Histgrams

Privatized for Fast, Level Performance

Nicholas Wilt
Author, The CUDA Handbook
@CUDAHandbook
http://www.facebook.com/cudahandbook
What Is A Histogram?

• Probability distribution
• $k$ categories and $N$ data elements
• Often represented by array of $k$ integers
• Many statistics can be inferred from the histogram
  – Min, max, mean, median
• Also a building block (e.g. Radix Sort)
Example Histogram

Histogram (coins.pgm)
void
hist1DCPU(
    unsigned int pHist[256],
    const unsigned char *p, size_t N )
{
    for ( size_t i = 0; i < N; i++ ) {
        pHist[ p[i] ] += 1;
    }
}
Naïve CUDA Code

- One Histogram In Global Memory
  - Use atomic add for correctness

```c
__global__ void
histogram1DPerGrid(
    unsigned int *pHist,
    const unsigned char *p, size_t N )
{
    for ( size_t i = blockIdx.x*blockDim.x+threadIdx.x; 
         i < N; 
         i += blockDim.x*gridDim.x ) {
        atomicAdd( &pHist[ p[i] ], 1 );
    }
}
```
## Performance

<table>
<thead>
<tr>
<th>Chip</th>
<th>Speed (Mpix/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla (GeForce GTX 280)</td>
<td>58</td>
</tr>
<tr>
<td>Fermi (M2050)</td>
<td>1530</td>
</tr>
<tr>
<td>Kepler (GeForce GTX 680)</td>
<td>10720</td>
</tr>
</tbody>
</table>

(256 possible pixels in input)
## Contention

<table>
<thead>
<tr>
<th>Values</th>
<th>Tesla</th>
<th>Fermi</th>
<th>Kepler</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>58</td>
<td>1530</td>
<td>10720</td>
</tr>
<tr>
<td>128</td>
<td>39</td>
<td>969</td>
<td>7074</td>
</tr>
<tr>
<td>64</td>
<td>28</td>
<td>660</td>
<td>4734</td>
</tr>
<tr>
<td>32</td>
<td>26</td>
<td>749</td>
<td>3058</td>
</tr>
<tr>
<td>16</td>
<td>22</td>
<td>557</td>
<td>3767</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>349</td>
<td>4422</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>210</td>
<td>2864</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>121</td>
<td>1725</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>68</td>
<td>988</td>
</tr>
</tbody>
</table>
Contestation (Visual)
Anti-Contention Strategies

• More histogram arrays!

• Per Block
  – Faster increments (shared memory)
  – Increments still have to be atomic

• Minuses:
  – Have to reduce histograms into final output
  – Threads within block can still contend
__global__ void 
histogram1DPerBlock(
    unsigned int *pHist,
    const unsigned char *base, size_t N)
{
    __shared__ int sHist[256];
    for (int i = threadIdx.x; i < 256; i += blockDim.x) {
        sHist[i] = 0;
    }
    __syncthreads();
    for (int i = blockIdx.x*blockDim.x+threadIdx.x; i < N; 
        i += blockDim.x*gridDim.x ) {
        atomicAdd( &sHist[ base[i] ], 1 );
    }
    __syncthreads();
    for (int i = threadIdx.x; i < 256; i += blockDim.x) {
        atomicAdd( &pHist[i], sHist[ i ] );
    }
}
Per-Thread Histograms

- Must use shared memory (addressible)
- 1 byte per element
- 64 threads/block = 16K
- Can fit 3 blocks per SM = 192 threads

- Many different layout options
Histogram Per Row?

Problem: For degenerate case, \textit{de facto} contention due to bank conflicts. And we cannot spend *any* shared memory on padding!
Histogram Per Column?

histIndex = blockDim.x*pixval+threadIdx.x;

Problem: Still prone to bank conflicts (every 4 threads contending for the same 32-bit value in shared memory)
Hybrid Scheme

- 32-bit elements *by column*

Degenerate case exhibits optimal shared memory behavior
32-Bit Increments

- Shared Memory Optimized for 32-bit Accesses

Rewrite:

```c
((unsigned char *) pHist)[i] += 1;
```

As:

```c
((unsigned int *) pHist)[i>>2] += 1<<(i&3)*8);
```

Fun fact: Kepler compiler translates to byte permute
Resulting Code

• No slower than previous 32-bit increment

\[(\text{unsigned int *} p\text{Hist})[i\gg2] += 1\ll((i\&3)*8);\]

```c
inline __device__ void
incPacked32Element( unsigned char pixval )
{
    extern __shared__ unsigned int privHist[];
    const int blockDimx = 64;
    unsigned int increment = 1\ll8*(pixval\&3);
    int index = pixval\gg2;
    privHist[index*blockDimx+threadIdx.x] += increment;
}
```
Gathering Histograms

• Privatized histograms are great! but...
• Now we have 64 histograms per block
  – And multiple blocks→many histograms to reduce.
• And they only contain 8-bit elements
  – need to be gathered frequently to avoid overflow

• Performance of this operation surprisingly important!
template<bool bClear>
  __device__ void
merge64HistogramsToOutput( unsigned int *pHist )
{
  extern __shared__ unsigned int privHist[];

  unsigned int sum02 = 0;
  unsigned int sum13 = 0;
  for ( int i = 0; i < 64; i++ ) {
    int index = (i+threadIdx.x)&63;
    unsigned int myValue = privHist[threadIdx.x*64+index];
    if ( bClear ) privHist[threadIdx.x*64+index] = 0;
    sum02 += myValue & 0xff00ff;
    myValue >>= 8;
    sum13 += myValue & 0xff00ff;
  }
  atomicAdd( &pHist[threadIdx.x*4+0], sum02&0xffff );
  sum02 >>= 16;
  atomicAdd( &pHist[threadIdx.x*4+2], sum02 );
  atomicAdd( &pHist[threadIdx.x*4+1], sum13&0xffff );
  sum13 >>= 16;
  atomicAdd( &pHist[threadIdx.x*4+3], sum13 );
}
Result: Level Performance

Fermi (GF100)  
Kepler (GK104)
CPU Comparison (Haswell)

- Per-thread privatized histograms
- 2 GB/s/core, and very level
- GF100 is only 9GB/s, GK104 is only 6.6 GB/s

- So this is a workload where GPUs don’t “pwn” CPUs. Best done on data already in the GPU.
Epilogue

• To improve performance of this and similar workloads, NVIDIA could add native hardware support for shared memory reductions. –November 2013

• Maxwell
  – Hardware support for smem atomics!
    • Lower-latency increments for per-block formulation!
  – 64K shared memory (not shared with L1)!
    • More occupancy!
Maxwell Performance

![Graph showing Maxwell Performance]

- **Per Grid**
- **Per Block**
- **Per Thread**

The graph illustrates the performance of Maxwell in terms of Mpix/s against the number of possible values. The performance decreases significantly as the number of possible values increases.
Per-Block Performance

Mpix/s vs # possible values for different SM versions.
Questions?

• Twitter handle: @CUDAHandbook
• http://www.facebook.com/cudahandbook