CS380P: Parallel Systems

Lab #2: KMeans with CUDA

Submission Guide

In this lab, you will experiment with different implementations of the Kmeans algorithm.

Among the files you have to tar and submit, one file required by our autograder is the `submit` file. It does **not** have an extension (e.g. .txt), so just call it `submit`

This file contains instructions on how to compile each variant of your code, where the binary will be located and, if your code requires, any extra arguments

A template submit file is provided. Please read the instructions carefully because the autograder relies on this submit file to grade your implementations.

Do not rename the submit file.

As an example, the Sequential part of your submit file could look something like this, assuming you are compiling the binaries in a folder called bin and you are going to have one binary per implementation (one for cpu, one for CUDA, etc.):

```
[Sequential]
How_To_Compile: gcc main.c kmeans_cpu.c -O2 -o bin/kmeans_cpu
Executable: bin/kmeans_cpu
Extra_Args:
```

You could also have just one application with all implementations, and switch which should execute using arguments (which you should implement yourself, feel free to reuse code from lab 1), for example:

```
[Sequential]
How_To_Compile: nvcc main.cpp kmeans_cpu.cpp kmeans_cuda.cu kmeans_thrust.cpp -O2 -o kmeans
Executable: kmeans
Extra_Args: --use_cpu
```

1. Inputs

In this lab, you can start to test your Kmeans implementations with the following 3 sets of multi-dimensional points. The dimensions of the points in each set are known. However, your program should accept the dimension of points as one of the command line arguments.

random-n2048-d16-c16.txt

random-n16384-d24-c16.txt

random-n65536-d32-c16.txt

Note that the autograder will use different inputs (different dims and number of points) to test your program. However, sample solutions for the above inputs can be found at the links below. Note that concerns like floating point associativity will mean that your answers can be different, so checking correctness requires verifying that answers are within a small epsilon (e.g. 10^-4) in each dimension of each point. Also, your answer need not produce the same order of centers or IDs, so again, make sure your own correctness checking does not assume a particular order of centers in the output. These outputs were generated with the seed: `-s 8675309` (see the section about Random Initial Centroids Generation below).

random-n2048-d16-c16-answer.txt

random-n16384-d24-c16-answer.txt

random-n65536-d32-c16-answer.txt

Students are encouraged to share newly generated inputs and their respective solutions, calculated by their implementation of kmeans, on Piazza for other students to compare. Just don't share code.

1.1 CmdLine Arguments

For all your implementations, your program should at least accept the following arguments

- `-k num_cluster`: an integer specifying the number of clusters
- `-d dims`: an integer specifying the dimension of the points
- `-i inputfilename`: a string specifying the input filename
- `-m max_num_iter`: an integer specifying the maximum number of iterations
- `-t threshold`: a double specifying the threshold for convergence test.
- `-c`: a flag to control the output of your program. If `-c` is specified, your program should output the centroids of all clusters. If `-c` is not specified, your program should output the labels of all points. See details below.
- `-s seed`: an integer specifying the seed for rand(). This is used by the autograder to simplify the correctness checking process. See details below.

Your program can also accept more arguments to control the behavior of your program for different implementations. These extra arguments can be specified in the `submit` file. Refer to the instruction in `submit` file for more details.

`-k`, `-d`, and `-i` should depend on the input files. The max number of iterations `-m` is used to prevent your program from an infinite loop if something goes wrong. In general, your implementation is expected to converge within 150 iterations. Therefore, an value of 150 to `-m` should be good enough. Depending on your methods for convergence test, you might want to use different thresholds `-t` for different implementations. As a reference, the threshold for comparing centers without any non-determinism issues can be as small as `10^-10`. However, for comparing centers with non-determinism issues, you might want to use a threshold of `10^-6`. The autograder will specify `-t 10^-5` for all implementations.

2. Output

For each input, each implementation is asked to classify the points into `k` clusters. You should measure the elapsed time and total number of iterations for Kmeans to converge. The averaged elapsed time per iteration and the number of iterations to converge should be written to STDOUT in the following form

```
printf("%d,%lf\n", iter_to_converge, time_per_iter_in_ms)
```

Note that the time should be measured in milliseconds. Part of your grade will be based on how fast your implementation is.

2.1 Points Labels

If `-c` is not specified to your program, it needs to write points assignment, i.e. the final cluster id for each point, to STDOUT in the following form

```
printf("clusters:")
for (int p=0; p < nPoints; p++)
    printf(" %d", clusterId_of_point[p]);
```

The autograder will redirect your output to a temp file, which will be further processed to check correctness.

2.1.1 Correctness of Output

To check the correctness of the output, the autograder will do the following

1. Compute the centroid of each cluster given by the points assignment.
2. For each point, select the cluster centroid that is closest to the point.
3. Compute the distance between the point and the selected centroid. Let the result be `dis0`.
4. Compute the distance between the point and the centroid of its assigned cluster. Let the result be `dis1`.
5. If the difference between `dis0` and `dis1` is larger than a threshold, this point is considered to be a *wrong point*. Your grade will be based on the total number of wrong points.

2.2 Centroids of All Clusters

if `-c` is specified, your program should output the centroids of final clusters in the following form

```
for (int clusterId = 0; clusterId < _k; clusterId ++){
  printf("%d ", clusterId);
  for (int d = 0; d < _d; d++)
    printf("%lf ", centers[clusterId + d * _k]);
  printf("\n");
}
```

Note that the first column is the cluster id.

2.2.1 Random Initial Centroids Generation

At the beginning of the kmeans algorithm, `k` points should be randomly chosen as the initial set of centroids. Since the final set of centroids depends heavily on the initial set of centroids, the autograder will specify the `seed` for random number generation so that your final set of centroids will be compared with the set of centroids using the same initial set of centroids.

Specifically, your program should use the `seed` provided by the cmdline argument to randomly generate `k` integer numbers between 0 and `num_points` and use these `k` integers as indices of points used as initial centroids.

E.g.

```
static unsigned long int next = 1;
static unsigned long kmeans_rmax = 32767;
int kmeans_rand() {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % (kmeans_rmax+1);
}
void kmeans_srand(unsigned int seed) {
    next = seed;
}
kmeans_srand(cmd_seed); // cmd_seed is a cmdline arg
for (int i=0; i<k; i++){
    int index = kmeans_rand() % _numpoints;
    // you should use the proper implementation of the following
    // code according to your data structure
    centers[i] = points[index];
}
```

Standard random number generators (like rand()) can differ accross platforms even when given the same seed. To make sure your starting centroids are consistent please use the `kmeans_rand` and `kmeans_srand` functions as they are written here.

2.2.2 Correctness of Output

The autograder will compare your centroids, refered to as `iCenters` to the centroids computed by MATLAB, referred to as `refCenters` according to the following steps

1. Map each centroid in `iCenters` to its closest centroid in `refCenters` using the Euclidean distance.
2. If more than one centroids in `iCenters` are mapped to the same centroid, say `refC0`, in `refCenters`, one of the centroids, say `iC`, will be forced to map to another centroid, say `refC1` in `refCenters` such that `refC1` is the closest centroid to `iC` that has a longer distance to `iC` than that from `refC0` to `iC`.
3. Repeat step 2 until no more than one centroid in `iCenters` are mapped to the same centroid in `refCenters`.
4. Compare the distance between each centroid in `iCenters` with its mapped centroid in `refCenters` with a threshold. If the distance is larger than the threshold, the centroid in `iCenters` is considered to be a *wrong centroid*. Your grade will be based on the number of wrong centers.

Note that there are three ways to pick `iC` in step 2: `Cur`, `Min`, and `Max`.

- `Cur`: the centroid with the largest centroid id will be forced to remap.
- `Min`: the centroid that has the shortest distance to `refC0` will be forced to remap.
- `Max`: the centroid that has the largest distance to `refC0` will be forced to remap.

The autograder will use all these strategies and choose the one with the fewest unmatched centroids.

Essentially, what this means is that we will generate the solution to an input file using our own code, then, for each centroid of our solution, we will find the centroid of the student's solution that is closest to it. Repeat for all centroids. Then we calculate the distance difference (error) of all these pair of centroids. Pairs that are too far (above a threshold) are considered wrong.

3. Implementations Hints

3.1 Seq --- Sequential CPU Implementation

You should figure out a convergence criterion for the algorithm to stop. The convergence check should be done at the end of every iteration and this should be included in the elapsed time.

3.2 CUDA_Basic --- Basic CUDA Implementation

3.2.1 Organization of files

The main goal of this step is to write a working CUDA program from scratch. There are different ways to organize your codes and here is one example:

- Put all your CUDA kernels in kmeans_kernel.cu. You may need several kernels to implement the Kmeans algorithm because kernels provide global synchronization among all threads.
- Write a wrapper function that sets up kernel launch configurations, calls the CUDA kernels to perform one iteration of computation, and checks if the algorithm has converged. Note that you can have different wrapper functions, which represent different optimizations over the basic implementation. Put all wrapper functions in kmeans_kernel.cu as well. If you use extra command line argument to control which Kmeans to run, remember to specify the command in the `submit` file.
- Put the main function in kmeans.cpp, in which the input file is read, centroids are initialized, host and device pointers are allocated, and most importantly, one of the wrapper functions are called to start one particular version of Kmeans implementation.
- All the wrapper functions can be declared in kmeans.h, which is #included in both kmeans_kernel.cu and kmeans.cpp.

- To compile your code, you need to do the following

```
nvcc -o kmeans.o -c kmeans.cpp
nvcc -o kmeans_kernel.o -c kmeans_kernel.cu
nvcc -o kmeans kmeans.o kmeans_kernel.o
```

3.2.2 Time measurement

To measure elapsed time of CUDA kernels, you will find that the `cudaEvent_t` API is key to doing this accurately. A good overview of how to use it to measure performance can be found here.

3.2.3 Non-determinism with `atomicAdd`

If `atomicAdd` is used by each CUDA thread to compute the aggregation, you should expect the number of iterations for the algorithm to converge becomes non-deterministic, i.e. every time you run the program, it stops at a different number of iteration even though every time it starts with the same initial centroids. This is because floating point number arithmetic is affected by the order of the computation. The use of `atomicAdd` makes the order non-deterministic, hence the result of the computation. A direct consequence of this non-determinism is when you compute the centroid of a group of points, every time you can have a different result (if your threshold is relatively small) even the group of points does not change. Therefore, a relatively larger (10X the threshold used for sequential implementation) might be used for convergence test. Or you can use other convergence tests which do not involve comparing centroids.

3.3 CUDA_Shmem --- CUDA Implementation with Shared Memory

Shared memory can be used to reduce the number of global memory accesses. The common pattern to use shared memory is as follows

```
load data from gmem to shmem
__syncthreads();
update shmem
__syncthreads();
store data from shmem to gmem
```

It is very easy to make mistakes when you do not need all threads to load and store the data but need them all to update the data in shmem. This happens when the total number of threads does not match the number of tasks (such as points). Be careful when you try to mask some threads when using shared memory.

3.4 Thrust --- Parallel GPU Implementation using Thrust

K-Means can be decomposed as a GroupBy-Aggregate, which can be implemented relatively easily in CUDA using `thrust::` primitives. The advantage of Thrust over CUDA implementation is that you are working on the algorithm level. The main work will be how to represent the Kmeans algorithm with a combination of Thrust implemented algorithms. You do not need to worry about data (de)allocation/copy and work assignment for each CUDA thread.

The performance of the Thrust implementation may not be as good as the CUDA implementation. But it should still be faster than the sequential implementation.

This is the hardest part to implement, so we suggest you leave it as the last part.

Some of the Thrust APIs you should look into are: `thrust::for_each`, `thrust::stable_sort_by_key`, `thrust::reduce_by_key`, `thrust::transform`, `thrust::upper_bound`

For this implementation, you will probably have to store more information than the previous ones. For example, while in CUDA you can use the index of an array to identify it (e.g. index 0 means centroid 0), in Thrust you might want to create an array to store the indices: have an array called centroids and a sequenced index array for each centroid.

This simplifies iterating through every centroid since APIs like for_each take a start and end iterator argument. The API `thrust::sequence` makes creating this trivial.

Before writing code, you should understand what `functors` are. Functors are where your own code will be implemented, and Thrust will simply map this functor to all the elements of an iterators. For example, you will probably need a functor for finding the nearest centroid, and apply this for each point in your input.

4 --- Your report

You should include a report that answers the following questions. In cases where we ask you to explain performance behavior, it is fine to speculate, but be clear whether your observations are empirical or speculation.

- Report the GPU hardware details, CPU hardware details, and OS version on the machine where you did your measurements if you measured in any environment other than Codio; if you just used Codio for measurements, it's fine to just report that fact.
- Which of your implementations is fastest? Does it match your expectations of which should be fastest? Estimate the best-case performance speedup your CUDA implementations should have based on the number of threads in your program and the number of processing contexts actually supported by your hardware. How far of that prediction is your best-case performance?
- Which of the parallel implementations is slowest, and does it match your expectations? Why or why not?
- What fraction of the end-to-end runtime in your CUDA versions is spent in data transfer?
- How much time did you spend on the lab?