Before We Get Started…

- Last time
  - Tiling as a programming pattern to speed up CUDA code
  - Simple example of finding bugs and improving performance: stencil operation
  - More complex example of optimizing code: vector reduction on the GPU (like your HW)

- Today:
  - Wrap up, vector reduction operation
  - CUDA specific issues that impact GPU computing performance
  - Example: Performing a scan operation

- Miscellaneous
  - CUDA Debugging document uploaded onto the class website:
    - [http://sbel.wisc.edu/Courses/ME964/2013/Lectures/debuggingCUDA.pdf](http://sbel.wisc.edu/Courses/ME964/2013/Lectures/debuggingCUDA.pdf)
  - Exam: Th, November 7, 7:15-9:15 PM (no class on Friday, Nov. 8). Room: 1153ME
    - Review session on Wd, Nov. 6 @ 6 PM in this room (2121ME)
    - Exam will draw on material covered in class and information provided in the primer
    - It’ll be a pen and paper exam. Open book and open anything
Vector Reduction, the Journey

- Step 1: we got something running correctly
- Step 2: got rid of modulo operation and eliminated thread divergence
- Step 3: sequential addressing, no more bank conflicts
- Step 4: each threads does an extra first add during load
- Step 5: only one warp active for last part of the algorithm; loop unrolling
- Step 6: templatize code to generate optimal code for any block dimension
- Step 7: have each thread in a block perform several reductions
Performance Comparison

1: Interleaved Addressing: Divergent Branches
2: Interleaved Addressing: Bank Conflicts
3: Sequential Addressing
4: First add during global load
5: Unroll last warp
6: Completely unroll
7: Multiple elements per thread (max 64 blocks)
Sources of Efficiency Improvement

- Algorithmic optimizations
  - Changes to addressing, algorithm cascading
  - 11.84x speedup, combined

- Code optimizations
  - Loop unrolling
  - 2.54x speedup, combined
Lessons Learned, Vector Reduction

- Understand CUDA performance characteristics
  - Memory coalescing
  - Warp divergence
  - Bank conflicts
  - Loop unrolling

- Use peak performance metrics to guide optimization (peak bandwidth or peak flop rate)

- Know how to identify type of bottleneck
  - E.g. memory, core computation, or instruction overhead

- Use template parameters to generate optimal code

- Understand parallel algorithm complexity theory (we skipped this part, 7th technique)
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>Compute Capability</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>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></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>32 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td>512 KB</td>
<td></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>

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 bend backwards to meet this requirement maybe GPU not the 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
    - Usually still enough registers to compile and invoke successfully
    - This all depends on your computation, so experiment!

- 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.
CUDA Optimization: A Balancing Act

- **Hardware constraints:**
  - Number of registers per kernel
    - 32K per multiprocessor, partitioned among concurrent threads active on the SM
  - Amount of shared memory
    - 16 or 48 KB per multiprocessor, partitioned among SM concurrent blocks

- **Use** `–maxrregcount=N` **flag on nvcc**
  - \( N \) = desired maximum registers / kernel
  - At some point “spilling” into local memory may occur
    - Might not be that bad, there is L1 cache that helps to some extent

- **Recall that you cannot have more than 8 blocks executed by one SM**
CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below (or click here for help).

1. Select Compute Capability (click).
2. Enter your resource usage:
   - Active Threads per Multiprocessor: 512
   - Active Warps per Multiprocessor: 16
   - Active Thread Blocks per Multiprocessor: 4
   - Occupancy of each Multiprocessor: 33%

3. GPU Occupancy Data is displayed here and in the graphs:
   - Multicore Warp Occupancy
   - Varying Block Size
   - Varying Register Count
   - Varying Shared Memory Usage

Physical Limits for GPU Compute Capability:
- Threads per Warp: 32
- Warps per Multiprocessor: 32
- Threads per Multiprocessor: 1024
- Thread Blocks per Multiprocessor: 32
- Total # of 32-bit registers per Multiprocessor: 3072
- Register allocation unit size: 64
- Register allocation granularity: warp
- Shared Memory per Multiprocessor (bytes): 65536
- Shared Memory Allocation unit size: 128
- Warp allocation granularity (for register allocation): 0

Allocation Per Thread Block:
- Warps: 4
- Registers: 8192
- Shared Memory: 12288

These data are used in computing the occupancy data in blue.

Maximum Thread Blocks Per Multiprocessor:
- Limited by Max Warps / Blocks per Multiprocessor: 32
- Limited by Registers per Multiprocessor: 16
- Limited by Shared Memory per Multiprocessor: 128
- Limited by Shared Memory Allocation unit size: 128

CUDA Occupancy Calculator
Version: 2.1

Copyright and License

Click here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Google: “cuda occupancy calculator”
Occupancy != Performance

- Increasing occupancy does not necessarily increase performance
  - If you want to read more about this, there is a Volkov paper on class website
  - What comes to the rescue is the Instruction Level Parallelism (ILP) that becomes an option upon low occupancy

  YET, OFTEN TIMES IS A BAD OMEN …

- 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
Need for This Discussion...

- We discussed at length about what you can do to operate at a memory effective bandwidth close to the nominal bandwidth
  - How to access global memory, bank conflict issues in ShMem, etc.

- Next, discuss for 10 minutes about instruction execution throughput

- In the rare situation you have high arithmetic intensity, it’s good to know how fast math gets executed on the device
Instruction Throughput
[How to Evaluate It]

- Throughputs typically given in number of operations per clock cycle per SM
  - Called “nominal throughput” for reasons explained shortly

- Note that for a warp size of 32, one instruction results in 32 operations
  - You have 32 threads operating in lockstep fashion

- Assume that for a given SM, T is the nominal throughput of operations per clock cycle for a certain math instruction
  - Then, 32 operations; i.e., one math instruction, will take x clock cycles, where
    \[ x = \frac{32}{T} \]

- Concluding: instruction throughput – one instruction every \( \frac{32}{T} \) clock cycles
  - In theory, even better: this number gets multiplied by # of SMs in your GPU

- Quick Remark: the higher the T (operations/clock-cycle), the better
## Nominal Throughputs for Native Arithmetic Instructions

[Operations per Clock Cycle per Multiprocessor]

<table>
<thead>
<tr>
<th>Throughput of Native Arithmetic Instructions</th>
<th>Compute Capability 1.x</th>
<th>Compute Capability 2.0</th>
<th>Compute Capability 2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating-point add, multiply, multiply-add</td>
<td>8</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>64-bit floating-point add, multiply, multiply-add</td>
<td>1</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>32-bit integer add, logical operation</td>
<td>8</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>32-bit integer shift, compare</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32-bit integer multiply, multiply-add, sum of absolute difference</td>
<td>Multiple instructions</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>24-bit integer multiply (__[u]mul24)</td>
<td>8</td>
<td>Multiple instructions</td>
<td>Multiple instructions</td>
</tr>
<tr>
<td>32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (__log2f), base-2 exponential (exp2f), sine (__sinf), cosine (__cosf)</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Type conversions</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
CUDA Instruction Performance

- Instruction performance (per warp), depends on
  - Operand read cycles
  - Nominal instruction throughput
  - Result update cycles

- In other words, instruction performance depends on
  - Nominal instruction throughput
  - Memory latency
  - Memory bandwidth

- “Cycle” refers to the multiprocessor clock rate
  - 1.4 GHz on the GTX480, for example

This is what we just discussed on the previous two slides
There are two types of runtime math operations:

- **__funcf()**: direct mapping to hardware ISA
  - Fast but lower accuracy (see programming guide for details)
  - Examples: __sinf(x), __expf(x), __powf(x,y)
- **funcf()**: compile to multiple instructions
  - Slower but higher accuracy
  - Examples: sinf(x), expf(x), powf(x,y)

- The `-use_fast_math` compiler option forces every `funcf()` to compile to `__funcf()`
FP Math is Not Associative!

- 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 threads 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

Writing CUDA Software:
Medium-Priority Recommendations

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

5. Prefer faster, more specialized math functions over slower, more general ones when possible

Writing CUDA Software: Low-Priority Recommendations

1. For kernels with long argument lists, place some arguments into constant memory to save shared memory.

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

3. Avoid automatic conversion of doubles to floats.

Example: 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: Flexing our CUDA muscles

- 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 $[a_0, a_1, \ldots, a_{n-1}]$ and returns the ordered set $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$.

- 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 ...
    ```
    out[0] = f(0)
    for (j=1; j<n; j++)
      out[j] = out[j-1] + f(j);
    ```
  - ... into parallel:
    ```
    forall(j) in parallel
      temp[j] = f(j);
      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: \( n=2^M \)
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)
The Plain English Perspective

- 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:

```
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

- This scan algorithm is not that work efficient
  - Sequential scan algorithm only needs \(n-1\) additions
  - A factor of \(\log(n)\) might hurt: 20x more work for \(10^6\) elements!
    - Homework requires a scan of about 16 million elements
    - Adding insult to injury: you need two buffers…
__global__ void scan(float *g_odata, float *g_idata, int n) {

    extern __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 swap the buffers.

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