Basic GPU Performance 2

CS378 – Spring 2015

Sreepathi Pai
Outline

- Modeling GPU Performance
- Scan Primitives
- SSSP (Bellman-Ford) on the GPU
The Cost of Everything

• What is the ordering of operations based on cost for a GPU?

• Operations are:
  - ALU: Integer, FP (+, *, /, %)
  - Special Function Unit: trig, log, etc.
  - Atomics: to the same address, to different addresses
  - Load/Stores: Global Memory, Shared Memory, Registers, Caches (Texture/Constant/L1/L2)
  - Barriers (__syncthreads)
  - Memory Fences
Latency vs Throughput

- Latency is time taken for an operation
  - A memory access takes 400 cycles
- Throughput is operations per unit time
  - 4.5 TFLOP/s
- Which would you prefer?
  - Low latency, low throughput
  - High latency, low throughput
  - Low latency, high throughput
  - High latency, high throughput
## Operation Throughputs

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>2.0</th>
<th>2.1</th>
<th>3.0, 3.2</th>
<th>3.5, 3.7</th>
<th>5.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating-point add, multiply, multiply-add</td>
<td>32</td>
<td>48</td>
<td>192</td>
<td>192</td>
<td>128</td>
</tr>
<tr>
<td>64-bit floating-point add, multiply, multiply-add</td>
<td>16</td>
<td>4</td>
<td>8</td>
<td>64</td>
<td>1</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>4</td>
<td>8</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>32-bit integer add, extended-precision add, subtract, extended-precision subtract</td>
<td>32</td>
<td>48</td>
<td>160</td>
<td>160</td>
<td>128</td>
</tr>
<tr>
<td>32-bit integer multiply, multiply-add, extended-precision multiply-add</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>Multiple instructions</td>
</tr>
<tr>
<td>24-bit integer multiply (__umulhi24)</td>
<td>Multiple instructions</td>
<td>Multiple instructions</td>
<td>Multiple instructions</td>
<td>Multiple instructions</td>
<td>Multiple instructions</td>
</tr>
<tr>
<td>32-bit integer shift</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>compare, minimum, maximum</td>
<td>32</td>
<td>48</td>
<td>160</td>
<td>160</td>
<td>64</td>
</tr>
<tr>
<td>32-bit integer bit reverse, bit field extract/insert</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>32-bit bitwise AND, OR, XOR</td>
<td>32</td>
<td>48</td>
<td>160</td>
<td>160</td>
<td>128</td>
</tr>
<tr>
<td>count of leading zeros, most significant non-sign bit</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>Multiple instructions</td>
</tr>
<tr>
<td>population count</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>warp shuffle</td>
<td>N/A</td>
<td>N/A</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>sum of absolute difference</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>SIMD video instructions</td>
<td>N/A</td>
<td>N/A</td>
<td>160</td>
<td>160</td>
<td>Multiple instructions</td>
</tr>
<tr>
<td>SIMD video instructions</td>
<td>N/A</td>
<td>N/A</td>
<td>160</td>
<td>160</td>
<td>Multiple instructions</td>
</tr>
</tbody>
</table>

- **Atomics**: 1/9 per clock (Fermi)  
  1 per clock (Kepler)

- **__syncthreads**: 16 per clock  
  128 per clock
Modeling GPU Performance

• Performance Equation
  – Time = Operations / Throughput
  – Throughput = Rate at which operations complete

• Example:
  – Load 144MByte from memory
  – Memory Bandwidth: 144GByte/s
  – Time = 144M/144G = 1ms
Identifying Bottlenecks

• A GPU program:
  – Reads 144M bytes
  – Performs 144M atomic operations
  – Carries out 144M FMADDs

• What is the most likely bottleneck?
  – Reading: \(\frac{144M}{144\text{GBps}} = 1\text{ms}\)
  – Atomics: \(\frac{144M}{1}/745\text{Mhz} = 193\text{ms}\)
  – FMADDs: \(\frac{144M}{192} /745\text{Mhz} = 1\text{ms}\)
Regular Programs: Roofline

Williams et al., Roofline: An Insightful Visual Performance Model for Multicore Architectures
Summar of Performance Modeling

• Cost models of operations can direct the performance effort
  – costs show up as “constant factors” in algorithm implementations

• Throughput models are useful to establish lower bounds on program performance

• Latencies are important, but must usually be discovered through microbenchmarking
The Scan Primitive

- “Fold” – reduce a list of values to a single value
  - \([1 \, 2 \, 0 \, 1 \, 3 \, 5], +\)
  - Result: 12

- “Scan” – reduce a list of values and return intermediate values
  - \([1 \, 2 \, 0 \, 1 \, 3 \, 5], +\)
  - Inclusive Scan: \([1 \, 3 \, 3 \, 4 \, 7 \, 12]\)
  - Exclusive Scan: \([0 \, 1 \, 3 \, 3 \, 4 \, 7]\)

- Also known as: All prefix sums, prefix scan, tree reduction, etc.
Serial Implementation

result[0] = 0;
for(int i = 1; i < N; i++)
    result[i] = result[i - 1] + A[i - 1]

result[0] = A[0];
for(int i = 1; i < N; i++)
    result[i] = result[i - 1] + A[i]
Parallel Implementation

Algorithm 1: The reduce (up-sweep) phase of a work-efficient parallel unsegmented scan algorithm.

1. for $d = 0$ to $\log_2 n - 1$ do
2. for all $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
3. \[x[k + 2^{d+1} - 1] \leftarrow x[k + 2^d - 1] + x[k + 2^{d+1} - 1]\]

Algorithm 2: The down-sweep phase of a work-efficient parallel unsegmented scan algorithm.

1. $x[n - 1] \leftarrow 0$
2. for $d = \log_2 n - 1$ down to 0 do
3. for all $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
4. \[t \leftarrow x[k + 2^d - 1]\]
5. \[x[k + 2^d - 1] \leftarrow x[k + 2^{d+1} - 1]\]
6. \[x[k + 2^{d+1} - 1] \leftarrow t + x[k + 2^{d+1} - 1]\]
GPU Implementation Sketch

- Where is the array stored?
- What is the synchronization mechanism between levels?
GPU Scan Implementations

- Very large arrays
  - Store array in global memory
  - Synchronize using multiple kernel calls
- Arrays that fit in shared memory
  - Store array in shared memory
  - Synchronize using __syncthreads()
- Arrays smaller than warpsize
  - Use *warp collective* instructions
- Use NVIDIA CUB!
The Many Uses of Scan

- Stream compaction/filtering
  - When you want to filter an array to another array
- Radix sort
- Many more ...
Reducing the Cost of Atomics

• Assume you have a worklist which is implemented as follows:

```c
int worklist[1024];
int tail = 0;

void push(int item) {
    worklist[tail++] = item;
}

void push_parallel(int item) {
    old_tail = atomicAdd(&tail, 1);
    worklist[old_tail] = item;
}
```

Wrong Parallel Implementation!
Client Code

1 Thread
for(int i = 0; i < n; i++)
    push_parallel(work[i]);

1 Thread
old_tail = atomicAdd(tail, n)
for(int i = 0; i < n; i++)
    worklist[old_tail + i] = work[i];

T Threads, each n items, n is same for every thread
__shared__ old_tail;
if(tid == 0) old_tail = atomicAdd(tail, n*T)
__syncthreads();
for(int i = 0; i < n; i++)
    worklist[old_tail + n*tid + i] = work[i];
Client Code using Scan

T Threads, each n items, n may be different for each thread
__shared__ old_tail;

if(tid == 0) old_tail = atomicAdd(tail, _)
__syncthreads();

for(int i = 0; i < n; i++)
    worklist[old_tail + _ + i] = work[i];

T Threads, each n items, n may be different for each thread
__shared__ old_tail;
int offset;

ExclusiveScan(n, offset, total);

if(tid == 0) old_tail = atomicAdd(tail, total)
__syncthreads();

for(int i = 0; i < n; i++)
    worklist[old_tail + offset + i] = work[i];
Results

Fermi

Kepler
__global__ void sssp_kernel(CSRGraph graph, int * ret_val)
{
    unsigned tid = TID_1D;
    unsigned nthreads = TOTAL_THREADS_1D;

    index_type node_end;
    node_end = (graph).nnodes;
    for (index_type node = 0 + tid; node < node_end; node += nthreads)
    {
        index_type edge_end;
        if(graph.node_data[node] == INF) continue;

        edge_end = (graph).getFirstEdge((node) + 1);
        for (index_type edge = (graph).getFirstEdge(node) + 0; edge < edge_end; edge += 1)
        {
            index_type dst = graph.getAbsDestination(edge);
            edge_data_type wt = graph.getAbsWeight(edge);

            old_value = atomicMin(graph.node_data + dst, graph.node_data[node] + wt);
            if(old_value > new_value) *ret_val = 1;
        }
    }
}
Reducing Atomics

old_value = atomicMin(graph.node_data + dst, graph.node_data[node] + wt);
if(old_value > new_value) *ret_val = 1;

Is the atomic required?
Can we use scan primitives?

if(graph.node_data[dst] > graph.node_data[node] + wt) {
    atomicMin(graph.node_data + dst, graph.node_data[node] + wt);
    *ret_val = 1;
}

Reduced $n$ unconditional atomics to $n$ conditional atomics

Is this the best we can do?
Using Textures

__global__ void sssp_kernel(CSRGraph graph, int * ret_val)
{
    unsigned tid = TID_1D;
    unsigned nthreads = TOTAL_THREADS_1D;

    index_type node_end;
    node_end = (graph).nnodes;
    for (index_type node = 0 + tid; node < node_end; node += nthreads)
    {
        index_type edge_end;
        if(graph.node_data[node] == INF) continue;

        edge_end = (graph).getFirstEdge((node) + 1);
        for (index_type edge = (graph).getFirstEdge(node) + 0;
             edge < edge_end; edge += 1)
        {
            index_type dst = graph.getAbsDestination(edge);
            edge_data_type wt = graph.getAbsWeight(edge);

            old_value = atomicMin(graph.node_data + dst, graph.node_data[node] + wt);
            if(old_value > new_value) *ret_val = 1;
        }
    }
}
CSR Graph Implementation

row_start[0..V+1] = [a b c d e f]  row_start[0..V+1] = [T0 T1 T2 T3 T4 T5]
edge_dst[0..E]   = [a1 a2 b1 c1 d1 e1 e2]  edge_dst[0..E]   = [T0 T0 T1 T2 T3 T4 T4]
node_data[0..V] = [n0 n1 n2 n3 n4 n5]  node_data[0..V] = [T0 T1 T2 T3 T4 T5] (read)
edge_data[0..E] = [E1 E2 E3 E4 E5 E6 E7]  node_data[0..V] = [T? T? T? T? T? T?] (write)


Which of these arrays can be accessed through the texture cache?

Would mapping some of these arrays to textures help?

Can node_data be mapped to a texture?
Correcting Load Imbalance

row_start[0..V+1] = [a b c d e f]  
edge_dst[0..E] = [a1 a2 b1 c1 d1 e1 e2]

row_start[0..V+1] = [T0 T1 T2 T3 T4 T5]  
edge_dst[0..E] = [T0 T0 T1 T2 T3 T4 T4]

for (index_type node = 0 + tid; node < node_end; node += nthreads) {
    index_type edge_end;
    if (graph.node_data[node] == INF) continue;
    edge_end = (graph).getFirstEdge((node) + 1);

    for (index_type edge = (graph).getFirstEdge(node) + 0; 
         edge < edge_end; edge += 1) {

What happens if some nodes have very high degrees?
Can we use an alternate schedule?

node_end = node_end * max_degree;

for (index_type node_ = 0 + tid; node_ < node_end; node_ += nthreads)
{
    index_type node = node_ / max_degree;
    index_type offset = node_ % max_degree;

    index_type edge_end;
    if(graph.node_data[node] == INF) continue;

    edge_end = (graph).getFirstEdge((node) + 1);

    index_type edge = (graph).getFirstEdge(node) + offset;
    if(edge < edge_end) {
        ...

This maps edges to individual threads – no load imbalance?
Recall Static Scheduling

- Develop two versions
- Examine max_degree of graph
- Invoke one of:
  - node per thread
  - edge per thread
- Depending on max_degree
- Does this work?
Dynamic Scheduling

- Change assignment of work to threads at runtime
- Uses scan primitive for communication
  - what else?
- Basic idea:
  - start out as node/thread
  - identify high degree nodes
  - process them using one thread per node \textit{dynamically}
Implementation - 1

```cpp
for (index_type node = 0 + tid; node < node_end; node_ += nthreads)
{
    index_type edge_end;
    if(graph.node_data[node] == INF) continue;

    edge_end = (graph).getFirstEdge((node) + 1);
    index_type edge = (graph).getFirstEdge(node);
    int degree = edge_end - edge;

    if(degree > 256) {
        Dynamic Schedule
    }

    if(degree < 256) {
        for(...)
    }
```
Dynamic Schedule

• How many threads have high-degree nodes?
• Which threads have high degree nodes?
• Which high-degree node should we process now?
Can we use an alternate schedule?

node_end = node_end * max_degree;

for (index_type node_ = 0 + tid; node_ < node_end; node_ += nthreads) {
    index_type node = node_ / max_degree;
    index_type offset = node_ % max_degree;

    index_type edge_end;
    if(graph.node_data[node] == INF) continue;

    edge_end = (graph).getFirstEdge((node) + 1);

    index_type edge = (graph).getFirstEdge(node) + offset;

    if(edge < edge_end) {
        ...
    }

This maps edges to individual threads – no load imbalance?
Implementation - 2

BlockScan(temp_storage).ExclusiveSum(degree > 256, offset, total);

for (int _np_long = 0; _np_long < total; _np_long++)
{
    if (degree > 256 && _np_long == offset)
    {
        _np_wb_start = edge;
        _np_wb_size = degree;
    }

    __syncthreads();
}

int ns = _np_wb_start;
int ne = _np_wb_size;

for (int _np_j = threadIdx.x; _np_j < ne; _np_j += blockDim.x)
{
    index_type edge;
    edge = ns + _np_j;
    
    index_type dst;
    dst = graph.getAbsDestination(edge);
    ...
}

How many threads have high degree nodes?
Which high degree node should we all process?
One edge per thread
Conclusion

- GPU programs can take advantage of several parallel programming primitives
  - scan, in particular
- Novel ways to reduce costs using such collective operations
- Dynamic scheduling may be required