EN 600.420 Parallel Programming

Companion Slides to Mark Harris’ Reduction
CUDA allows some synchronization between threads via `__syncthreads()`.

Unfortunately, this only synchronizes within a multi-processor: there is no way to synchronize across multiprocessors.

To get maximum performance we want to use all multiprocessors!
To introduce a global synchronization point: 
*Make multiple kernel invocations*

- Relatively efficient

- We can combine the use of thread block synchronization, and grid synchronization
- Reduce in a thread block as much as we can
Blocks can be efficiently reduced with synchronization (reduces the data to $\frac{1}{512}$th)
Block Reduction

- Start with half the number of threads as data items, and reduce pairs
Block Reduction

- Iteratively reduce, halving the number of threads that do work in each step
Iteratively reduce, halving the number of threads that do work in each step
for(unsigned int i=0; i<BLOCKSIZE; i++)
    Reduction<<<8*BLOCKSIZE*BLOCKSIZE, BLOCKSIZE>>>(pDeviceData + 8*i*BLOCKSIZE*BLOCKSIZE, 1);

Reduction<<<8*BLOCKSIZE, BLOCKSIZE>>>(pDeviceData, BLOCKSIZE);
Reduction<<<8, BLOCKSIZE>>>(pDeviceData, BLOCKSIZE*BLOCKSIZE);
__global__ void Reduction(float* pData, unsigned Stride)
{
    unsigned tid = threadIdx.x;
    unsigned i = blockIdx.x * blockDim.x + threadIdx.x;

    for (unsigned j=1;j<blockDim.x; j*=2)
    {
        if (tid % (2*j) == 0)
            pData[Stride*i] += pData[Stride*i + Stride*j];
        __syncthreads();
    }
}
Performance

- Block size = 256
- Vector Size = 128 million elements (=512MB)

- Reduction Time: 408ms
- Bandwidth: 1.2GBs$^{-1}$

- Can we do better? .... Yes, much better!
### Instructions / Branching

<table>
<thead>
<tr>
<th>Total Instructions</th>
<th>Total Branches</th>
<th>Divergent Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>62,907,050</td>
<td>16,806,279</td>
<td>1,545,783</td>
</tr>
<tr>
<td>26.72%</td>
<td>9.20%</td>
<td></td>
</tr>
</tbody>
</table>

### Memory Access

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Coherent</th>
<th>Incoherent</th>
<th>Coherency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global Reads</strong></td>
<td>22,018,442</td>
<td>15,842,752</td>
<td>6,175,690</td>
<td>28.05%</td>
</tr>
<tr>
<td><strong>Global Writes</strong></td>
<td>2,0854,996</td>
<td>2,0738,964</td>
<td>116,032</td>
<td>99.44%</td>
</tr>
</tbody>
</table>
Instructions / Branching

<table>
<thead>
<tr>
<th>Total Instructions</th>
<th>Total Branches</th>
<th>Divergent Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>62,907,050</td>
<td>16,806,279</td>
<td>1,545,783</td>
</tr>
<tr>
<td></td>
<td>26.72%</td>
<td>9.20%</td>
</tr>
</tbody>
</table>

Memory Access

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Coherent</th>
<th>Incoherent</th>
<th>Coherency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Reads</td>
<td>22,018,442</td>
<td>6,175,690</td>
<td>15,842,752</td>
<td>28.05%</td>
</tr>
<tr>
<td>Global Writes</td>
<td>2,085,996</td>
<td>2,073,964</td>
<td>116,032</td>
<td>99.44%</td>
</tr>
</tbody>
</table>
Problems Identified:
- High Memory Use
- ~10% of instructions are diverging branches
- ~72% of memory loads incoherent

Coherent memory accesses are sequential and aligned
Problems Identified:

- High Memory Use
- ~10% of instructions are diverging branches
- ~72% of memory loads incoherent

- Coherent memory accesses are sequential and aligned
Coalesced Reads/Writes

- For maximum bandwidth, the requests from all the threads in a half warp should be able to form a single **sequential** and **aligned** memory access
  - Aligned to $16 \times \text{sizeof(type)}$

- Non-sequential or non-aligned requests are serialized into multiple memory access
  - Each individual request sequential and aligned
Coalesced Reads/Writes

- Aligned?
- Sequential?
Coalesced Reads/Writes

- **Aligned?**
  - Yes, 128 = 32 \times 4

- **Sequential?**
  - Yes
Coalesced Reads/Writes

- Aligned?
- Sequential?
Coalesced Reads/Writes

- Aligned?
  - Yes, 128 = 32 x 4

- Sequential?
  - Yes
Coalesced Reads/Writes

- Aligned?
- Sequential?
Coalesced Reads/Writes

- Aligned?
  - No, 132 not multiple of 16 x 4 = 64

- Sequential?
  - Yes
Coalesced Reads/Writes

- Aligned?
- Sequential?
Coalesced Reads/Writes

- **Aligned?**
  - Yes

- **Sequential?**
  - No
Step 1: Use shared memory to reduce memory loads/stores

We can perform the reduction of a block completely in shared memory:
- Read in all values once
- Write out one result
__global__ void Reduction(float* pData, unsigned Stride)
{
    extern __shared__ float SharedData[];
    unsigned tid = threadIdx.x;
    unsigned i = blockIdx.x * blockDim.x + threadIdx.x;

    SharedData[tid] = pData[Stride*i];
    __syncthreads();

    for (unsigned j=1; j<blockDim.x; j*=2)
    {
        if (tid % (2*j) == 0)
            SharedData[tid] += SharedData[tid+j];
        __syncthreads();
    }

    if (tid == 0)
        pData[Stride*i] = SharedData[tid];
}
for(unsigned int i=0; i<BLOCKSIZE; i++)
    Reduction<<<8*BLOCKSIZE*BLOCKSIZE, BLOCKSIZE, BLOCKSIZE*4>>> 
    (pDeviceData + 8*i*BLOCKSIZE*BLOCKSIZE, 1);

Reduction<<<8*BLOCKSIZE, BLOCKSIZE, BLOCKSIZE*4>>> 
    (pDeviceData, BLOCKSIZE);
Reduction<<<8, BLOCKSIZE, BLOCKSIZE*4>>> 
    (pDeviceData, BLOCKSIZE*BLOCKSIZE);

- Third argument in <<<>>> is size of shared memory region
  - Allocated at kernel launch
Performance

- Block size = 256
- Vector Size = 128 million elements (=512MB)
- Reduction Time: 312ms
- Bandwidth: 1.6GBs$^{-1}$
- 1.31 times speedup
Profiling

- Instructions / Branching

<table>
<thead>
<tr>
<th>Total Instructions</th>
<th>Total Branches</th>
<th>Divergent Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>49,204,423</td>
<td>9,244,057</td>
<td>32,897</td>
</tr>
<tr>
<td>18.79%</td>
<td>0.36%</td>
<td></td>
</tr>
</tbody>
</table>

- Memory Access

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Coherent</th>
<th>Incoherent</th>
<th>Coherency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Reads</td>
<td>1,114,624</td>
<td>1,048,576</td>
<td>66,048</td>
<td>94.07%</td>
</tr>
<tr>
<td>Global Writes</td>
<td>263,176</td>
<td>263,176</td>
<td>0</td>
<td>100.00%</td>
</tr>
</tbody>
</table>
Problems

- Use of Modulo operator
- Inefficient operation in the inner loop

```c
for (unsigned j=1; j<blockDim.x; j*=2) {
    if (tid % (2*j) == 0)
        SharedData[tid] += SharedData[tid+j];
    __syncthreads();
}
```
Step 2: Change to strided indexing, Branch in inner loop:
  - Helps compiler turn branches into predicated code

```c
for (unsigned j=1; j<blockDim.x; j*=2) {
    int k = 2 * j * tid;
    if (k < blockDim.x)
        SharedData[k] += SharedData[k+j];
    __syncthreads();
}
```
Branches lead to divergence
- if, for, do, etc.

Compiler converts to predicated code
- When possible and within limits (7 ops)

```c
for (unsigned i=0, j=0; i<n; i++)
{
    /* branch version */
    if ( src[i] < V )
        j++;

    /* predicated version */
    bool b = (src[i] < V);
    j+=b;
}
```
Performance

- Block size = 256
- Vector Size = 128 million elements (=512MB)
- Reduction Time: 135ms
- Bandwidth: 3.7GBs$^{-1}$
- 2.31 times speedup
- 3.02 times speedup overall
Are there shared memory bank conflicts?

```c
for (unsigned j=1; j<blockDim.x; j*=2)
{
    int k = 2 * j * tid;

    if (k < blockDim.x)
        SharedData[k] += SharedData[k+j];

    __syncthreads();
}
```
To increase memory bandwidth in parallel systems, memory is often arranged in banks.

Addresses are interleaved between banks.
To increase memory bandwidth in parallel systems, memory is often arranged in *banks*.

With *n* banks, can perform *n* simultaneous memory operations.
To increase memory bandwidth in parallel systems, memory is often arranged in banks.

When two threads try to access the same bank at the same time a bank conflict occurs.
Bank Conflicts

- Bank conflicts are resolved by serializing memory accesses
- An $n$-way bank conflict means that there are $n$ threads accessing the same bank
- Resolved by issuing $n$ sequential requests to the bank
A shared memory request for a warp is performed as two requests: one for each half warp in alternating clock cycles.

Bank conflicts occur only within half warps.
No Bank Conflicts
Bank Conflicts

- No Bank Conflicts
Bank Conflicts

- 2-way Bank Conflict
Bank Conflicts

- 8-way Bank Conflict
Broadcast Mode

- Shared memory has a special bank conflict case
- A number of threads can read from the same address without conflict
- 16-way Bank Conflict?
- If reading same address in same bank: \(\rightarrow\) No Conflict
Registers

- Registers are generally zero clock cycle access.
- Read-after-write conflicts can occur but can be ignored after there are 192 active threads per multi-processor.

- Registers are banked (unknown config).
- Compiler reduces bank conflicts best when threads per block is a multiple of 64.
Are there shared memory bank conflicts?

```c
for (unsigned j=1; j<blockDim.x; j*=2) {
    int k = 2 * j * tid;

    if (k < blockDim.x)
        SharedData[k] += SharedData[k+j];

    __syncthreads();
}
```
Problems

- Shared Memory Bank conflicts:
Problems

- Shared Memory Bank conflicts:

```
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
       ↑       ↑       ↑       ↑       ↑       ↑
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
       ↑       ↑       ↑       ↑       ↑       ↑
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
       ↑       ↑       ↑       ↑       ↑       ↑
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
```
Step 3: Change to sequential addressing:
Step 3: Change to sequential addressing:
Step 3: Change to sequential addressing:

```cpp
for (unsigned int j=blockDim.x >> 1; j>0; j>>=1) {
    if (tid < j)
        SharedData[tid] += SharedData[tid+j];
    __syncthreads();
}
```
Performance

- Block size = 256
- Vector Size = 128 million elements (=512MB)
- Reduction Time: 70ms
- Bandwidth: 7.2GBs⁻¹
- 1.93 times speedup
- 5.81 times speedup overall
Problem

- How many threads are there?
- How many threads are doing work?

```c
for (unsigned int j=blockDim.x >> 1; j>0; j>>=1) {
    if (tid < j)
        SharedData[tid] += SharedData[tid+j];
    __syncthreads();
}
```
Problem

- How many threads are there?
- How many threads are doing work?

```c
for (unsigned int j=blockDim.x >> 1; j>0; j>>=1)
{
    if (tid < j)
        SharedData[tid] += SharedData[tid+j];
    __syncthreads();
}
```
Problem

- How many threads are there?
  - blockDim.x
- How many threads are doing work?
  - 1st iteration: blockDim.x / 2
  - 2nd iteration: blockDim.x / 4
  - ....
Problem

- How many threads are there?
  - blockDim.x
- How many threads are doing work?
  - 1\textsuperscript{st} iteration: blockDim.x / 2
  - 2\textsuperscript{nd} iteration: blockDim.x / 4
  - ....

- Half the threads never do anything!
Step 4: We need to give each block twice as much data, and launch half as many blocks.
We need to give each block twice as much data, and launch half as many blocks

```c
for(unsigned int i=0; i<BLOCKSIZE; i++)
    Reduction<<<8*BLOKCSIZE, BLOCKSIZE, BLOCKSIZE*4>>>> (pDeviceData + 8*i*BLOKCSIZE*BLOCKSIZE, 1);
Reduction<<<8*BLOKCSIZE, BLOCKSIZE, BLOCKSIZE*4>>>> (pDeviceData, BLOCKSIZE);
Reduction<<<8, BLOCKSIZE, BLOCKSIZE*4>>>> (pDeviceData, BLOCKSIZE*BLOCKSIZE);
```
We need to give each block twice as much data, and launch half as many blocks

```c
for(unsigned int i=0; i<BLOCKSIZE; i++)
    Reduction<<<4*BLOCKSIZE, BLOCKSIZE, BLOCKSIZE*4>>>(pDeviceData + 8*i*BLOCKSIZE*BLOCKSIZE, 1);

Reduction<<<2*BLOCKSIZE, BLOCKSIZE, BLOCKSIZE*4>>>(pDeviceData, 2*BLOCKSIZE);
Reduction<<<1, BLOCKSIZE, BLOCKSIZE*4>>>(pDeviceData, 4*BLOCKSIZE*BLOCKSIZE);
```
In kernel, perform two memory lookups and the first addition outside the loop:

```c
unsigned i = blockIdx.x * 2*blockDim.x + threadIdx.x;
SharedData[tid] = pData[Stride*i] + pData[Stride*(i + blockDim.x)];
```
Performance

- Block size = 256
- Vector Size = 128 million elements (=512MB)

- Reduction Time: 41ms
- Bandwidth: 12.3GBs⁻¹

- 1.70 times speedup
- 9.87 times speedup overall
Profiling

- Instructions / Branching

<table>
<thead>
<tr>
<th>Total Instructions</th>
<th>Total Branches</th>
<th>Divergent Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,150,578</td>
<td>2,496,144</td>
<td>16,422</td>
</tr>
<tr>
<td>60.14%</td>
<td>0.66%</td>
<td></td>
</tr>
</tbody>
</table>

- Memory Access

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Coherent</th>
<th>Incoherent</th>
<th>Coherency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Reads</td>
<td>1,081,344</td>
<td>1,048,064</td>
<td>33,280</td>
<td>96.92%</td>
</tr>
<tr>
<td>Global Writes</td>
<td>131,268</td>
<td>131,268</td>
<td>0</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
Instructions / Branching

<table>
<thead>
<tr>
<th>Total Instructions</th>
<th>Total Branches</th>
<th>Divergent Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,150,578</td>
<td>2,496,144</td>
<td>16,422</td>
</tr>
<tr>
<td></td>
<td><strong>60.14%</strong></td>
<td><strong>0.66%</strong></td>
</tr>
</tbody>
</table>

Memory Access

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Coherent</th>
<th>Incoherent</th>
<th>Coherency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Reads</td>
<td>1,081,344</td>
<td>1,048,064</td>
<td>33,280</td>
<td>96.92%</td>
</tr>
<tr>
<td>Global Writes</td>
<td>131,268</td>
<td>131,268</td>
<td>0</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
Problem

- Large number of Branch instructions:

```c
for (unsigned int j=blockDim.x >> 1; j>0; j>>=1)
{
    if (tid < j)
        SharedData[tid] += SharedData[tid+j];
    __syncthreads();
}
```
Optimization

- **Step 5:** Unroll the loop
**Step 5:** Unroll the loop (blockDim.x = 256)

```cpp
if (tid < 128)
    SharedData[tid] += SharedData[tid+128];
__syncthreads();
if (tid < 64)
    SharedData[tid] += SharedData[tid+64];
__syncthreads();
...
if (tid < 1)
    SharedData[tid] += SharedData[tid+1];
__syncthreads();
```
Optimization

- Wait: The compiler should do this for us!
Wait: The compiler should do this for us!

It probably does, but we can do better

- We know blockDim.x = 256
- We also know something else...
Question: What happens when tid <= 32?
**Question:** What happens when tid <= 32?

**Answer:** There is only one warp executing

- A warp executes in SIMD lockstep
  - No `__syncthreads()` needed!
  - The if (tid < x) doesn’t save any work
if (tid < 128)
    SharedData[tid] += SharedData[tid+128];
__syncthreads();
if (tid < 64)
    SharedData[tid] += SharedData[tid+64];
__syncthreads();

if (tid < 32)
{
    SharedData[tid] += SharedData[tid+32];
    SharedData[tid] += SharedData[tid+16];
    SharedData[tid] += SharedData[tid+8];
    SharedData[tid] += SharedData[tid+4];
    SharedData[tid] += SharedData[tid+2];
    SharedData[tid] += SharedData[tid+1];
}
### Performance

- Block size = 256
- Vector Size = 128 million elements (=512MB)
- Reduction Time: 22ms
- Bandwidth: 22GBs⁻¹
- 1.77 times speedup
- 17.45 times speedup overall
## Profiling

### Instructions / Branching

<table>
<thead>
<tr>
<th>Total Instructions</th>
<th>Total Branches</th>
<th>Divergent Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,768,800</td>
<td>919,464</td>
<td>16,419</td>
</tr>
<tr>
<td></td>
<td>33.21%</td>
<td>1.79%</td>
</tr>
</tbody>
</table>

### Memory Access

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Coherent</th>
<th>Incoherent</th>
<th>Coherency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Reads</td>
<td>1,081,120</td>
<td>1,048,352</td>
<td>32,768</td>
<td>96.97%</td>
</tr>
<tr>
<td>Global Writes</td>
<td>131,300</td>
<td>131,300</td>
<td>0</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
In a series of 5 optimizations:
1. Using Shared Memory
2. Reducing Thread Divergence, Removed Modulo
3. Reduced Bank Conflicts
4. Maximizing Thread Utilization
5. Maximizing Arithmetic Density

We have increased performance by a factor of 17.5
Summary
Summary – Instructions

The bar chart illustrates the distribution of total instructions, total branches, and divergent branches across different code revisions. The x-axis represents the code revision number, while the y-axis shows the instruction count. The chart highlights the evolution of code complexity and branching behavior across revisions.
What Else?

- Is this kernel fully optimized?
  - Probably not
- What else could we do?
Is this kernel fully optimized?
  - Probably not

What else could we do?
  - Can eliminate all incoherent memory access
  - Likely get better performance if each thread sums more elements