

# Introduction

The purpose of this assignment is for you to become familiar with GPU programming and the CUDA API. We will do this by parallelizing the *k-medoids clustering* algorithm, which we have already worked with in the first assignment.

**[Input]** You need to write a program called *km\_cuda* that, as before, will accept the following parameters as input: a filename of data points, the number of clusters, the number of thread blocks and the number of threads per block. Although the third and fourth parameters suggest a 1D grid with 1D thread blocks, please feel free to adapt them to higher dimensional grids and blocks if needed. For example, a configuration with 256 thread blocks can be reinterpreted as a 16 x 16 2D grid.

**Update:** In your CUDA program, if it is more intuitive for you to dynamically determine the number of thread blocks for each kernel based on the data being processing, then use -1 as the value for the thread blocks parameter (third parameter during the invocation) and use that as a flag to indicate that the number of thread blocks will need to be computed during runtime based on the size of the data (for example, based on the number of elements in an array being processed). The number of threads per block (fourth parameter) should still be specified as a positive integer, which can then be used to determine the number of thread blocks.

**[Output]** Your program should, again as before, read in the specified file, cluster the given data points, and output the cluster assignments.

## K-Medoids Clustering Algorithm [Review]

The k-medoids clustering algorithm clusters  $N$  data points into  $K$  clusters. Each cluster is characterized by a medoid, which is a data point in the cluster that has the smallest distance to all other points in the cluster. The algorithm proceeds as follows:

1. Select the initial  $K$  medoids.
  - a. For reproducibility, use points 0, 1, ...,  $K-1$ .
2. Assign each data point to the closest medoid (measured via Euclidean distance).
3. While not converged:
  - a. In each cluster, make the point that minimizes the average Euclidean distance to all other points in the cluster the (new) medoid.
  - b. Assign all data points to the closest medoid as measured via Euclidean distance.

Convergence is detected when no data points change their cluster assignment, or if the maximum number of iterations have been executed.

For this assignment, you should set the maximum number of iterations to twenty.

## Input/Output Formats

Each program will take as input one file (the list of data points), and output two files ("*clusters.txt*": the cluster assignments and "*medoids.txt*": the centers of each cluster).

The input file contains  $N+1$  lines. The first line contains two space-separated integers: the number of data points ( $N$ ), and the dimensionality of each data point ( $D$ ). The following  $N$  lines each contain  $D$  space-separated floating-point numbers which represent the coordinates of the current data point. Each floating-point number contains at least one digit after the decimal point. For example, an input with four two-dimensional data points would be stored in a file as:

```
4 2
502.1 505.9
504.0 489.4
515.2 514.7
496.7 498.3
```

The output file cluster assignments must be called *clusters.txt* and contain  $N$  lines. Each line should contain a single zero-indexed integer which specifies the cluster that the current data point belongs to. For example, a clustering of the above input file into two clusters may look like:

```
0
1
0
0
```

The second output file, *medoids.txt*, should **mostly** follow the same format as the input data file.

*There are a couple of changes we will introduce to this format for the output file.*

- It should contain  $K$  data points, one for each cluster. However, do not include the line in the output file which indicates the number of clusters and number of dimensions since this is already known from the input parameters.
- Each coordinate should be written with at least **8 digits** after the decimal point.

Your program must also print the clustering time to standard out. You should use a high-precision, monotonic, wall-clock timer and **also omit the time spent reading and writing to files**. We recommend the function `clock_gettime()` when on a Linux system.

Since you want the GPU to finish the execution, call the `cudaDeviceSynchronize()` API before making the calls to the timer.

Here is a function that you may use for timing:

```
/* Gives us high-resolution timers. */
#define _POSIX_C_SOURCE 199309L
#include <time.h>

/* OSX timer includes */
#ifdef __MACH__
    #include <mach/mach.h>
    #include <mach/mach_time.h>
#endif

/**
 * @brief Return the number of seconds since an unspecified time (e.g., Unix
 *        epoch). This is accomplished with a high-resolution monotonic timer,
 *        suitable for performance timing.
 *
 * @return The number of seconds.
 */
static inline double monotonic_seconds()
{
#ifdef __MACH__
    /* OSX */
    static mach_timebase_info_data_t info;
    static double seconds_per_unit;
    if(seconds_per_unit == 0) {
        mach_timebase_info(&info);
        seconds_per_unit = (info.numer / info.denom) / 1e9;
    }
    return seconds_per_unit * mach_absolute_time();
#else
    /* Linux systems */
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec * 1e-9;
#endif
}
```

You should use the following function to output your clustering time:

```
/**
 * @brief Output the seconds elapsed while clustering.
 *
 * @param seconds Seconds spent on k-medoids clustering, excluding IO.
 */
static void print_time(double const seconds)
{
    printf("k-medoids clustering time: %0.04fs\n", seconds);
}
```

Timing information should be the **ONLY** thing printed to standard out.

**Failure to follow any of these output instructions will result in significant loss of points.**

## Testing

Test inputs can be found in `/export/scratch/CSCI-5451/assignment-3/` on any of the cuda lab machines. We provide two test files: *small\_gaussian.txt* and *small\_cpd.txt*. The former is a small two-dimensional dataset that you should use for testing the correctness of your code.

The cuda machines are located at `cse-cuda-0{1..5}.cselabs.umn.edu`. You must first load the cuda modules with the following commands before using `nvcc`.

```
module load soft/cuda/local
module initadd soft/cuda/local
```

If the command 'module' cannot be found. Add the following lines into your `~/.bashrc` file.


```
MODULESINIT="/usr/local/modules-tcl/init/bash"
if [ -f $MODULESINIT ]; then
  . $MODULESINIT
  module load java system soft/ocaml soft/cuda/local
fi
unset MODULESINIT
```

After adding, run

```
source ~/.bashrc
```

More info on testing/using the cuda machines can be found at [Parallel Lab Systems](#).

**You should only run your code on one of the cuda machines.** You will use the last digit of your student ID to select the machine. ID's ending in {0, 5} should use cuda01, ID's ending in {1, 6} should use cuda02, ID's ending in {3, 7} should use cuda03, ID's ending in {4, 8} should use cuda04, and ID's ending in {5, 9} should use cuda05.

For the longer running tests, you should look into CUDA compiler optimization flags as well as the [screen](#)  command as these will be helpful.

Remember that the TA will be evaluating your data with a different dataset than those provided for testing.

## What you need to turn in ↓

1. The source code of your programs.
2. A file named '**best\_run.txt**' containing the complete command used to achieve the best-performing result. This command will be used by the TA to generate the runtime for computing speedup. For instance, if using 32 blocks with 16 threads each gives you the best performance for creating 512 clusters, then store './km\_cuda /export/scratch/CSCI-5451/assignment-3/small\_cpd.txt 512 32 16' inside the file.
3. A short report including the following parts:
  1. A short description of how you went about parallelizing the k-medoids algorithm. You should include how you decomposed the problem and why, i.e., what were the tasks being parallelized.
  2. Give details about the number of elements and what computations in your kernels are handled by a thread.
  3. Provide details about the thread hierarchy chosen, i.e., whether the threads are organized in a 1D, 2D, or, 3D fashion in a thread-block, and whether the thread-blocks are arranged in 1D, 2D, or, 3D grid.
  4. You need to perform a parameter study in order to determine how the number of elements processed by a thread and the size of a thread-block, i.e., the #threads in a block, affect the performance of your algorithm. Your writeup should contain some results showing the runtime that you obtained for different choices. **NOTE:** If you choose to write CUDA kernels where the number of thread blocks is determined dynamically by the program during runtime, then analyze how changing the number of threads affects the runtime performance.
  5. You should include results on the 'small\_cpd.txt' dataset with 256, 512, and 1024 clusters.
  6. Compare your best performing results on the GPU with the serial and the OpenMP implementation from assignment 1. A sample OpenMP implementation is provided in this [link](#) ↓.
  7. Remember, speed counts. Programs that fail to use the GPU efficiently will lose significant points.

**Do NOT include the test files; TAs will have their own test files for grading.** You will lose significant points for including test files.

### Additional specifications related to assignment submission

- A makefile must be provided to compile and generate the executable. The executable should be named:
  - km\_cuda
- **Program invocation:** Your programs should take as arguments the input file to be read from, the number of clusters to be used for parallel execution, the number of thread blocks, and the number of threads.
- For example, to create 512 clusters using 32 thread blocks with 16 threads in each block, your program would be invoked as follows:

```
./km_cuda /export/scratch/CSCI-5451/assignment-3/small_cpd.txt 512 32 16
```

- **NOTE:** If you choose to write CUDA kernels where the number of thread blocks is determined dynamically by the program during runtime, then specify -1 as the input argument for the number of thread blocks during invocation. Example:

```
./km_cuda /export/scratch/CSCI-5451/assignment-3/small_cpd.txt 512 -1 16
```

- All files (code + report) MUST be in a single directory and the directory's name MUST be your UMN login ID (e.g., your UMN email or Moodle username). Your submission directory MUST include at least the following files (other auxiliary files may also be included):

```
<UMN ID>/km_cuda.c  
<UMN ID>/Makefile  
<UMN ID>/best_run.txt  
<UMN ID>/report.pdf
```

If you choose to code in C++, then replace the .c suffixes with .cpp or .cc.

- Submission MUST be in .tar.gz
- The following sequence of commands should work on your submission file:

```
tar xzvf <UMN ID>.tar.gz  
cd <UMN ID>  
make  
ls -ld km_cuda
```

This ensures that your submission is packaged correctly, your directory is named correctly, your makefile works correctly, and your output executables are named correctly. If any of these does not work, modify it so that you do not lose points. The TAs can answer questions about correctly formatting your submission **BEFORE** the assignment is due. Do not expect them to answer questions the night it is due.

**Failure to follow any of these submission instructions will result in significant loss of points.**

## Evaluation criteria

The goal for this assignment is for you to become familiar with the CUDA APIs.

As such, full points will be given to the programs that:

1. follows the assignment directions, make sure that the format and the file names are followed as provided in the description;
2. solve the problem correctly;
3. do so in parallel (i.e., both clustering sub-steps are parallelized);
4. give details about the study of program performance for different kernel launch parameters;
5. create the file (best\_run.txt) which contains the command that results in your program's best performance;