CUDA C: PERFORMANCE MEASUREMENT AND TYPES OF MEMORY

Will Landau, Prof. Jarad Niemi

OUTLINE

- Measuring GPU performance
- Global vs. local vs. shared memory
- Implementing the dot product

Featured examples:

- `time.cu`
- `dot_product.cu`

EVENTS: MEASURING PERFORMANCE ON THE GPU

Event: a time stamp for the GPU.

Use events to measure the amount of time the GPU spends on a task.

TEMPLATE: `time.cu`

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

int main(){
  float   elapsedTime;
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord( start, 0 );

  // SOME GPU KERNEL YOU WANT TIMED HERE

  cudaEventRecord( stop, 0 );
  cudaEventSynchronize( stop );
  cudaEventElapsedTime( &elapsedTime, start, stop );
  cudaEventDestroy( start );
  cudaEventDestroy( stop );
  printf("GPU Time elapsed: %f\n", elapsedTime);
}
```

The variable, `elapsedTime`, is the GPU time spent on the task. You can now print it any way you like.

Note: only GPU elapsed time is measured, not CPU time.

GPU time and CPU time must be measured separately.

ASIDE: MEASURING CPU TIME

```
#include <stdio.h>
#include <time.h>

int main(){

  clock_t start = clock();

  // SOME CPU CODE YOU WANT TIMED HERE

  float elapsedTime = ((double)clock() - start) /
                                        CLOCKS_PER_SEC;

  printf("CPU Time elapsed: %f\n", elapsedTime);
}
```

EXAMPLE: pairwise_sum_timed.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <unistd.h>
#include <cuda.h>
#include <cuda_runtime.h>

/*
 * This program computes the sum of the elements of
 * vector v using the pairwise (cascading) sum algorithm.
 */

#define N 1024 // length of vector v. MUST BE A POWER OF 2.

// Fill the vector v with n random floating point numbers.
void vfill(float* v, int n){
  int i;
  for(i = 0; i < n; i++){
    v[i] = (float) rand() / RAND_MAX;
  }
}
```

```
// Print the vector v.
void vprint(float* v, int n){
  int i;
  printf("v = \n");
  for(i = 0; i < n; i++){
    printf("%7.3f\n", v[i]);
  }
  printf("\n");
}

// Pairwise-sum the elements of vector v and store the result in
  v[0].
__global__ void psum(float *v){
  int t = threadIdx.x; // Thread index.
  int n = blockDim.x; // Should be half the length of v.

  while (n != 0) {
    if(t < n)
      v[t] += v[t + n];
    __syncthreads();
    n /= 2;
  }
}
```

```
// Linear sum the elements of vector v and return the result
float lsum(float *v, int len){
  float s = 0;
  int i;
  for(i = 0; i < len; i++){
    s += v[i];
  }
  return s;
}


int main (void){
  if(N % 2){
    printf("\nERROR: N is not a power of 2. Exiting.\n");
    exit(1);
  }

  float *v_h, *v_d; // host and device copies of our vector,
    respectively

  // dynamically allocate memory on the host for v_h
  v_h = (float*) malloc(N * sizeof(*v_h));
```

```
// dynamically allocate memory on the device for v_d
cudaMalloc ((float**) &v_d, N *sizeof(*v_d));

// Fill v_h with N random floating point numbers.
vfill(v_h, N);

// Print v_h to the console
// vprint(v_h, N);

// Write the contents of v_h to v_d
cudaMemcpy( v_d, v_h, N * sizeof(float), cudaMemcpyHostToDevice
    );

// compute the linear sum of the elements of v_h on the CPU and
    return the result
// also, time the result.
clock_t start = clock();
float s = lsum(v_h, N);

float elapsedTime = ((float) clock() - start) / CLOCKS_PER_SEC;
printf("Linear Sum = %7.3f, CPU Time elapsed: %f\n", s,
   elapsedTime);

// Compute the pairwise sum of the elements of v_d and store
```

```
   the result in v_d[0].
// Also, time the computation.

float   gpuElapsedTime;
cudaEvent_t gpuStart, gpuStop;
cudaEventCreate(&gpuStart);
cudaEventCreate(&gpuStop);
cudaEventRecord( gpuStart, 0 );

psum<<< 1, N/2 >>>(v_d);

cudaEventRecord( gpuStop, 0 );
cudaEventSynchronize( gpuStop );
cudaEventElapsedTime( &gpuElapsedTime, gpuStart, gpuStop );
cudaEventDestroy( gpuStart );
cudaEventDestroy( gpuStop );

// Write the pairwise sum, v_d[0], to v_h[0].
cudaMemcpy(v_h, v_d, sizeof(float), cudaMemcpyDeviceToHost );

// Print the pairwise sum.
printf("Pairwise Sum = %7.3f, GPU Time elapsed: %f\n", v_h[0],
   gpuElapsedTime);
```

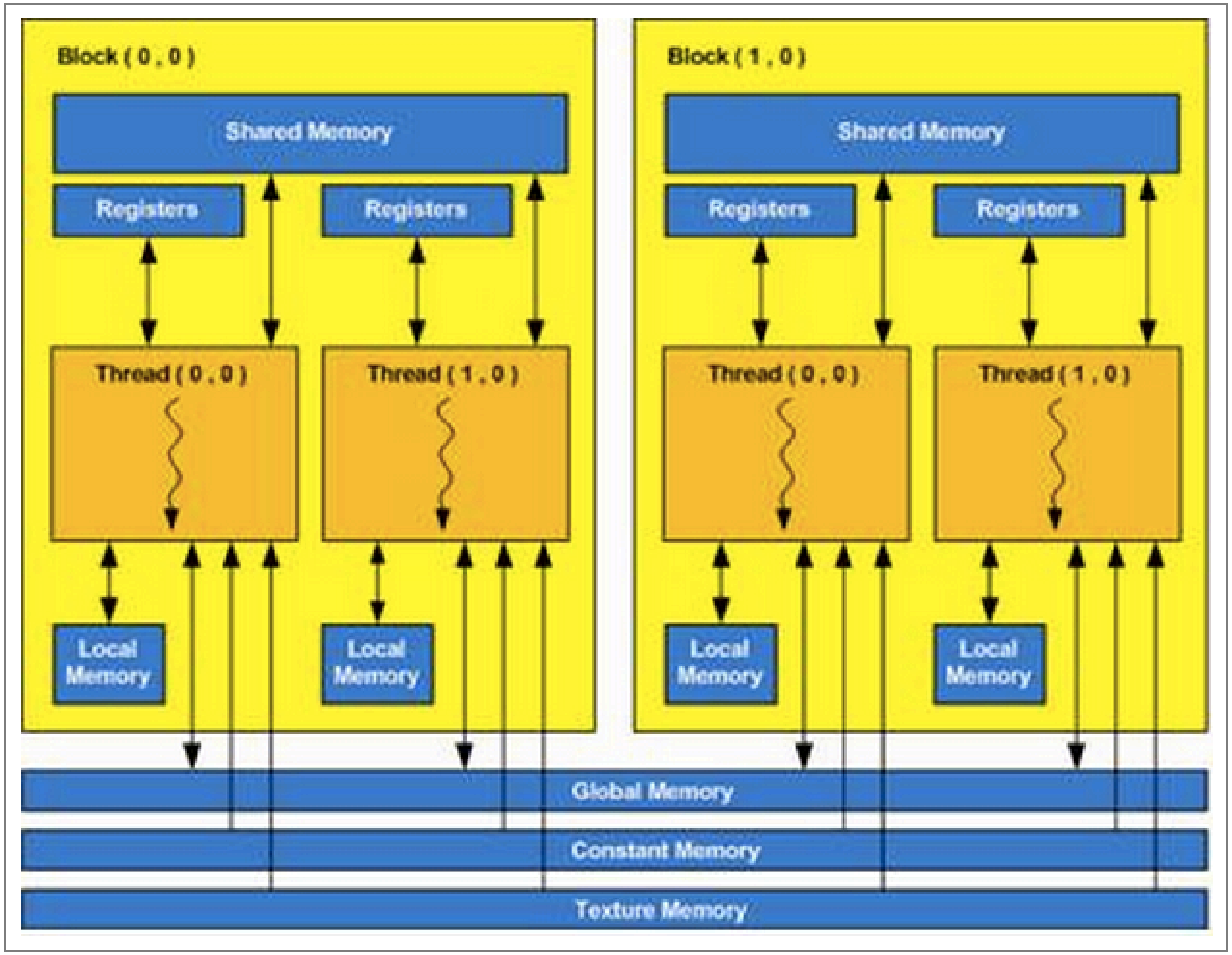
```
  // Free dynamically-allocated host memory
  free(v_h);

  // Free dynamically-allocated device memory
  cudaFree(&v_d);
}
```

OUTPUT

```
[landau@impact1 pairwise_sum_timed]$ make
nvcc pairwise_sum_timed.cu -o pairwise_sum_timed
[landau@impact1 pairwise_sum_timed]$ ./pairwise_sum_timed
Linear Sum = 518.913, CPU Time elapsed: 0.000000
Pairwise Sum = 518.913, GPU Time elapsed: 0.036640
[landau@impact1 pairwise_sum_timed]$
```

TYPES OF MEMORY



THOUGHT EXPERIMENT: GLOBAL VS. SHARED VS. LOCAL MEMORY

Let's say we have a kernel:

```
__global__ void kernel(int *a){
  *a = blockIdx.x * blockDim.x + threadIdx.x;
  int b  =  blockIdx.x * blockDim.x + threadIdx.x;
  __shared__ int c = blockIdx.x * blockDim.x + threadIdx.x;
}
```

What are *a, b, and c after a call to `kernel<<<3, 2>>>(a)`?

`*a`: There is one copy of `*a` in GLOBAL MEMORY common to all threads and blocks. Hence, the value of `*a` depends on which thread finishes last.

`b`: There are $3 \cdot 2 = 6$ copies of `b` in LOCAL MEMORY, one for each of the six threads. The values of `b` will be:

(Block ID, Thread ID)	(0, 0)	(0, 1)	(1, 0)	(1, 1)	(2, 0)	(2, 1)
Value of `b`	0	1	2	3	4	5

`c`: There are three copies of c in SHARED MEMORY, one for each block. The values of `c` might be:

(Block ID, Thread ID)	(0, 0)	(0, 1)	(1, 0)	(1, 1)	(2, 0)	(2, 1)
Value of `b`	0	0	3	3	4	4

depending on which thread finishes last within each block.

NOW, WE'RE READY FOR THE DOT PRODUCT

$$(a_0,\ a_1, \ldots,\ a_{15}) \bullet (b_0,\ b_1, \ldots,\ b_{15}) = a_0 b_0 + a_1 b_1 + \cdots + a_{15} b_{15}$$

The basic workflow is:

- Pass vectors `a` and `b` to the GPU.

- Give each block a sub vector of `a` and the analogous subvector of `b`, For example:

  Block 0 works on:

$$(a_0,\ a_1,\ \ldots,\ a_7)$$
$$(b_0,\ b_1,\ \ldots,\ b_7)$$

Block 1 works on:

$$(a_8,\ a_9,\ \ldots,\ a_{15})$$
$$(b_8,\ b_9,\ \ldots,\ b_{15})$$

- Within each block, compute a vector of pairwise products:

Block 0:

$$\texttt{cache} = (a_0 \cdot b_0,\ a_1 \cdot b_1,\ \ldots,\ a_7 \cdot b_7)$$

Block 1:

$$\texttt{cache} = (a_8 \cdot b_8,\ a_9 \cdot b_9,\ \ldots,\ a_{15} \cdot b_{15})$$

where `cache` is an array in shared memory.

- Within each block, compute the pairwise sum of `cache` and write it to `cache[0]`. In our example:

  Block 0:

  $$\texttt{cache[0]} = a_0 \cdot b_0 + a_1 \cdot b_1 + \cdots + a_7 \cdot b_7$$

  Block 1:

  $$\texttt{cache[0]} = a_8 \cdot b_8 + a_9 \cdot b_9 + \cdots + a_{15} \cdot b_{15}$$

- Fill a new vector in global memory, `partial_c`, with these partial dot products:

  $$\texttt{partial\_c} = (\texttt{block 0 cache[0]}, \texttt{block 1 cache[0]})$$

- Return to the CPU and compute the linear sum of `partial_c` and write it to `partial_sum[0]`. Then:

$$\texttt{partial\_sum[0]} \quad = a_0 \cdot b_0 + a_1 \cdot b_1 + \cdots + a_{15} \cdot b_{15}$$

First part of the code:

```
#include "../common/book.h"
#include <stdio.h>
#include <stdlib.h>
#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid = imin( 32, (N+threadsPerBlock-1) /
  threadsPerBlock );

__global__ void dot( float *a, float *b, float *partial_c ) {

  __shared__ float cache[threadsPerBlock];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float temp = 0;

  while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
  }
```

```
// set the cache values
cache[cacheIndex] = temp;
```

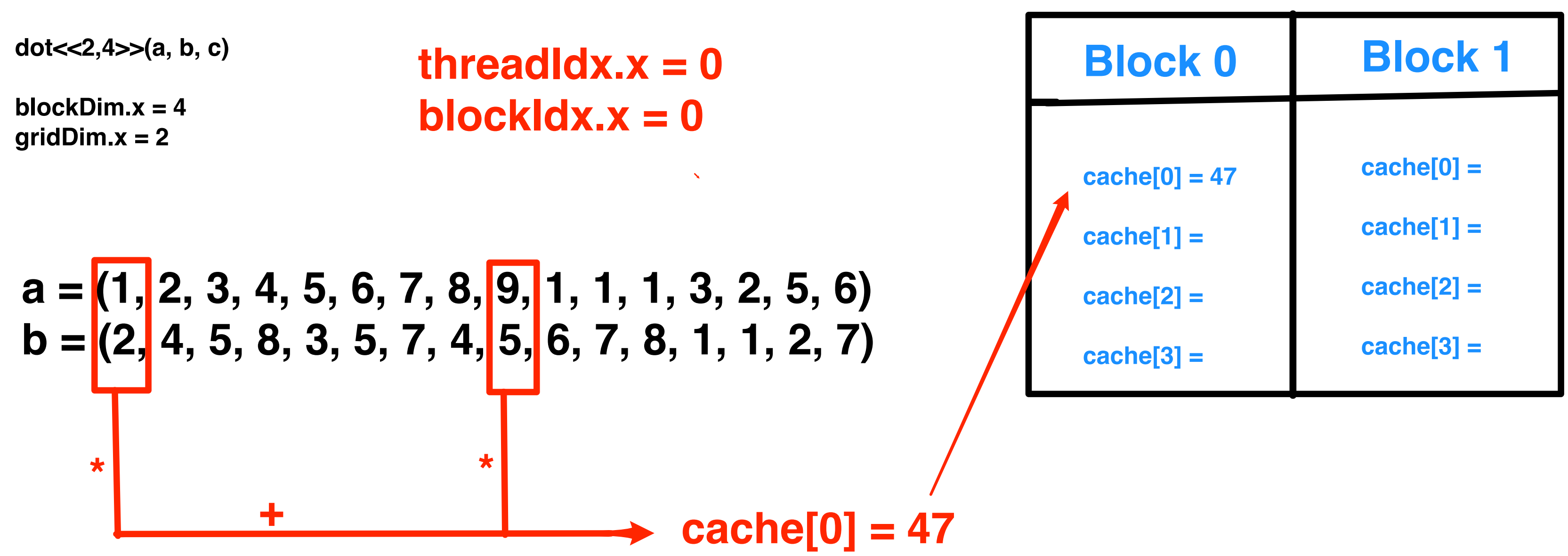
What the
code would do in a call to `dot<<<2, 4>>>(a, b, c)` with $N = 16$:

dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

Block 0	Block 1
cache[0] =	cache[0] =
cache[1] =	cache[1] =
cache[2] =	cache[2] =
cache[3] =	cache[3] =


dot<<2,4>>(a, b, c)
blockDim.x = 4
gridDim.x = 2
threadIdx.x = 0
blockIdx.x = 0
Block 0
Block 1
cache[0] = 47
cache[1] =
cache[2] =
cache[3] =
cache[0] =
cache[1] =
cache[2] =
cache[3] =
a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)
*
*
+
cache[0] = 47

`dot<<2,4>>(a, b, c)`

blockDim.x = 4
gridDim.x = 2

threadIdx.x = 1
blockIdx.x = 0

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

* \+ *

cache[1] = 14

Block 0	Block 1
cache[0] = 47	cache[0] =
cache[1] = 14	cache[1] =
cache[2] =	cache[2] =
cache[3] =	cache[3] =

dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

threadIdx.x = 2
blockIdx.x = 0

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

*   +   *

cache[2] = 22

Block 0	Block 1
cache[0] = 47	cache[0] =
cache[1] = 14	cache[1] =
cache[2] = 22	cache[2] =
cache[3] =	cache[3] =

dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

threadIdx.x = 3
blockIdx.x = 0

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

* * +

cache[3] = 40

Block 0	Block 1
cache[0] = 47	cache[0] =
cache[1] = 14	cache[1] =
cache[2] = 22	cache[2] =
cache[3] = 40	cache[3] =

dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

threadIdx.x = 0
blockIdx.x = 1

Block 0	Block 1
cache[0] = 47	cache[0] = 18
cache[1] = 14	cache[1] =
cache[2] = 22	cache[2] =
cache[3] = 40	cache[3] =

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

* * +

cache[0] = 18

dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

threadIdx.x = 1
blockIdx.x = 1

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

* * +

cache[1] = 32

Block 0	Block 1
cache[0] = 47	cache[0] = 18
cache[1] = 14	cache[1] = 32
cache[2] = 22	cache[2] =
cache[3] = 40	cache[3] =

`dot<<2,4>>(a, b, c)`

blockDim.x = 4
gridDim.x = 2

threadIdx.x = 2
blockIdx.x = 1

Block 0	Block 1
cache[0] = 47	cache[0] = 18
cache[1] = 14	cache[1] = 32
cache[2] = 22	cache[2] = 59
cache[3] = 40	cache[3] =

a = (1, 2, 3, 4, 5, 6, $\boxed{7,}$ 8, 9, 1, 1, 1, 3, 2, $\boxed{5,}$ 6)
b = (2, 4, 5, 8, 3, 5, $\boxed{7,}$ 4, 5, 6, 7, 8, 1, 1, $\boxed{2,}$ 7)

* + *

cache[2] = 59

dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

threadIdx.x = 3
blockIdx.x = 1

a = (1, 2, 3, 4, 5, 6, 7, $\boxed{8}$, 9, 1, 1, 1, 3, 2, 5, $\boxed{6}$)
b = (2, 4, 5, 8, 3, 5, 7, $\boxed{4}$, 5, 6, 7, 8, 1, 1, 2, $\boxed{7}$)

* * +

cache[3] = 74

Block 0	Block 1
cache[0] = 47	cache[0] = 18
cache[1] = 14	cache[1] = 32
cache[2] = 22	cache[2] = 59
cache[3] = 40	cache[3] = 74

We want to make sure that each block's copy of `cache` is filled up before we continue further.

Hence, the next line of code is:

```
// synchronize threads in this block
__syncthreads();
```

NEXT, WE EXECUTE A PAIRWISE SUM ON cache FOR EACH BLOCK

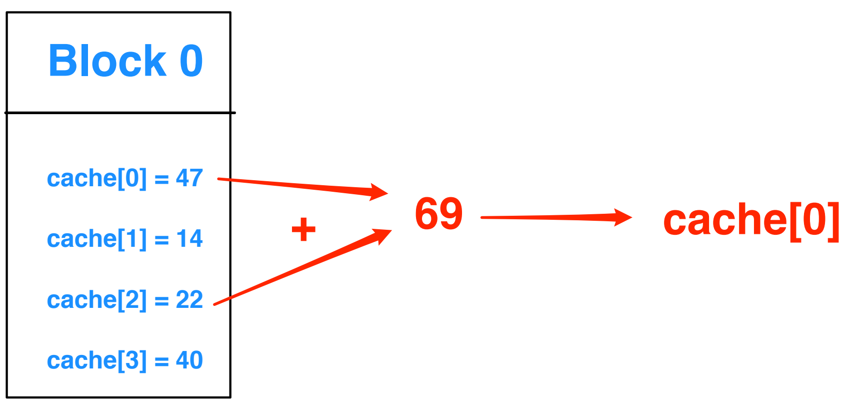
```
// for reductions, threadsPerBlock must be a power of 2 //
   because of the following code
int i = blockDim.x/2;
while (i != 0) {
  if (cacheIndex < i)
    cache[cacheIndex] += cache[cacheIndex + i];
  __syncthreads();     i /= 2;
}
```

WHAT'S GOING ON IN OUR CALL TO `dot<<<2, 4>>>(a, b, c)`

dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

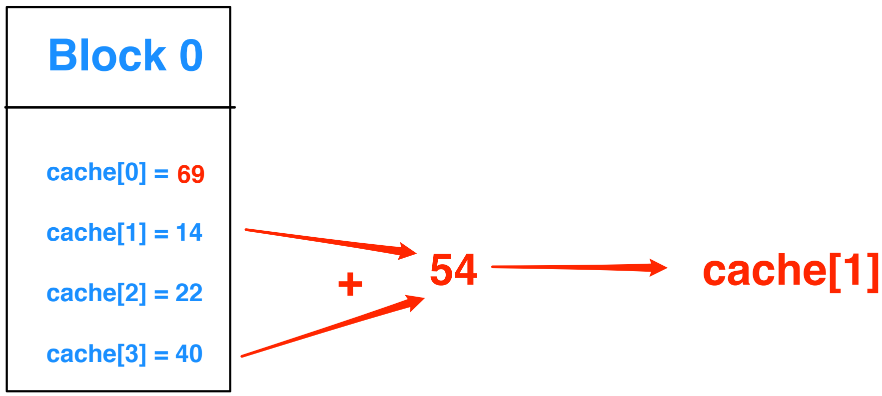
cacheIndex = threadIdx.x = 0
blockIdx.x = 0
i = 2



`dot<<2,4>>(a, b, c)`

blockDim.x = 4
gridDim.x = 2

cacheIndex = threadIdx.x = 1
blockIdx.x = 0
i = 2



dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

Block 0
cache[0] = 69
cache[1] = 54
cache[2] = 22
cache[3] = 40

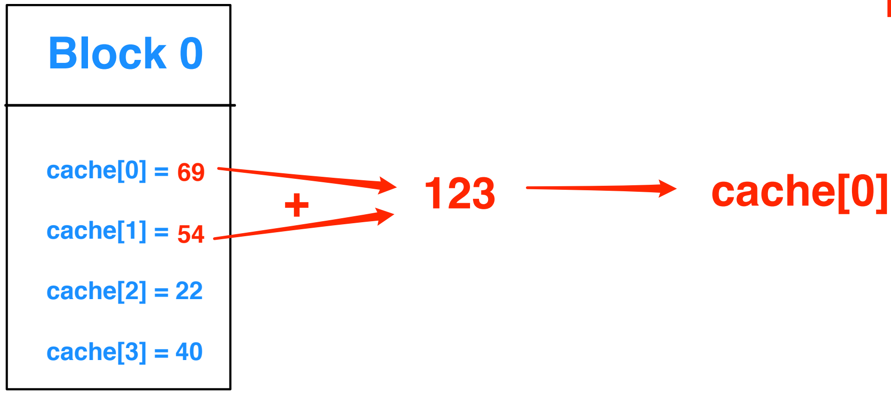
cacheIndex = threadIdx.x = 1
blockIdx.x = 0
i = 2

__synchthreads();

`dot<<2,4>>(a, b, c)`

blockDim.x = 4
gridDim.x = 2

cacheIndex = threadIdx.x = 0
blockIdx.x = 0
i = 1



dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

Block 0

cache[0] = 123

cache[1] = 54

cache[2] = 22

cache[3] = 40

cacheIndex = threadIdx.x = 0
blockIdx.x = 0
i = 1

__synchthreads();

dot<<2,4>>(a, b, c)

blockDim.x = 4
gridDim.x = 2

Block 0

cache[0] = 123

cache[1] = 54

cache[2] = 22

cache[3] = 40

cacheIndex = threadIdx.x = 0
blockIdx.x = 0
i = 0

i = 0, so end the pairwise sum.

The result for block 0 is cache[0] = 123.

Similarly, `cache[0]` for block 1 is 183.

Next:

```
  if (cacheIndex == 0)
    partial_c[blockIdx.x] = cache[0];
}
```

So now, `partial_c[0]` = 123 and `partial_c[1]` is 183.

We return from the kernel, `dot()`, and compute the linear sum of the elements of `partial_c`:

```
// finish up on the CPU side
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
  c += partial_c[i];
}
```

Now, `partial_c[0]` is the final answer.

COMPLETE CODE

```
#include "../common/book.h"
#include <stdio.h>
#include <stdlib.h>
#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid = imin( 32, (N+threadsPerBlock-1) /
  threadsPerBlock );

__global__ void dot( float *a, float *b, float *partial_c ) {

  __shared__ float cache[threadsPerBlock];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float temp = 0;

  while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
  }
```

```
  // set the cache values
  cache[cacheIndex] = temp;

  // synchronize threads in this block
  __syncthreads();

  // for reductions, threadsPerBlock must be a power of 2 //
     because of the following code
  int i = blockDim.x/2;
  while (i != 0) {
    if (cacheIndex < i)
      cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();     i /= 2;
  }

  if (cacheIndex == 0)
    partial_c[blockIdx.x] = cache[0];
}


int main( void ) {
  float *a, *b, c, *partial_c;
  float *dev_a, *dev_b, *dev_partial_c;
```

```
// allocate memory on the CPU side
a = (float*)malloc( N*sizeof(float) );
b = (float*)malloc( N*sizeof(float) );
partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c, blocksPerGrid
   *sizeof(float) ) );

// fill in the host memory with data
for (int i=0; i<N; i++) {
  a[i] = i;
  b[i] = i*2;
}

// copy the arrays    a    and    b    to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
   cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
   cudaMemcpyHostToDevice ) );

dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
```

```
    dev_partial_c );

  // copy the array 'c' back from the GPU to the CPU
  HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
    blocksPerGrid*sizeof(float),
                              cudaMemcpyDeviceToHost ) );

  // finish up on the CPU side
  c = 0;
  for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
  }

  #define sum_squares(x)( x * (x + 1) * (2 * x + 1) / 6 )

  printf("Does GPU value %.6g = %.6g?\n", c, 2 * sum_squares( (
    float)(N - 1) ) );

  // free memory on the GPU side
  cudaFree( dev_a );
  cudaFree( dev_b );
  cudaFree( dev_partial_c );
  // free memory on the CPU side
  free( a );
```

```
  free( b );
  free( partial_c );
}
```

OUTPUT

```
[landau@impact1 dot_product]$ nvcc dot_product.cu -o dot_product
[landau@impact1 dot_product]$ ./dot_product
Does GPU value 2.57236e+13 = 2.57236e+13?
[landau@impact1 dot_product]$
```

OUTLINE

- Respecting the SIMD paradigm
- Shared memory
- Implementing the dot product

Featured examples:

- `dot_product.cu`

LECTURE SERIES MATERIALS

These lecture slides, a tentative syllabus for the whole lecture series, and code are available at:

https://github.com/wlandau/gpu.

After logging into you home directory on impact1, type:

```
git clone https://github.com/wlandau/gpu
```

into the command line to download all the course materials.

REFERENCES

David B. Kirk and Wen-mei W. Hwu. “Programming Massively Parallel Processors: a Hands-on Approach.” Morgan Kaufman, 2010.

J. Sanders and E. Kandrot. *CUDA by Example*. Addison-Wesley, 2010.

Michael Romero and Rodrigo Urra. ”CUDA Programming.” Rochester Institute of Technology. http://cuda.ce.rit.edu/cuda_overview/cuda_overview.html