Assignment 3

William Wang ww7964

March 10, 2020

1 Approach

In this lab we implemented sequential and parallel versions of the k-means clustering algorithm. Our sequential implementation primarily used the C++ STL, while our parallel implementations ran on a GPU by means of the Thrust and CUDA APIs. The parallel programs were further broken down into separate Thrust-based, CUDA-based, and CUDA-based shared-memory versions.

1.1 Sequential Algorithm

The k-means clustering algorithm we implemented roughly executes the following procedure with random initial centroids:

1. Label each point with its nearest centroid
2. Update each centroid with the average of all its points
3. Check if centroids have converged

To this end, our sequential algorithm was fairly straightforward, performing these steps serially within a while loop.

Initially, we attempted to check centroid convergence by comparing the magnitude of the difference vector between each centroid's new and old values against some fixed ϵ. We learned, however, that this method can perform badly at scale, as it can demand arbitrarily small element-wise errors as the dimensionality of the space increases. Our solution to this issue was to just do an element-wise comparison, asserting convergence when the differences among all respective elements for all new/old centroid pairs was within ϵ. This approach also exposed additional parallelism in that all the differences could be computed independently, as opposed to the atomic sum of squares required in our first attempt.

1.2 Thrust

Our Thrust implementation took the same steps as our sequential algorithm, but doing each step in parallel. To maximize parallelism, we attempted to use as few intermediate synchronization points as possible in each iteration of the loop. We accomplished this by only having one such point, namely the point when all the point counts are known for each centroid. We must know this value to compute the new centroids, and in the process of finding this value, we can accumulate all the other information we need as shown below. This was implemented in Thrust using Thrust vectors and a variety of Thrust primitives and iterators.

- **for each** point, do in parallel:
  - **reduce** centroids on that point to find the closest one
  - label point with closest centroid
  - increment closest centroid's count
  - atomically add to centroid's point accumulator
- *synchronize*
- **for each** centroid, do in parallel:
  - **transform** centroid to average of accumulated points
- *synchronize*

1.3 CUDA

Our CUDA program followed exactly the same parallelism framework as our Thrust program. Everywhere we needed to invoke a top-level Thrust primitive, we spawned a kernel instead to partition the work. Our kernel parameters were determined by a combination of hardware features and experimentation guided by these features, described in more detail in the Analysis section.

1.4 Shared Memory

There were five types of high-usage data that could potentially be stored in shared memory: features (points), point labels, centroids, centroid counts, and accumulated coordinate totals. We observed that the first two of these were not accessed very frequently in comparison to how many of them they were - roughly once per memory location. They would also not fit in shared memory at once, so we focused instead on sharing the centroid-based data. These were used with relatively high "density", with each memory location being accessed many times. We ended up loading the centroids themselves into shared memory, although there was plenty of room for the other datasets, which we would have included as well with more time.

2 Analysis

We tested and profiled our programs on an Ubuntu 18.04 machine with an 8-core CPU and a 2560-core, 20-SM GPU with 48K each of L1 cache and shared memory per SM. Each program - sequential, Thrust, CUDA, shared memory - was executed 20 times on inputs of 2048, 16384, and 65536 points with 16, 24, and 32 dimensions respectively. The programs were instructed to produce 16 centroids in all cases. The below results (Figure 1) are of the average running times within each set of 20 trials, with speedups being relative to the sequential implementation.

Before we could generate this data, we had to determine the number of blocks and thread per block to spawn. We obtained some initial estimates by brute-force searching the parameter space on the 64K-point dataset. This informed us that performance increased up to around `BLOCKS` = 16 and `THREADS/BLOCK` = 256, past which speedup slowly decreased. This more or less corresponded with the properties of the hardware described above; thus, we used these parameters to benchmark our GPU tests. (With more time, we would have implemented dynamic computation of kernel parameters using the CUDA Occupancy API.)

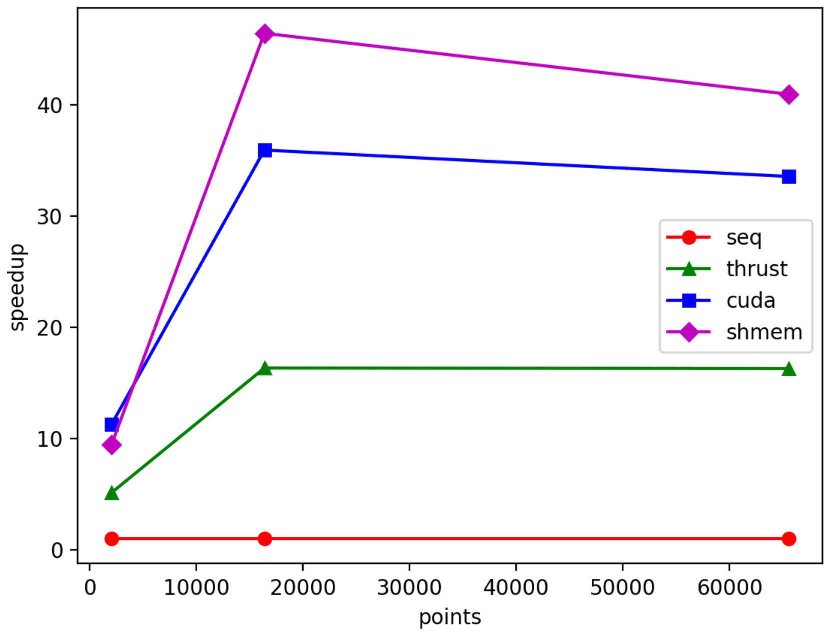

Figure 1: Speedup vs. number of points

In each parallel version, we see a reasonable speedup for small input sizes, which rapidly approaches the maximum observed speedup as input size increases. This is likely due to occupancy getting saturated fairly early on due to the largeness of the input with respect to the amount of hardware resources, although more experimentation is needed to confirm this. Additionally, we observe that the shared-memory implementation is the fastest for large numbers of points, and only slightly slower than the basic CUDA implementation for smaller datasets. This is exactly what we expected, as this version gave us the greatest amount of freedom for exposing parallelism. Conversely, we had to make many contortions to fit our algorithm into the high-level Thrust API, causing it to be the slowest of the parallel implementations. With 2560 CUDA cores on the GPU, we did not see speedup anywhere close to the absolute upper bound; however, this is totally reasonable as inordinate amounts of time are needed to communicate with the device, which we will soon see. Even without this, we didn't make any extreme efforts to manually load balance, reduce synchronization overhead, etc., all of which can and will decrease speedup.

We used `nvprof` to profile our programs, which showed us just how much time was being spent moving data around. The only significant offender was `cudaMemcpyHostToDevice`, which accounted for 30% of runtime in the plain CUDA implementation and 40% in shared memory. This makes sense, as `cudaMemcpy` performance was not improved by our shared memory optimizations and would therefore take a larger proportion of the time there.

Time spent: about 20 hours

References

- NVIDIA Developer Blog https://devblogs.nvidia.com
- NVIDIA Toolkit Documentation https://developer.nvidia.com