CUDA Optimization Issues
Case study: parallel prefix scan w/ CUDA

October 21, 2015
Quote of the Day

“Attitude is a little thing that makes a big difference.”
-- Winston Churchill
1874-1965
Before We Get Started

- Issues covered last time:
  - CUDA profiling and optimization
    - Lazy man’s approach (environment variable)
    - Profiling with `nvvp`

- Today’s topics
  - Further CUDA optimization issues
  - Case study: parallel prefix scan in CUDA

- Assignment:
  - HW05 – due today at 11:59 PM
  - HW06 – posted today, due in one week
Tiling [Blocking]:
A Fundamental CUDA Programming Pattern

- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a block of threads
  - No different from cache blocking!
    - Except you now have full control over it

- Provides several significant performance benefits
  - Working in shared memory reduces memory latency dramatically
  - More likely to have address access patterns that coalesce well on load/store to shared memory
Fundamental CUDA Pattern: Tiling

- Partition data into subsets that fit into __shared__ memory

This is your data: one big chunk, about to be broken into subsets suitable to be stored into shared memory
Process each data subset with one thread block
Fundamental CUDA Pattern: Tiling

- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
Fundamental CUDA Pattern: Tiling

- Perform the computation on the subset from shared memory
Fundamental CUDA Pattern: Tiling

- Copy the result from **shared** memory back to global memory
A large number of CUDA kernels are built this way.

However, tiling [blocking] may not be the only approach to solving a problem, sometimes it might not apply…

Two questions that can guide you in deciding if tiling is it:
- Does a thread require several loads from global memory to serve its purpose?
- Could data used by a thread be used by some other thread in the same block?
- If answer to both questions is “yes”, consider tiling as a design pattern

The answer to these two questions above is not always obvious
- Sometime it’s useful to craft an altogether new approach (algorithm) that is capable of using tiling: you force the answers to be “yes”
CUDA Optimization:
Execution Configuration Heuristics
### Technical Specifications and Features

#### [Short Detour]

- **This is us:** most GPUs on Euler are Fermi

#### Technical Specifications

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>65535</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td></td>
<td></td>
<td>512</td>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td></td>
<td></td>
<td>512</td>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td></td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td></td>
<td>32 K</td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 K</td>
<td></td>
<td></td>
<td>48 K</td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td></td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td></td>
<td>512 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td>64 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td>2 million</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Feature Support

<table>
<thead>
<tr>
<th>Feature Support (Unlisted features are supported for all compute capabilities)</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer atomic functions operating on 32-bit words in global memory (Section B.11)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Integer atomic functions operating on 64-bit words in global memory (Section B.11)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Integer atomic functions operating on 32-bit words in shared memory (Section B.11)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Warp vote functions (Section B.12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double-precision floating-point numbers</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.11)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>__ballot() (Section B.12)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>__threndence_system() (Section B.5)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Surface functions (Section B.9)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### Legend:

“multiprocessor” stands for Stream Multiprocessor (what we called SM)
Blocks per Grid Heuristics

- # of blocks > # of stream multiprocessors (SMs)
  - If this is violated, then you’ll have idling SMs

- # of blocks / # SMs > 2
  - Multiple blocks can run concurrently on a multiprocessor
  - Blocks that aren’t waiting at a `__syncthreads()` keep the hardware busy
  - Subject to resource availability – registers, shared memory

- # of blocks > 100 to scale to future devices
  - Blocks waiting to be executed in pipeline fashion
  - To be on the safe side, 1000’s of blocks per grid will scale across multiple generations
  - If you have to bend backwards to meet this requirement GPU maybe not right choice
Threads Per Block Heuristics

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
  - Facilitates coalescing

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
  - This all depends on your computation, so experiment via a parametrized kernel
    - Have a loop in which you keep trying multiple execution configurations

- Use the nvvp profiler to understand how many registers you used, what bandwidth you reached, etc.
Occupancy

- In CUDA, executing other warps is the only way to hide latencies and keep the hardware busy

- Occupancy = Number of warps running concurrently on a SM divided by maximum number of warps that can run concurrently
  - When adding up the number of warps, they can belong to different blocks

- Can have up to 48 warps managed by one Fermi SM
  - For 100% occupancy your application should run with 48 warps on an SM

- Many times one can’t get 48 warps going due to hardware constraint
  - See next slide
Test Your Understanding

- Given a compute capability, what is the information needed to compute occupancy?
  - That is, what choices that you make dictate occupancy?
CUDA Optimization: A Balancing Act

- Hardware constraints that might prevent from reaching high occupancy:

  - Number of registers used per thread
    - Fermi: 32K, 4Byte, registers per SM, partitioned among concurrent threads active on the SM
      - Use `--maxrregcount=N` flag on `nvcc`
      - \(N\) = desired maximum registers / kernel
      - At some point "spilling" into local memory may occur
        - Might not be very bad, there is L1 cache that helps to some extent

  - Amount of shared memory (Fermi)
    - 16 or 48 KB per multiprocessor, partitioned among SM concurrent blocks

  - Limit on the number of blocks resident on one SM
    - 8 on Fermi
    - 16 on Kepler
    - 32 on Maxwell
NVIDIA CUDA Occupancy Calculator

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Varying Block Size

Varying Register Count

Varying Shared Memory Usage

Google: “cuda occupancy calculator”
Occupyancy != Performance
[yet a pretty good proxy]

- Increasing occupancy does not necessarily increase performance
  - If you want to read more about this, there is a Volkov paper on class website
  - What controls the damage is the Instruction Level Parallelism (ILP) that the compiler can capitalize on (better efficiency in pipelining, instruction reordering, etc.)

  HOWEVER,

- Low-occupancy multiprocessors are likely to have a hard time when it comes to hiding latency on memory-bound kernels
  - This latency hiding draws on Thread Level Parallelism (TLP); i.e., having enough threads (warps, that is) that are ready for execution
Parameterize Your Application

- Parameterization helps adaptation to different GPUs

- GPUs vary in many ways
  - # of SMs
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Max. threads per block
  - Max. number of warps per SM

- You can even make apps self-tuning (like FFTW and ATLAS)
  - “Experiment” mode discovers and saves optimal configuration
- There are two types of runtime math operations:
  - \texttt{__funcf()}\,: direct mapping to hardware ISA
    - Fast but lower accuracy (see programming guide for details)
    - Examples: \texttt{__sinf(x)}, \texttt{__expf(x)}, \texttt{__powf(x,y)}
  - \texttt{funcf()}\,: compile to multiple instructions
    - Slower but higher accuracy
    - Examples: \texttt{sinf(x)}, \texttt{expf(x)}, \texttt{powf(x,y)}

- The \texttt{-use_fast_math} compiler option forces every \texttt{funcf()}\, to compile to \texttt{__funcf()}
FP Math is Not Associative!
[an aside 2/2]

- In symbolic math, \((x+y)+z == x+(y+z)\)

- This is not necessarily true for floating-point addition

- When you parallelize computations, you likely change the order of operations
  - Round off error propagates differently

- Parallel results may not exactly match sequential results
  - This is not specific to GPU or CUDA – inherent part of parallel execution

- Beyond this associativity issue, there are many other variables (hardware, compiler, optimization settings) that make sequential and parallel computing results be different
CUDA Optimization: Wrap Up…
Performance Optimization

[Wrapping Up…]

- We discussed many rules and ways to write better CUDA code

- The next several slides sort this collection of recommendations based on their importance

- Writing CUDA software is a craft/skill that is learned
  - Just like playing a game well: know the rules and practice
  - A list of high, medium, and low priority recommendations wraps up discussion on CUDA optimization
    - For more details, check the CUDA C Best Practices Guide:

Writing CUDA Software: High-Priority Recommendations

1. To get the maximum benefit from CUDA, focus first on finding ways to parallelize sequential code

2. Use the effective bandwidth of your computation as a metric when measuring performance and optimization benefits

3. Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU

4. Strive to have aligned and coalesced global memory accesses

5. Minimize the use of global memory. Prefer shared memory access where possible (consider tiling as a design solution)

6. Avoid different execution paths within the same warp

Writing CUDA Software: Medium-Priority Recommendations

1. Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts

2. To hide latency arising from register dependencies, maintain sufficient numbers of active warps per multiprocessor (i.e., sufficient occupancy)

3. The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing

4. Use the fast math library whenever speed is very important and you can live with a tiny loss of accuracy

Writing CUDA Software: Low-Priority Recommendations

1. Use constant memory whenever makes sense

2. Use shift operations to avoid expensive division and modulo calculations

3. Avoid automatic conversion of doubles to floats

CUDA Case Study: Parallel Prefix Scan on the GPU
Software Design Exercise: Parallel Prefix Scan

- Vehicle for software design exercise: parallel implementation of prefix sum
  - Serial implementation – assigned as HW early in the semester
  - Parallel implementation: topic of future assignment

- Goal 1: Getting some more exposure to CUDA programming

- Goal 2: Understand that
  - Different algorithmic designs lead to different performance levels
  - Different constraints dominate in different applications and/or design solutions

- Goal 3: Identify design patterns that can result in superior parallel performance
  - Understand that there are patterns and it’s worth being aware of them
  - To a large extend, patterns are shaped up by the underlying hardware
Parallel Prefix Sum (Scan)

- **Definition:**
  The all-prefix-sums operation takes a binary associative operator $\oplus$ with identity $I$, and an array of $n$ elements $[a_0, a_1, ..., a_{n-1}]$ and returns the ordered set $[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$.

- **Example:**
  If $\oplus$ is addition, then scan on the set $[3 1 7 0 4 1 6 3]$ returns the set $[0 3 4 11 11 15 16 22]$.

(From Blelloch, 1990, “Prefix Sums and Their Applications”)
Scan on the CPU

```c
void scan( float* scanned, float* input, int length) {
    scanned[0] = 0;
    for(int i = 1; i < length; ++i) {
        scanned[i] = scanned[i-1] + input[i-1];
    }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
  - Tempted to say that algorithms don’t come more sequential than this…
- Requires exactly $n-1$ adds
Applications of Scan

- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential …
    \[
    \text{out}[0] = f(0) \\
    \text{for}(j=1; j<n; j++) \\
    \quad \text{out}[j] = \text{out}[j-1] + f(j); \\
    \]
  - … into parallel:
    \[
    \forall(i) \in \text{parallel} \\
    \quad \text{temp}[j] = f(j); \\
    \quad \text{scan(out, temp);} \\
    \]

- Useful in implementation of several parallel algorithms:
  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Etc.
Parallel Scan Algorithm: Solution #1
Hillis & Steele (1986)

- Note that an implementation of the algorithm shown in picture requires two buffers of length $n$ (shown is the case $n=8=2^3$)
- Assumption: the number $n$ of elements is a power of 2: $n=2^M$
The Plain English Perspective

- First iteration, I go with stride 1=2^0
  - Start at \(x[2^M]\) and apply this stride to all the array elements before \(x[2^M]\) to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have \(2^M - 2^0\) additions

- Second iteration, I go with stride 2=2^1
  - Start at \(x[2^M]\) and apply this stride to all the array elements before \(x[2^M]\) to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have \(2^M - 2^1\) additions

- Third iteration: I go with stride 4=2^2
  - Start at \(x[2^M]\) and apply this stride to all the array elements before \(x[2^M]\) to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have \(2^M - 2^2\) additions

- … (and so on)
Consider the $k^{th}$ iteration (where $1 < k < M-1$): I go with stride $2^{k-1}$
- Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
  - This means that I have $2^M-2^{k-1}$ additions

... 

$M^{th}$ iteration: I go with stride $2^{M-1}$
- Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
  - This means that I have $2^M-2^{M-1}$ additions

NOTE: There is no ($M+1)^{th}$ iteration since this would automatically put me beyond the bounds of the array (if you apply an offset of $2^M$ to “&x[2^M]” it places you right before the beginning of the array – not good….)
Hillis & Steele Parallel Scan Algorithm

- Algorithm looks like this:

```plaintext
for d := 0 to M-1 do
  forall k in parallel do
    if k - 2^d >= 0 then
      x[out][k] := x[in][k] + x[in][k - 2^d]
    else
      x[out][k] := x[in][k]
  endforall
  swap(in, out)
endfor
```

Double-buffered version of the sum scan
Operation Count
Final Considerations

● The number of operations tally:

\[(2^M-2^0) + (2^M-2^1) + \ldots + (2^M-2^{k-1}) + \ldots + (2^M-2^{M-1})\]

● Final operation count:

\[M \cdot 2^M - (2^0 + \ldots + 2^{M-1}) = M \cdot 2^M - 2^M + 1 = n(\log(n) - 1) + 1\]

● This is an algorithm with \(O(n \cdot \log(n))\) work

● Concluding remarks: is this approach good or not?
  ● Sequential scan algorithm only needs \(n-1\) additions
  ● A factor of \(\log_2(n)\) might hurt: 20x more work for \(10^6\) elements!
    ● Homework requires a scan of about 16 million elements
  ● One more drawback: you need two buffers…
__global__ void scan(float *g_odata, float *g_idata, int n) {
    extern volatile __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;

    // load input into shared memory.
    // Exclusive scan: shift right by one and set first element to 0
    temp[thid] = (thid == 0) ? 0 : g_idata[thid-1];
    __syncthreads();

    for (int offset = 1; offset<n; offset <<= 1) {
        pout = 1 - pout; // swap double buffer indices
        pin = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] = temp[pin*n+thid] + temp[pin*n+thid - offset];
        else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads(); // I need this here before I start next iteration
    }

    g_odata[thid] = temp[pout*n+thid]; // write output
}
The kernel is very simple, which is good.

Note the pin/pout trick that was used to alternate the destination buffer.

The kernel only works when the entire array is processed by one block:
- One block in CUDA has at most 1024 threads
- In this setup we cannot handle yet 16 million entries, which is what your assignment will call for.
A common parallel algorithm pattern:

*Balanced Trees*

- Build a balanced binary tree on the input data and sweep it to the root and then back into the leaves.
- Tree is not an actual data structure, but a concept to determine what each thread does at each step.

For scan:

- Traverse from leaves to root building partial sums at internal nodes in the tree.
  - Root holds sum of all leaves → nice, this is a reduction algorithm.
- Traverse the tree back building the scan from the partial sums.
  - Called down-sweep phase.
for k=0 to M-1
  offset = 2^k
  for j=1 to 2^{M-k-1} in parallel do
    x[j \cdot 2^{k+1}-1] = x[j \cdot 2^{k+1}-1] + x[j \cdot 2^{k+1}-2^{k}-1]
  endfor
endfor
Operation Count, Reduce Phase

By inspection:

\[ \sum_{k=0}^{M-1} 2^{M-k-1} = 2^M - 1 = n - 1 \]

Looks promising…
The Down-Sweep Phase

for \( k = M-1 \) to 0
\[
\text{offset} = 2^k
\]
for \( j = 1 \) to \( 2^{M-k-1} \) in parallel do
\[
\text{dummy} = x[j \cdot 2^{k+1} - 2^{k-1}]
\]
\[
x[j \cdot 2^{k+1} - 2^{k-1}] = x[j \cdot 2^{k+1} - 1]
\]
\[
x[j \cdot 2^{k+1} - 1] = x[j \cdot 2^{k+1} - 1] + \text{dummy}
\]
endfor
endfor

NOTE: This is just a mirror image of the reduction stage. Easy to come up with the indexing scheme…
Down-Sweep Phase, Remarks

- Number of operations for the down-sweep phase:
  - Additions: n-1
  - Swaps: n-1 (each swap shadows an addition)

- Total number of operations associated with this algorithm
  - Additions: 2n-2
  - Swaps: n-1
  - Looks very comparable with the work load in the sequential solution

- The algorithm is convoluted though, it won’t be easy to implement
  - Kernel shown on next slide
__global__ void prescan(float *g_odata, float *g_idata, int n) {
    extern volatile __shared__ float temp[]; // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;
    temp[2*thid]  = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];
    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();
        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset <<= 1; //multiply by 2 implemented as bitwise operation
    }
    if (thid == 0) { temp[n - 1] = 0; } // clear the last element
    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();
        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t   = temp[ai];
            temp[ai]  = temp[bi];
            temp[bi] += t;
        }
        __syncthreads();
        g_odata[2*thid] = temp[2*thid]; // write results to device memory
        g_odata[2*thid+1] = temp[2*thid+1];
    }
}
Upon first invocation of the kernel (kernel #1), each will bring into shared memory 2048 elements:
- 1024 “lead” elements (see vertical arrows ↑ on slide 9), and…
- 1024 mating elements (the blue, oblique, arrows on slide 9)
- Two consecutive “lead” elements are separated by a stride of $k=2^1$
- A “lead” element and its “mating” element are separated by a stride of $k/2=1$

Suppose you take 6 reduction steps in this first kernel and bail out after writing into the global memory the preliminary data that you computed and stored in shared memory

The next kernel invocation should pick up the unfinished business where the previous kernel left…
- Call this a “flawless reentry requirement”
Going Beyond 2048 Entries

Upon the second next kernel call, each block will bring into shared memory 2048 elements:
- 1024 “lead” elements, and...
- 1024 “mating” elements
- Two consecutive “lead” elements will now be separated by a stride of $k=2^6$
- A “lead” element and its “mating” element are separated by a stride of $k/2=2^5$
  - Thus, when brining in data from global memory, you are not going to bring over a contiguous chunk of memory of size 2048, rather you’ll have to jump $2^5$ locations between successive “lead and mating element” pairs
- However, once you bring data in shared memory, you process as before
- Before you exit kernel #2 you have to write back data from shared memory into global memory
  - Again, you have to choreograph this shared to global memory store since there is a $2^5$ stride that comes into play
- If you exit kernel #2 after say 4 more reduction steps, the next time you re-enter the kernel (#3) you will have $k=2^{10}$
Going Beyond 2048 Entries

[3/3]

- You will continue the reduction stage until the stride is $2^{M-1}$
  - At this point you are ready to start the down-sweep phase
  - Down-sweep phase carried out in a similar fashion: we will have to invoke the kernel several times
  - Always work in shared memory and copy back data to global memory before bailing out

- The challenges here are:
  - Understanding the indexing into the global memory to bring data to ShMem
  - How to loop across the data in shared memory

- There are very many shared memory bank conflicts since you move with strides that are power of 2

- Advanced topic: get rid of the bank conflict through padding
Concluding Remarks, Parallel Scan

- Intuitively, the scan operation is not the type of procedure ideally suited for parallel computing
- Even if it doesn’t fit like a glove, leads to nice speedup:

<table>
<thead>
<tr>
<th># elements</th>
<th>CPU Scan (ms)</th>
<th>GPU Scan (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.002231</td>
<td>0.079492</td>
<td>0.03</td>
</tr>
<tr>
<td>32768</td>
<td>0.072663</td>
<td>0.106159</td>
<td>0.68</td>
</tr>
<tr>
<td>65536</td>
<td>0.146326</td>
<td>0.137006</td>
<td>1.07</td>
</tr>
<tr>
<td>131072</td>
<td>0.726429</td>
<td>0.200257</td>
<td>3.63</td>
</tr>
<tr>
<td>262144</td>
<td>1.454742</td>
<td>0.326900</td>
<td>4.45</td>
</tr>
<tr>
<td>524288</td>
<td>2.911067</td>
<td>0.624104</td>
<td>4.66</td>
</tr>
<tr>
<td>1048576</td>
<td>5.900097</td>
<td>1.118091</td>
<td>5.28</td>
</tr>
<tr>
<td>2097152</td>
<td>11.848376</td>
<td>2.099666</td>
<td>5.64</td>
</tr>
<tr>
<td>4194304</td>
<td>23.835931</td>
<td>4.062923</td>
<td>5.87</td>
</tr>
<tr>
<td>8388688</td>
<td>47.390906</td>
<td>7.987311</td>
<td>5.93</td>
</tr>
<tr>
<td>16777216</td>
<td>94.794598</td>
<td>15.854781</td>
<td>5.98</td>
</tr>
</tbody>
</table>

Source: 2007 paper of Harris, Sengupta, Owens
Concluding Remarks, Parallel Scan

- The Hillis-Steele (HS) implementation is simple, but suboptimal

- The Harris-Sengupta-Owen (HSO) solution is convoluted, but $O(n)$ scaling
  - The complexity of the algorithm due to an acute bank-conflict situation

- Finally, we have not solved the problem yet: we only looked at the case when our array has up to 1024 elements
  - You will have to think how to handle the $16,777,216=2^{24}$ elements case
  - Likewise, it would be fantastic if you implement as well the case when the number of elements is not a power of 2