CME213/ME339
Lecture 10

Erich Elsen

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2013

Lecture Outline

- Floating Point and Reductions
- Matrix Vector Product
- Matrix Matrix Multiplication
- Applications of Scan
- Intro to other parallel primitives

Floating Point Properties

- Is commutative $a + b = b + a$
- Is NOT associative $(a + b) + c \neq a + (b + c)$
- Is NOT distributive $a * (b + c) \neq a * b + a * c$
- $a - b = 0$ may not even imply that $a == b$
- Lack of associativity means the order of summation in a reduction matters

Points about Floating Point

Format

	Sign	Exponent	Mantissa
Single	1 bit	8 bits	23 bits
Double	1 bit	11 bits	52 bits

- $\pm Mantissa * 2^{exponent}$
- Sign bit is 0 for positive, 1 for negative
- The Mantissa is *always** in the range $[1, 2)$
- Make the leading 1 implicit, gaining us an extra bit
- Exponent is *biased* by 127 (1023 for double)
- -126 $=$ 1, 0 $=$ 127

Points about Floating Point

Examples

- To represent 192 is single precision floating point:
- It is positive - the sign bit is 0
- It is between 2^7 and 2^8 so the exponent is $7 + 127 = 134$
- $= 1.5 * 2^7$ so the Mantissa is 1.5 (remember the leading 1 is implicit in the binary format)

0	10000110	.10000000000000000000000

Floating Point Addition

- Basic idea is that you align the decimal point and then only get to keep the highest N digits
- In Base 10:

```
 1034.237
+     .0192380
----------------
 1034.246
```

- When we add big numbers to small ones we lose precision
- It is even possible that $a + b == a$

Reductions and Floating Point

Serial Implementation:

```
float sum = 0.f;
for (int i = 0; i < N; ++i)
    sum += vals[i];
```

- sum keeps getting bigger and bigger causing the roundoff error to grow with each subsequent addition
- The error associated with this summation grows is $O(\sqrt{N})$

A Better Way

- By changing the order of the summation, we can do a lot better without doing any more work
- Use a tree!
- Ex: $a_0 + a_1 + a_2 + a_3$

$$
\begin{aligned}
b_0 &= a_0 + a_1 \\
b_1 &= a_2 + a_3 \\
sum &= b_0 + b_1
\end{aligned}
$$

- Terms that are summed tend to be approximately equal in magnitude
- Leads to an error bound of $O(\sqrt{logN})$

Matrix Vector Product

- Naive approach : map one thread to each row and loop over the columns accumulating
- Bad because ... ?

Matrix Vector Product

- Naive approach doesn't expose much parallelism except for very tall matrices
- Naive approach doesn't have coalesced memory access
- Better approach is to map blocks to each row
- More parallelism and coalesced memory access
- How many bytes read
  - Every item in matrix read once
  - input vector is read numRows times
  - output vector is written once
  - (2 * numRows * numCols + numRows) * sizeof(int) bytes

Matrix Vector Product

- Some of the memory reads for the vector x will be from the cache and not global memory
- Our Effective Bandwidth is higher than our Real Bandwidth
- When dealing with caches calculating Real Bandwidth is very difficult, profilers can help
- How would you handle matrices that are very wide and not tall
- Matrices that are very tall but not wide?
- Could we do better by explicitly putting x in shared memory?

Matrix Matrix Multiplication

- C = A * B
- Naive approach of assigning one thread to each output location in C and looping over columns accumulating sum performs very poorly
- Unlike in Matrix Vector product, here entries in the Matrices are used more than once
- Each row in A is read numColsB times
- Use Shared Memory to facilitate memory reuse and reduce trips to global memory
- Break the matrix into blocks that correspond to CUDA blocks and perform blocked Matrix Multiplication

Matrix Matrix Multiplication

- Each 32x32 block is responsible for an output region of the same size in C
- Bring the sub-blocks we need from A and B into shared memory
- Perform local block computations - $2 * 32^3 = 65536$
- Read/Write $3 * 32 * 32 * sizeof(int) = 12288$ bytes
- Flops / byte ratio of 5 - not quite enough math to hide memory latency
- Card peaks flops - 1030 GFlops vs 144 GBytes/sec a ratio of 7

Performance Metrics

- Effective Memory Bandwidth - as if everything was read from global memory
- 2 * numColsA * numRowsA * numColsB * sizeof(float) bytes read
- numRowA * numColsB * sizeof(float) bytes written
- Actual Memory Bandwidth - take into account our use of shared memory
- Determine how many blocks we read / write, multiply by the size of a block

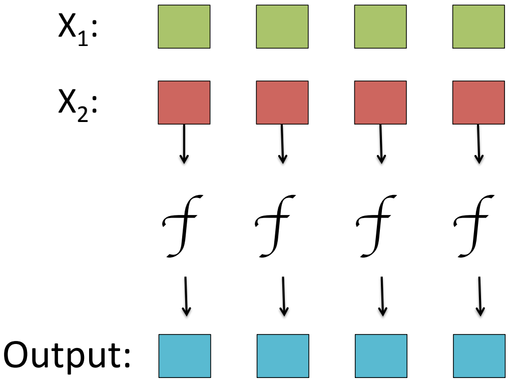
Results

- Flops - well below peak, not the limiting factor
- Effective Memory Bandwidth is quite high - above the peak of the card
- Due to shared memory amplifying our memory bandwidth
- Our Real Memory Bandwidth is well below the peak of the card
- What is limiting performance?
- In this implementation shared memory bandwidth is actually a limiting factor

Parallel Primitives

- Transform / Map
- Reduce
- Scan
- Segmented Reduce
- Segmented Scan
- Sort
- Gather / Scatter
- Upper / Lower Bound

Transform / Map



Transform / Map

- The shift problem from HW2 is easily implemented as a transform
- $f(x) = x + shift$
- Less obvious is that the stencil from HW3 is also a transform
- 1D case: $f(x1, x2, x3) = (x1 - 2 * x2 + x3)/dx^2$
- $transform(in, in + 1, in + 2, N - 2, output)$

Segmented Reduce

We perform multiple reductions using adjacent identical keys to determine which reductions to perform

```
Keys      : 1 3 3 3 2 2 1
Vals      : 9 8 7 6 5 4 3

Out Keys : 1 3  2 1
Out Vals : 9 21 9 3
```

Segmented Scan

We perform multiple scans using adjacent identical keys to determine where to reset sum

```
Keys     : 1 3 3 3  2 2 1
Vals     : 9 8 7 6  5 4 3

Out Keys : 1 3 3 3  2 2 1
Out Vals : 0 0 8 15 0 5 0
```

Gather

```
output[tid] = input[gatherLoc[tid]];
```

```
Values     :  5  4  3  2  1  0
GatherLoc  :  3  2  2  1  4  0

Output     :  2  3  3  4  1  5
```

Scatter

```
output[scatterLoc[tid]] = input[tid];
```

```
Values     :  5  4  3  2  1  0
ScatterLoc :  3  2  2  1  4  0

Output     :  0  2  (3, 4?)  5  1  -
```

Need to be careful with scatter since multiple appearances of the same output location leads to race conditions

Radix Sort

Transform, Scan, Scatter

Sort this sequence: 0 1 1 0 1 0 1

- Idea: Each 0 needs to know how many 0s are before it
- Equivalent to knowing the current position and how many 1s before us
- Each 1 needs to know many 1s are before it and also how many 0s

Scan it: 0 0 1 2 2 3 3 4
Transform: 0 3 4 1 5 2 6
Scatter: 0 0 0 1 1 1 1