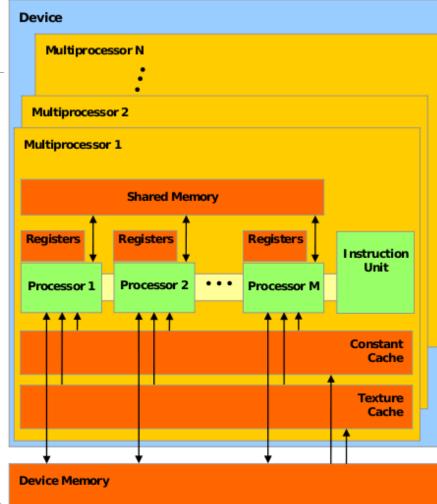
- However, we'll often only work with 1 dimensional grids and blocks
- e.g. Kernel<<<block\_count, block\_size>>>(...);



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

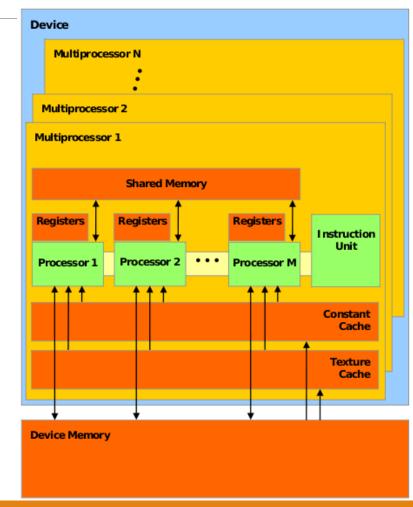
- If you go over either of these numbers your GPU will just give up or output garbage data
- Much of GPU programming is dealing with this kind of hardware limitations! Get used to it
- This limitation also means that your Kernel must compensate for the fact that you may not have enough threads to individually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



Each block is assigned to an SM

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

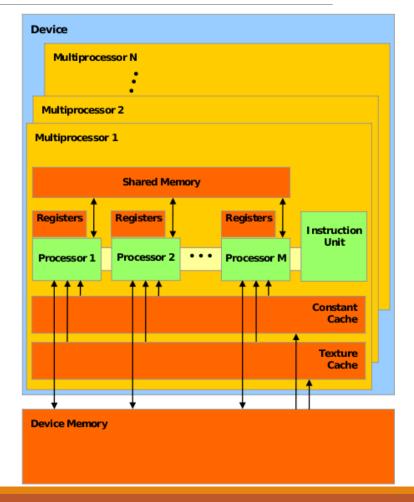
- Warps consist of 32 threads
- All 32 threads MUST run the exact same set of instructions at the same time
  - Due to the fact that there is only one instruction unit
- Warps are run concurrently in an SM
- If your Kernel tries to have threads do different things in a single warp (using if statements 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)
  - <u>https://international.download.nvidia.com/geforce-</u> <u>com/international/pdfs/GeForce\_GTX\_1080\_Whitepaper\_FINA</u> <u>L.pdf</u>
  - <u>http://www.nvidia.com/content/PDF/product-</u> <u>specifications/GeForce\_GTX\_680\_Whitepaper\_FINAL.pdf</u>
  - "nvidia kepler whitepaper"

SM												
					Instruct	io:	n Cache					
	h		on Buffe	r				l		on Buffe	r	
	spatch Uni	Warp So		lispatch Ur	-16		D	ispatch Un		heduler	lispatch Ur	.14
	Regist	er File (1	16,384 x	32-bit)				Regist	er File (1	16,384 x	32-bit)	
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
	Texture / L1 Cache											
	Tex			Tex				Tex			Tex	
	I	nstructi	on Buffe	er.				I	nstructi	on Buffe	er	
			cheduler							cheduler		
	ispatch Un	it		Dispatch U	nit		D	ispatch Un	it		Dispatch U	nit
	Regist	er File (	16,384 x	32-bit)				Regist	er File (	16,384 x	32-bit)	
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU		Core	Core	Core	Core	LD/ST	SFU
	Texture / L1 Cache											
	Tex			Tex				Tex			Tex	
	96KB Shared Memory											

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

	~/Documents/test/test.cpp - Sublime Text (UNREGISTERED)	- <b>D</b> X
★ test.cpp x cuda_test.cu x test.hpp x cuda_test.cuh x		
1 #include "test.hpp"		
2		
3 #include <stdio.h></stdio.h>		
4 <pre>#include <iostream></iostream></pre>		Patrick Concernent
5 #include <stdlib.h></stdlib.h>		
6 #include <assert.h></assert.h>		
7 <pre>#include <cmath></cmath></pre>		
8		
9 #include "cuda_test.cuh"		
10		
<pre>11 using namespace std;</pre>		
12		
<pre>13 int main(int argc, char **argv) {</pre>		
14 /* setup block size and max block count */		
<pre>15 const uint per_block_thread_count = 1024;</pre>		
16		
17		
<pre>18 /* setup host memory */</pre>		
19 const uint array_size = 10000000;		
20 float *a = new float[array_size];		
<pre>21 float *b = new float[array_size]; 22 float *c = new float[array_size];</pre>		
<pre>22 float *c = new float[array_size]; 22 ((fill a and b</pre>		
23 // fill a and b		
24		
25 a[i] = i; 26 b[i] = array size - i;		
27 } 28		
20 29 /* setup device memory */		
30 float *dev a;		
31 float *dev b;		
32 float *dev c;		
		Spaces: 4 C++
🔲 Line 27, Column 6		Spaces: 4 C++

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

<pre>* mature * code_metcur * texture * code_metcure * code_metcur</pre>
<pre>33 cudaMalloc((void**) &amp; dev a, array_size * sizeof(float)); 34 cudaMalloc((void**) &amp; dev_b, array_size * sizeof(float)); 35 cudaMemcpy(dev_a, a, array_size * sizeof(float)); 36 37 /* copy a and b into dev a and dev b */ 38 cudaMemcpy(dev_b, b, 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 44 (uint) ceil(array size / (float) per block_thread_count)); 44 cudacattAddVectorKernet( 45 block_count, 46 per_block_thread_count, 47 dev_a, 48 dev_b, 48 dev_b, 49 dev_c, 50 array_size); 51 47 /* copy dev_c into c */</pre>
<pre>33 cudaMalloc((void**) &amp; dev a, array_size * sizeof(float)); 34 cudaMalloc((void**) &amp; dev_b, array_size * sizeof(float)); 35 cudaMemcpy(dev_a, a, array_size * sizeof(float)); 36 37 /* copy a and b into dev a and dev b */ 38 cudaMemcpy(dev_b, b, 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 44 (uint) ceil(array size / (float) per block_thread_count)); 44 cudacattAddVectorKernet( 45 block_count, 46 per_block_thread_count, 47 dev_a, 48 dev_b, 48 dev_b, 49 dev_c, 50 array_size); 51 47 /* copy dev_c into c */</pre>
<pre>33 cudaMalloc((void**) &amp; dev a, array_size * sizeof(float)); 34 cudaMalloc((void**) &amp; dev_b, array_size * sizeof(float)); 35 cudaMemcpy(dev_a, a, array_size * sizeof(float)); 36 37 /* copy a and b into dev a and dev b */ 38 cudaMemcpy(dev_b, b, 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 44 (uint) ceil(array size / (float) per block_thread_count)); 44 cudacattAddVectorKernet( 45 block_count, 46 per_block_thread_count, 47 dev_a, 48 dev_b, 48 dev_b, 49 dev_c, 50 array_size); 51 47 /* copy dev_c into c */</pre>
<pre>33 cudaMalloc((void**) &amp; dev a, array_size * sizeof(float)); 34 cudaMalloc((void**) &amp; dev_b, array_size * sizeof(float)); 35 cudaMemcpy(dev_a, a, array_size * sizeof(float)); 36 37 /* copy a and b into dev a and dev b */ 38 cudaMemcpy(dev_b, b, 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 44 (uint) ceil(array size / (float) per block_thread_count)); 44 cudacattAddVectorKernet( 45 block_count, 46 per_block_thread_count, 47 dev_a, 48 dev_b, 48 dev_b, 49 dev_c, 50 array_size); 51 47 /* copy dev_c into c */</pre>
<pre>33 cudaMalloc((void**) &amp; dev a, array_size * sizeof(float)); 34 cudaMalloc((void**) &amp; dev_b, array_size * sizeof(float)); 35 cudaMemcpy(dev_a, a, array_size * sizeof(float)); 36 37 /* copy a and b into dev a and dev b */ 38 cudaMemcpy(dev_b, b, 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 44 (uint) ceil(array size / (float) per block_thread_count)); 44 cudacattAddVectorKernet( 45 block_count, 46 per_block_thread_count, 47 dev_a, 48 dev_b, 48 dev_b, 49 dev_c, 50 array_size); 51 47 /* copy dev_c into c */</pre>
<pre>33 cudaMalloc((void**) &amp; dev a, array_size * sizeof(float)); 34 cudaMalloc((void**) &amp; dev_b, array_size * sizeof(float)); 35 cudaMemcpy(dev_a, a, array_size * sizeof(float)); 36 37 /* copy a and b into dev a and dev b */ 38 cudaMemcpy(dev_b, b, 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 44 (uint) ceil(array size / (float) per block_thread_count)); 44 cudacattAddVectorKernet( 45 block_count, 46 per_block_thread_count, 47 dev_a, 48 dev_b, 48 dev_b, 49 dev_c, 50 array_size); 51 47 /* copy dev_c into c */</pre>
<pre>34 cudaMalloc((void**) &amp; dev_b, array_size * sizeof(float)); 35 cudaMalloc((void**) &amp; dev_c, array_size * sizeof(float)); 36 /* copy a and b into dev_a and dev_b */ 37 cudaMemcpy(dev_b, a, array_size * sizeof(float), cudaMemcpyHostToDevice); 39 cudaMemcpy(dev_b, b, array_size * sizeof(float), cudaMemcpyHostToDevice); 40 /* call kernel to add the two arrays into dev_c */ 41 /* call kernel to add the two arrays into dev_c */ 42 uint block count, 43 (uint) ceil(array size / (float) per block thread count)); 44 cudaCalLAddVectorKernet( 45 block count, 46 per_block_thread_count, 47 dev_a, 48 dev_b, 48 dev_c, 50 array_size); 51 /* copy dev_c into c */</pre>
<pre>36 /* copy a and b into dev a and dev b */ 37 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</pre>
<pre>/* 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,</pre>
<pre>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 cudaCatLAddVectorKernel( 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 */</pre>
<pre>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 */</pre>
<pre>40 /* call kernel to add the two arrays into dev_c */ 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 cudaCatLAddvectorKernel( 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 */</pre>
<pre>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 */</pre>
<pre>42 uint block_count = min(max_block_count, 43 (uint) ceil(array size / (float) per_block thread count)); 44 cudacattAddvectorKernet( 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 */</pre>
<pre>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 */</pre>
<pre>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 */</pre>
<pre>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 */</pre>
<pre>46</pre>
47 dev_a, 48 dev_b, 49 dev_c, 50 array_size); 51 52 /* copy dev_c into c */
48 dev_b, 49 dev_c, 50 array_size); 51 52 /* copy dev_c into c */
49 dev_c, 50 array_size); 51 52 /* copy dev_c into c */
50 array_size); 51
51 52 /* copy dev_c into c */
52 /* copy dev_c into c */
54
55 /* check the output */
56
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_D);
67
68 return 0;
ob lettin o,
🗿 Menu 🔚 🔃 🚔 🦻 Edit (s179_2015_]e 🔟 Terminal 🔤 ~/Documents/test 🖆 Pictures

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

~/Documents/test/cuda_test.cu - Sublime Text (UNREGISTERED)	_ <b>_</b> ×
✓ ▶ test.cpp x cuda_test.cu x test.hpp x cuda_test.cuh x	v
1 #include "cuda_test.cuh" 2 3global	THE THE THE THE THE
4 void cudaAddVectorKernel(	
5 const float *a,	
6 const float *b, 7 float *c,	
8 const uint size)	
9 $\frac{1}{2}$	
10 /* get current thread's id */	
<pre>11 uint thread index = blockIdx.x * blockDim.x + threadIdx.x;</pre>	
<pre>13 /* while this thread is dealing with a valid index */</pre>	
14 while (thread index < size) { 15 /* add a and b into c */	
<pre>16 c[thread_index] = a[thread_index] + b[thread_index];</pre>	
<pre>18 /* advance thread id */ 19 thread index += blockDim.x * gridDim.x;</pre>	u de la construcción de la constru
20 }	
21 }	
22	
23 void cudaCallAddVectorKernel(	
24 const uint block_count,	
<pre>25 const uint per_block_thread_count,</pre>	
26 const float *a,	
27 const float *b,	
28 float *c, 29 const uint size)	
29 const uint size) 30 {	
<pre>30 1 31 cudaAddVectorKernel&lt;&lt;<block block="" count="" count,="" per="" thread="">&gt;&gt;(a, b, c, size);</block></pre>	
32 }	
33	

### Questions so far?

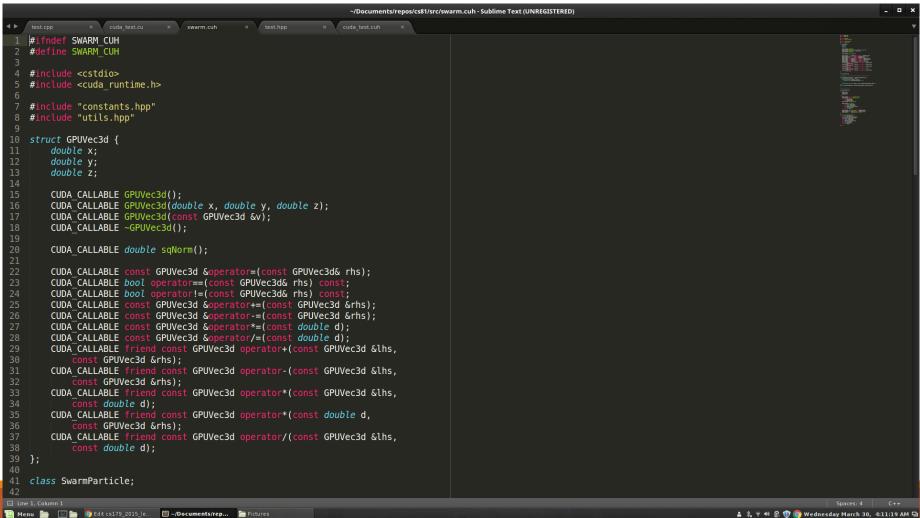
### Stuff that will be useful later

	~/Documents/test/cuda_test.cuh - Sublime Text (UNREGISTERED)	_ = ×
▲ test.cpp x cuda_test.cu x test.hpp x cuda_test.cuh	*	
1 #ifndef CUDA_TEST_HPP 2 #define CUDA_TEST_HPP		nn Nas Santa ann Milliann an
3 4 #include <stdio.h> 5 #include <stdlib.h> 6 #include <cuda_runtime.h> 7</cuda_runtime.h></stdlib.h></stdio.h>		
8 #ifdefCUDACC 9 #define CUDA_CALLABLEhostdevice 10 #else 11 #define CUDA_CALLABLE		
<pre>12 #endif 13 14 void cudaCallAddVectorKernel(</pre>		
<pre>15 const uint block_count, 16 const uint per_block_thread_count, 17 const float *a, 18 const float *b, 19 float *c,</pre>		
20 const uint size); 21 22 #endif		
☐ Line 19. Column 14		ces: 4 C++

~/Documents/test

🛄 Menu

### Stuff that will be useful later



### Stuff that will be useful later

		~/Documents/test/cuda test.cu - Sublime Text (UNREGISTERED)	×
<b>4</b> F	test.cpp		
1		clude "cuda test.cuh"	
2	"		
3	g	lobal	:
4		a cudaAddVectorKernel(	
5		const float *a,	
6		const float *b,	
7		float *c,	
8 9	r	const uint size)	
10	<u>{</u>	/* 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		<pre>while (thread index &lt; size) {</pre>	
15		/* add a and b into c */	
16		c[thread_index] = a[thread_index] + b[thread_index];	
17			
18		/* advance thread id */	U
19 20		thread_index += blockDim.x * gridDim.x;	
21	ı	J	
22	<u>,</u>		
23	voi	d cudaCallAddVectorKernel(	
24		<pre>const uint block_count,</pre>	
25		<pre>const uint per_block_thread_count,</pre>	
26		const float *a,	
27		<pre>const float *b, float *c</pre>	
28 29		float *c, const uint size)	
30	۲.		
31	Ľ	<pre>cudaAddVectorKernel&lt;&lt;<block block="" count="" count,="" per="" thread="">&gt;&gt;(a, b, c, size);</block></pre>	
32	}		
33	_		

### 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