CUDA C: performance measurement and memory

Will Landau
Iowa State University

October 14, 2013
Outline

Timing kernels on the GPU

Memory
Outline

Timing kernels on the GPU

Memory
Measuring CPU time

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

int main()
{
    float elapsedTime;
    clock_t start = clock();

    // SOME CPU CODE YOU WANT TO TIME

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

    printf("CPU time elapsed: %f seconds \n", elapsedTime);
    return 0;
}
```
Events

- **Event**: a time stamp on the GPU
- Use events to measure GPU execution time.

**time.cu**:

```c
#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 WORK YOU WANT TIMED HERE
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsedTime, start, stop);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    printf("GPU Time elapsed: %f milliseconds\n", elapsedTime);
}
```

- GPU time and CPU time must be measured separately.
Example: pairwise_sum_timed.cu

```c
#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 = \
    for(i = 0; i < n; i++)
    {
        printf("%7.3f\n", v[i]);
    }
    printf("\n");
}
```
Example: pairwise_sum_timed.cu

```c
// 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;
}
```
Example: pairwise_sum_timed.cu

```c
int main (void) {
    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);
}
```
Example: pairwise_sum_timed.cu

```c
float elapsedTime = ((float) clock() - start) / CLOCKS_PER_SEC;
printf("Linear Sum = %7.3f, CPU Time elapsed: %f seconds\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); // time in milliseconds

cudaEventDestroy(gpuStart);
cudaEventDestroy(gpuStop);

// Write the pairwise sum, v_d[0], to v_h[0].
cudamemcpy(v_h, v_d, sizeof(float), cudaMemcpyDeviceToHost);
```
Example: pairwise_sum_timed.cu

```c
// Print the pairwise sum.
printf("Pairwise Sum = %7.3f, GPU Time elapsed: %f seconds\n", v_h[0], gpuElapsedTime / 1000.0);

// Free dynamically−allocated host memory
free(v_h);

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

Output:

```
1 > nvcc pairwise_sum_timed.cu −o pairwise_sum_timed
2 > ./pairwise_sum_timed
3 Linear Sum = 518.913, CPU Time elapsed: 0.000000 seconds
4 Pairwise Sum = 518.913, GPU Time elapsed: 0.000037 seconds
```
Outline

Timing kernels on the GPU

Memory
Types of memory

[Diagram showing types of memory, including shared memory, registers, local memory, global memory, constant memory, and texture memory.]
What happens in `myKernel<<<2, 2>>>(b, t)`?

```c
__global__ void myKernel(int *b_global, int *t_global){
    __shared__ int t;
    __shared__ int b;

    int b_local, t_local;

    *t_global = threadIdx.x;
    *b_global = blockIdx.x;

    t_shared = threadIdx.x;
    b_shared = blockIdx.x;

    t_local = threadIdx.x;
    b_local = blockIdx.x;
}
```
At the end of `myKernel<<<4, 7>>>(b, t)...`

- `b_local` and `t_local` are in local memory (or registers), so each thread gets a copy.

<table>
<thead>
<tr>
<th>(block, thread)</th>
<th>(0, 0)</th>
<th>(0, 1)</th>
<th>(1, 0)</th>
<th>(1, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b_local</code></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>t_local</code></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- `b_shared` and `t_shared` are in shared memory, so each block gets a copy.

<table>
<thead>
<tr>
<th>(block, thread)</th>
<th>(0, 0)</th>
<th>(0, 1)</th>
<th>(1, 0)</th>
<th>(1, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b_shared</code></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>t_shared</code></td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

- ? = last thread in its block to write to `t_shared`. 
At the end of `myKernel<<<4, 7>>>(b, t)...`

- `b_global` and `t_global` point to global memory, so there is only one copy.

<table>
<thead>
<tr>
<th>(block, thread)</th>
<th>(0, 0)</th>
<th>(0, 1)</th>
<th>(1, 0)</th>
<th>(1, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*b_global</code></td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td><code>*t_global</code></td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

- ? = last thread in its block to write to `*t_global`.
- ?? = block of the last thread to write to `*b_global`.

Will Landau (Iowa State University)  
CUDA C: performance measurement and memory  
October 14, 2013  
15 / 40
Example: dot product

\[ a \cdot b = (a_0, \ldots, a_{15}) \cdot (b_0, \ldots, b_{15}) = a_0 \cdot b_0 + \cdots + a_{15} \cdot b_{15} \]

1. In this example, spawn 2 blocks and 4 threads per block.
2. Give each block a subvector of \(a\) and an analogous subvector of \(b\).
   - Block 0:
     \[
     (a_0, a_1, a_2, a_3, a_8, a_9, a_{10}, a_{11})
     (b_0, b_1, b_2, b_3, b_8, b_9, b_{10}, b_{11})
     \]
   - Block 1:
     \[
     (a_4, a_5, a_6, a_7, a_{12}, a_{13}, a_{14}, a_{15})
     (b_4, b_5, b_6, b_7, b_{12}, b_{13}, b_{14}, b_{15})
     \]
Example: dot product

3. Create an array, cache, in shared memory:
   ▶ Block 0:
   
   cache[0] = a_0 \cdot b_0 + a_8 \cdot b_8
   cache[1] = a_1 \cdot b_1 + a_9 \cdot b_9
   cache[2] = a_2 \cdot b_2 + a_{10} \cdot b_{10}
   cache[3] = a_3 \cdot b_3 + a_{11} \cdot b_{11}

   ▶ Block 1:
   
   cache[0] = a_4 \cdot b_4 + a_{12} \cdot b_{12}
   cache[1] = a_5 \cdot b_5 + a_{13} \cdot b_{13}
   cache[2] = a_6 \cdot b_6 + a_{14} \cdot b_{14}
   cache[3] = a_7 \cdot b_7 + a_{15} \cdot b_{15}
Example: dot product

4. Compute the pairwise sum of cache in each block and write it to cache[0]
   - Block 0:
     \[
     \text{cache}[0] = a_0 \cdot b_0 + a_8 \cdot b_8 \\
     + a_1 \cdot b_1 + a_9 \cdot b_9 \\
     + a_2 \cdot b_2 + a_{10} \cdot b_{10} \\
     + a_3 \cdot b_3 + a_{11} \cdot b_{11}
     \]

   - Block 1:
     \[
     \text{cache}[0] = a_4 \cdot b_4 + a_{12} \cdot b_{12} \\
     + a_5 \cdot b_5 + a_{13} \cdot b_{13} \\
     + a_6 \cdot b_6 + a_{14} \cdot b_{14} \\
     + a_7 \cdot b_7 + a_{15} \cdot b_{15}
     \]
Example: dot product

5. Compute an array, `partial_c` in global memory:

   `partial_c[0] = cache[0]` from block 0
   `partial_c[1] = cache[0]` from block 1

6. The pairwise sum of `partial_c` is the final answer.
dot_product.cu

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

const int N = 32 * 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;
```
dot<<<2, 4>>>(a, b, c) with $N = 16$

$$\text{dot}<<<2, 4>>>(a, b, c)$$

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)

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache[0] =</td>
<td>cache[0] =</td>
</tr>
<tr>
<td>cache[1] =</td>
<td>cache[1] =</td>
</tr>
</tbody>
</table>
dot<<<2, 4>>>(a, b, c) with $N = 16$

**CUDA C: performance measurement and memory**

**Will Landau**

**Timing kernels on the GPU**

**Memory**

```
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)
threadIdx.x = 0
blockIdx.x = 0
cache[0] = 47
```

```
<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache[0] = 47</td>
<td></td>
</tr>
<tr>
<td>cache[1] =</td>
<td></td>
</tr>
<tr>
<td>cache[2] =</td>
<td></td>
</tr>
<tr>
<td>cache[3] =</td>
<td></td>
</tr>
</tbody>
</table>
```
dot<<<2, 4>>>(a, b, c) with $N = 16$

**Thread Indexes:**
- threadIdx.x = 1
- blockIdx.x = 0

**Block Dimensions:**
- blockDim.x = 4
- blockDim.y = 4
- blockDim.z = 4
- gridDim.x = 2
- gridDim.y = 2
- gridDim.z = 2

**Array Definitions:**
- `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)`

**Memory Access:**
```
dot<<<2, 4>>>(a, b, c)
```

**Cache Access:**
```
cache[0] = 47
cache[1] = 14
cache[2] = 
cache[3] = 
```

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

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

**Note:**
- `a` and `b` are arrays of length 16.
- The dot product is computed within a CUDA kernel.
- The memory access and cache usage are visualized for both blocks.
CUDA C: performance measurement and memory

Will Landau

Timing kernels on the GPU

Memory

dot\langle\langle 2, 4\rangle\rangle(a, b, c) with N = 16

dot\langle\langle 2, 4\rangle\rangle(a, b, c)
blockDim.x = 4
gridDim.x = 2

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 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
  cache[1] = 14
  cache[2] = 22
  cache[3] =

threadIdx.x = 2
blockIdx.x = 0

Block 0

Block 1

+ cache[2] = 22
dot<<<2, 4>>>(a, b, c) with $N = 16$
dot<<<2, 4>>>(a, b, c) with $N = 16$

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

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

threadIdx.x = 0
blockIdx.x = 1

blockDim.x = 4
gridDim.x = 2

Block 0

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

Block 1

| cache[0] = 18 |
| cache[1] = |
| cache[2] = |
| cache[3] = |

$\text{cache[0]} = 18$
dot\( \lll 2, 4 \rrr \) \((a, b, c)\) with \(N = 16\)

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache[0] = 47</td>
<td>cache[0] = 18</td>
</tr>
</tbody>
</table>

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

\(\text{threadIdx.x} = 1\)
\(\text{blockIdx.x} = 1\)

\(\text{cache}[1] = 32\)
dot<<<2, 4>>>(a, b, c) with $N = 16$

```
dot<<2,4>>(a, b, c)
blockDim.x = 4
gridDim.x = 2
```

```
threadIdx.x = 2
blockIdx.x = 1
```

```
<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache[0] = 47</td>
<td></td>
</tr>
<tr>
<td>cache[1] = 14</td>
<td></td>
</tr>
<tr>
<td>cache[2] = 22</td>
<td></td>
</tr>
<tr>
<td>cache[3] = 40</td>
<td></td>
</tr>
<tr>
<td>cache[0] = 18</td>
<td></td>
</tr>
<tr>
<td>cache[1] = 32</td>
<td></td>
</tr>
<tr>
<td>cache[2] = 59</td>
<td></td>
</tr>
<tr>
<td>cache[3] =</td>
<td></td>
</tr>
</tbody>
</table>
dot<<<2, 4>>>(a, b, c) with $N = 16$

```
dot<<2,4>>(a, b, c)
blockDim.x = 4
gridDim.x = 2
Block 1
Block 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)
```

```
threadIdx.x = 3
blockIdx.x = 1
```

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

```
cache[0] = 18
cache[1] = 32
cache[2] = 59
```

```
cache[3] = 74
```

```
+ * cache[3] = 74
```
Make sure cache is full before continuing.

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

Execute a pairwise sum of cache for each block.

```
26 // threadsPerBlock must be a power of 2
27 int i = blockDim.x/2;
28 while (i != 0) {
29     if (cacheIndex < i)
30         cache[cacheIndex] += cache[cacheIndex + i ];
31         __syncthreads();
32     i /= 2;
33 }
```

Record the result in partial_c.

```
34 if (cacheIndex == 0)
35     partial_c[blockIdx.x] = cache[0];
36 }
```
\texttt{dot<<<2, 4>>>(a, b, c)} with \( N = 16 \)

\begin{align*}
\text{cacheIndex} &= \text{threadIdx.x} = 0 \\
\text{blockIdx.x} &= 0 \\
i &= 2
\end{align*}

Block 0

\begin{align*}
\text{cache}[0] &= 47 \\
\text{cache}[1] &= 14 \\
\text{cache}[2] &= 22 \\
\text{cache}[3] &= 40
\end{align*}

\[ + \quad 69 \quad \text{cache}[0] \]
dot<<<2, 4>>>(a, b, c) with $N = 16$
dot<<<2, 4>>>(a, b, c) with $N = 16$

dot<<<2,4>>>(a, b, c)
blockDim.x = 4
gridDim.x = 2

Block 0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>54</td>
<td>22</td>
<td>40</td>
</tr>
</tbody>
</table>

```
cachIndex = threadIdx.x = 1
blockIdx.x = 0
i = 2

__synchthreads();
```
$\text{dot}<<<2, 4>>>(a, b, c)$ with $N = 16$

```
blockDim.x = 4
gridDim.x = 2
```

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

```
cachIndex = threadIdx.x = 0
blockIdx.x = 0
i = 1
```

```
Block 0
```

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

```
+ 123 -> cache[0]
```
dot<<<2, 4>>>(a, b, c) with $N = 16$

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

__synchthreads();
dot<<<2, 4>>>(a, b, c) with $N = 16$

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

blockDim.x = 4
gridDim.x = 2

<table>
<thead>
<tr>
<th>Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache[0] = 123</td>
</tr>
<tr>
<td>cache[1] = 54</td>
</tr>
<tr>
<td>cache[2] = 22</td>
</tr>
<tr>
<td>cache[3] = 40</td>
</tr>
</tbody>
</table>

```
cachIndex = threadIdx.x = 0
blockIdx.x = 0
i = 0
```

i = 0, so end the pairwise sum.

The result for block 0 is cache[0] = 123.
Sum up partial_c inside int main()

    dot<<<blocksPerGrid, threadsPerBlock>>>( dev_a, dev_b, dev_partial_c);

    // copy partial_c to the CPU
    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];
    }
Outline

Timing kernels on the GPU

Memory
Resources

- Guides:

- Code:
  - time.cu
  - pairwise_sum_timed.cu
  - dot_product.cu
That’s all for today.

Series materials are available at http://will-landau.com/gpu.