EECS 468: Lab 4 Write-up

Scott Young, Kapil Garg

Near the top of "scan_largearray.cu", set #define DEFAULT_NUM_ELEMENTS to 16777216. Set #define MAX_RAND to 3. Record the performance results when run without arguments, including the host CPU and GPU processing times and the speedup.

Sample	CPU Processing Time (ms)	GPU Processing Time (ms)	Speedup (1 / (T_{GPU} / T_{CPU}))
1	46.205002	7.196000	6.420928
2	46.272999	7.193000	6.433060
3	47.578999	7.193000	6.614625
4	46.338001	7.193000	6.442097
5	46.230000	7.193000	6.427082
6	46.872002	7.195000	6.514524
7	46.191002	7.194000	6.420768
8	46.316002	7.195000	6.437248
9	46.275002	7.196000	6.430656
10	46.244999	7.192000	6.430061

Average CPU Processing Time: 46.4524008 ms
Average GPU Processing Time: 7.194000 ms
Average Speedup: 6.4571049 times

CPU Processing Time Variance: 0.195849898 ms
GPU Processing Time Variance: 0.000002 ms
Speedup Variance: 0.027225863 ms

Describe how you handled arrays not a power of two in size, and how you minimized shared memory bank conflicts. Also describe any other performance-enhancing optimizations you added.

To handle arrays that are not a power of two in size, we pad the number of elements to the next power of 2 of elements and then set 0's as values for all entries that do not have any data. Then, once all the given values are added, the padded 0s will not affect the total sum. This allows us to use our general purpose kernel instead of having to handle the issue with special cases, but we found that having a separate kernel that does not need to do the check for power of 2 makes computation on power of 2 arrays slightly faster.

To minimize bank conflicts, we apply a variable amount of offset to all operations interacting with shared memory such that no two operations will concurrently hit the same bank for a read or write.

We used a block size of 256 since 256^3 = 16,777,216. Beyond this point, both the CPU host and GPU algorithms suffer from single point floating precision loss and can no longer be evaluated consistently. We additionally create a separate kernel that handles padding for non-power of two so that for cases that are power of two, we can save computation time when checking to see if padding needs to be done.

How do the measured FLOPS rate for the CPU and GPU kernels compare with each other, and with the theoretical performance limits of each architecture? For your GPU implementation, discuss what bottlenecks your code is likely bound by, limiting higher performance.

For the analysis below, we will use n = 16,777,216 elements.

Host Code on CPU
Algorithm complexity: O(n)

Each element requires 1 floating point addition to the final sum. Thus, we have 16777216 floating point operations.

$$\frac{16777216\ Floating\ Point\ Additions}{46.3002\ \ ms} = 362357311.631483\ FLOPS = 0.362357312\ GFLOPS$$

The theoretical performance of the Intel(R) Xeon(R) CPU E5-1620 is 115.2 GFLOPS[1]. The major bottleneck for the CPU host code is the constant read/write to memory.

GPU code
Algorithm complexity: O(n) + O(n)

A binary tree with *n* leaves has *d* = $\log_2 n$ levels, and each level *d* has 2^d nodes. If we perform one add per node, then we will perform *O*(*n*) adds on a single traversal of the tree. Since n = 16777216, this results in 16777216 floating point additions. We additionally apply an addition kernel that has a complexity of O(n), resulting in 16777216 floating point additions. Therefore, our GPU code makes a total of 16777216 * 2 = 33,554,432 floating point additions to for n = 16777216.

$$\frac{2 * 16777216\ Floating\ Point\ Additions}{7.5662\ ms} = 4434779942.3753\ FLOPS = 4.434779942\ GFLOPS$$

The theoretical performance of the GeForce GTX 680 is 3090 GFLOPS.[2]

The major bottleneck for the GPU code is the number of blocks and threads that can be run concurrently. We are limited to 2048 threads per multiprocessor at 16 multiprocessors. At a block size of 256 threads, we can only run 2048 / 256 * 16 = 128 blocks at once, while we have 16777216 / 256 = 65536 blocks that need to be processed. By adding more multiprocessors or more threads per multiprocessor, it is possible to achieve greater performance.

Additionally, we currently run one kernel at a time, greatly reducing our total potential output. However, due to the fact that the number of concurrent blocks is the main bottleneck, the effect is minimized.

---

[1] http://www.intel.com/content/dam/support/us/en/documents/processors/xeon/sb/xeon_E5-1600.pdf
[2] https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/