Parallel Histogram

Mark Greenstreet and Ian M. Mitchell

CpSc 418 – March 28, 2018

Table of Contents

Outline

Histograms

Given

- A set of data values $\{x_i\}_{i=0}^{n-1}$.
- A set of bins $\{b_j\}_{j=0}^{m-1}$.
  - Often the bins are intervals; for example, $b_j = \{x \mid \ell_j \leq x < u_j\}$.

Then a histogram of the data is a set of counts:

$$h_j = \text{count}\left(\{x_i \mid x_i \in b_j\}\right)$$

- Histograms visualize the distribution of the data values.
- Typically displayed as a 2D chart using bars
  - Horizontal position represents b_j
  - Vertical height is proportional to h_j.
- Some sources distinguish between “histograms” (for continuous data) and “bar charts” (for discrete or categorical data).

We will focus on computing the $\{h_j\}_{j=0}^{m-1}$.

Serial Histogram

```
hist_cpu(float *x, uint n, float *l, float *u, float *h, uint m)
  for(uint i = 0; i < n; i++) {
    uint j = find_bin(x[i], l, u, m);
    h[j]++;
  }
}
```

Compare with K&H figure 9.2 for alphabetical data.

Computational cost per element depends on `find_bin()`:

- Constant time: Simple arithmetic to find `j` given `x[i]`; for example, equally spaced bins.
- Log time: Use binary search for non-equally spaced bins.
- Linear time: Check inclusion one at a time for bins with no particular ordering.

What is serial computation cost?

Outline

Parallelization

Basic GPU implementation: Divide the input data among the threads:

```
hist_kernel(float *x, uint n, float *l, float *u, float *h, uint
  uint i = blockIdx.x * blockDim.x + threadIdx.x;
  uint j = find_bin(x[i], l, u, m);
  h[j]++;
}
```

What is the CGMA?

- In practice we typically assign multiple input elements to each thread (see K&H figure 9.6).
  - Be careful to arrange assignment of elements to allow coalescing of global memory reads (K&H figures 9.7 and 9.8).
- Is low CGMA the problem?

The Promise and the Curse of Shared Memory

Multiple threads may wish to update the count in a single histogram bin all at the same time.

Use “atomic addition” to avoid a race condition:

- Implemented by calling the “intrinsic” `atomicAdd()`.
  - Looks like a regular C function call.
  - Compiler converts it into a specialized instruction.
- Locks a memory location so that the read, addition and write occur “atomically”.
  - No other thread is allowed access to the memory location until all three steps are complete.
- This operation always has a long latency, and it can be *very* long.
  - Latency is the sum of read, add and write latencies, including any DRAM access delay.
  - If multiple threads issue atomic additions to the same memory location, they are serialized.
  - See K&H figure 9.9.

Improving Histogram Efficiency

Key: Reduce frequency and cost of bin collisions.

- Reduce cost: Resolve atomic additions in the cache.
  - Supported transparently in L2 cache on all but earliest GPUs.
- Reduce frequency: Aggregate multiple increments to the same bin into a single atomic addition.
  - Helps if a single thread is likely to get many consecutive elements in the same bin.
  - Also reduces memory traffic.
  - Will lead to increase in thread divergence.
- Reduce cost and frequency: Each block keeps its own private copy of all of the histogram bins in its shared memory.
  - Reduces cost because shared memory is fast.
  - Reduces frequency because fewer threads are sharing the bins.
  - Need to sum the counts in the separate copies of the same bins at the end. See K&H figure 9.10 for one solution.