HiPC Parallel Programming Challenge - GPU Track

Background

To investigate the subatomic structure of matter, high-energy physicists typically collide particles travelling at very high speeds. By observing the resulting particles with different sensors, the properties of these resulting particles can be studied. One sensor used in experiments at CERN, SLAC and other high-energy physics research facilities is a “Ring-Imaging Cherenkov (RICH) detector”, which helps to determine the type of charged particles.

A RICH detector measures the Cherenkov radiation produced by a charged particle when travelling faster than the phase speed of light in a medium, similar to the sonic boom of a jet. This Cherenkov light, emitted in a cone around the particle, can be optically imaged and appears as rings in the imaging plane. Physicists need to characterise these rings as quickly as possible, since this provides them with important information to help them understand the physics behind the phenomenon they are observing.

Figure 1 depicts the detector output of a typical high-energy physics experiment.

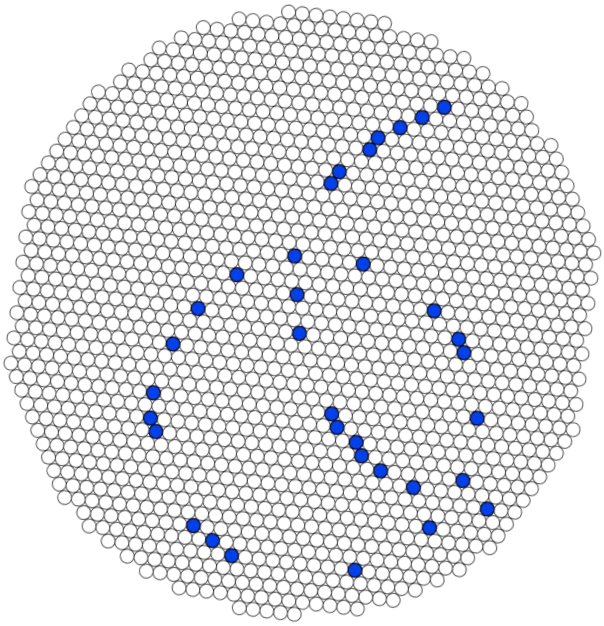

Figure 1. Typical detection event

The hollow circles represent the detector array; each individual circular sensor detects photons. In the figure, the sensors that have turned blue represent detection activity. If enough particles from same experiments are allowed to hit the detector, circular patterns emerge on the detector

(two circles can be observed in figure 1). By looking at the properties of the circle, such as the radius and location, physicists can uniquely identify the particles that were emitted from the collision.

The detector consists of discrete sensors, meaning that the detection of photons is subject to some noise (i.e. error). Therefore the first step in processing the sensor data is to find the circles that pass through the detected set of points; in other words we need to compute one or more circles that could fit a given set of points with minimal error.

Throughout an experiment, many thousands or even millions of detections need to processed every second; this needs to happen in real-time in order to discard uninteresting events (thereby dramatically reducing storage requirements).

Problem Statement

To allow teams to focus on parallel algorithms and optimisation, the problem has been simplified from the actual RICH detector problem described above, primarily by eliminating noise and by using a continuous coordinate system (as opposed to a discrete system which would represent the sensor array more realistically).

You are given a batch of event images, where each event image consists of a set of points (x, y) on a 2D plane. For each event, find the minimal set of circles that fit the given points. The following axioms should be considered:

1. Each circle has five or more points on its circumference.
2. Although there is no noise, you should consider that the rounding of coordinates can introduce some rounding error.
3. A point can be a member of one or more circles, **all** points will be a member of **at least one** circle.

The coordinates of the points will be in the range [-1.0f, 1.0f], and the radius of the circles will in the range [0.1f, 1.0f]. Accuracy will be measured using the Least Absolute Deviations (LAD) measure[1], an event is said to match the reference if the EventLAD (the sum of LADs for all points in the event) is less than 10^{-10}.

Points are awarded for accuracy and performance.

There will be maximum 100 points per event (minimum 3).

There will be maximum 5 circles per event (minimum 1).

There will be maximum 10000 events in a batch (minimum 1).

[1] Least Absolute Deviations (LAD) for a point is defined as the minimum of the *deviation* from each circle, where *deviation* of point $i=(x_i, y_i)$ from circle $k=(x_k, y_k, r_k)$ is defined as $((x_i - x_k)^2 + (y_i - y_k)^2 - r_k^2)^2$.
$LAD(P_i) = \min(\text{deviation}(P_i, C_k)) \ \forall \ k = 1..K$
$\text{EventLAD} = \sum LAD(P_i)$

Example Data Sets and Serial Implementation

Together with this document, you are provided with input data sets, corresponding reference outputs, and a serial example program written in Octave. This serial program should be considered as **an example only** and is described in more detail below; there are **various other approaches** to solving this problem which you could consider.

The example program can be invoked from the command line as follows:

```
octave -q -f findrings ../data/batchXX.dat
```

Input Format

Each input file (`batchXX.dat`) begins with an integer N in the first line which indicates the number of event images in the data set. This is followed by N sets of points (i.e. one set of points per event), where each set of points consists of the number of points, M, on the first line followed by M lines containing the coordinates of the points.

For example, a batch with two events, four points in the first event and three in the second, would have the following format:

```
2
4
x.xxxx y.yyyy
x.xxxx y.yyyy
x.xxxx y.yyyy
x.xxxx y.yyyy
3
x.xxxx y.yyyy
x.xxxx y.yyyy
x.xxxx y.yyyy
```

Output Format

On executing the program, your program should print a header “Event: X” for each event (counting from 1), followed by the circles, one circle per line with the coordinates of the centre followed by the radius.

After outputting the circles, the final line should report the time taken. A simple timer is provided in the `team_name` directory for this purpose; the timer should be started *immediately after* the points are loaded from the file, and stopped *immediately before* the circles information is printed. The time should be reported in milliseconds.

For example, a batch with two events, two circles in the first and one in the second, would have the following format:

```
Event: 1
x.xxxx y.yyyy r.rrrr
```

```
x.xxxx y.yyyy r.rrrr
Event: 2
x.xxxx y.yyyy r.rrrr
Time: t.tttt ms
```

Your program should **not** print any other output on `stdout` (any output on `stderr` will be ignored for verification).

Algorithm Used in Serial Implementation

The algorithm used in the serial example implementation is inspired from the K-closest-circles (KCC) algorithm as described in [1]. In turn, KCC is a variation of K-means clustering.

The KCC algorithm proceeds as follows.

Input:

1. N = number of points
2. X = the N points in the 2D coordinate system
3. K = number of circles to fit

Output:

1. P_i , i ∈ {1..K} : the assignment of points in X to the i-th circle (hence, sum(count(P_i)) = N)
2. C_i , i ∈ {1..K} : the circles, represented as centre coordinates and radii.

Algorithm Steps:

1. Create the set C of initial circles (C_i , i ∈ {1..K})
2. While not finished:
    a. For each point, assign it to the circle that is closest to it - this populates all P_i , i ∈ {1..K}
    b. For every P_i, fit a circle using least squares method - this creates a *better* set C.

Stopping Criteria:

Stop if *either* of these conditions are met:

1. No change is assignment of points to circles from the previous iteration
2. Maximum number of iterations is reached

Having defined the KCC Algorithm, the serial algorithm is straightforward.

1. Define the maximum number of circles, M. In the serial implementation, M is 5 as specified in the problem statement.
2. For *k* from 1 to M:
    a. Try to fit *k* circles to the input points using KCC algorithm.

    b. Evaluate this solution by summing up the minimum distance of each point from all circles, adding a penalty for a larger number of circles - in the serial implementation we use a square penalty[2] 0.001 × k × k.
3. Select the solution which minimizes the above summation across all M

The quality of the results (i.e. the LAD measure) from the KCC algorithm depend on the original guess for the *k* circles, so the quality of the initial guess can have a significant impact on accuracy. There are many possible approaches for the initial guess, the serial implementation tries the following two approaches and chooses the one producing the lower error:

1. Initialize the *k* circles randomly
2. Partition the points into *k* groups, assigning points to a group randomly, and fit a circle to each group as the initial guess

Data Sets

You are provided with the following datasets (`batchXX.dat`) and their corresponding output (`circlesXX.dat`) in the data directory.

Name	# Events	Description
batch00.dat	10	A basic correctness check for development. 10 events, 2 circles per event, 25 points per circle
batch01.dat	10	A basic correctness check for development, the main difference from 00 is that the number of circles and points per circle is random, thus stressing more paths through your code.
batch02.dat	1000	A larger case, suitable for timing and profiling.
batch03.dat	2000	A larger case, suitable for timing and profiling.

[2] We penalise results with a larger number of circles to avoid over-fitting [2]

Getting Started

The first thing to do is to ensure you can build and run the example program and compare the output with the corresponding `circlesXX.dat` file. There should be no problems with this step but please post about any issues on the mentor forum.

The next step is to decide how to parallelise the algorithm and how to implement your parallel algorithm. The system you will use is described below; to avoid problems during evaluation, you should not install any other components unless you can include those with your final submission (please ensure you have the rights to distribute any code you have developed, and please mention this in your submission README).

The test and evaluation systems will have the following configuration (subject to change):

- CUDA Toolkit v7.5
- PGI OpenACC Compiler 15.7
- NVIDIA Tesla K40 GPU

Note that developing and building your GPU implementation does not require access to a CUDA GPU since the CUDA and OpenACC Toolkits can be installed without a GPU (freely downloadable from http://www.nvidia.com/getcuda and http://www.nvidia.com/openacctoolkit).

Access to GPUs

Teams will be provided with access to a GPU system in the second week of the challenge. The same systems will be used for evaluation so you should ensure that your submission builds and runs on these systems.

There is no intermediate evaluation step; **all** teams will be given access to GPUs in the second week of the challenge, and a single submission is expected at the end of the challenge.

Submission and Evaluation

You are expected to produce a GPU implementation of the algorithm using CUDA C/C++, OpenACC or OpenCL, producing output in the format described above.

Your final submission should consist of a single gzipped tar file (`.tar.gz`) containing the following:

- team_name/README
- team_name/Makefile
- team_name/<filename> [One or more source code files]

Do **not** include the serial or data directories that were provided to you.

The `README` file should contain a brief description of your algorithm and how it is parallelised across the GPU. Please also cite any references you may have used in your implementation, there is no penalty for referencing prior work.

The `Makefile` can contain any number of build targets, but for evaluation only the default target (i.e. the first target) will be tested. You should ensure that the build completes and produces an executable named `findrings`. It should include any command line options you expect to provide to the compiler.

Note that the evaluation will be done on a system matching the system provided to you, please test your solution on that system before submitting.

The submission will be validated with the following steps:

1. Change to the team_name directory (`cd team_name`)
2. Build the candidate code (`make`)
3. Execute the candidate program and verify the output
   (`for n in $(seq -f "%02g" 0 N); do ./findrings ../data/batch$n.dat > output$n.dat; ../scripts/checkLAD output$n.dat ../data/batch$n.dat; done`)

Note that if `checkLAD` indicates your program did not validate, you can use the `toldiff` script to determine which of the circles failed to matchthe reference solution. The usage is as follows:

```
../scripts/toldiff outputXX.dat ../data/circlesXX.dat
```

If your submission validates (i.e. the `checkLAD` script reports that 85% or more events matched for all batches) then the submission will be benchmarked as follows:

1. Change to the team_name directory (`cd team_name`)
2. Build the candidate code (`make`)
3. Repeat N_ITER times:
   a. Execute the candidate program and verify the output
      ```
      ./findrings ../bench/batch.dat > output_bench.dat
      ../script/checkLAD output_bench.dat ../bench/batch.dat
      ```

b. Extract the time from `output_bench.dat`

The Least Absolute Deviations (LAD) summed over all events and the average time for the N_ITER executions will be recorded for evaluation - points are awarded for accuracy and performance.

Note that the benchmark data set (`bench/batch.dat` and `bench/circles_bench.dat`) are not provided to the candidate teams.

References

[1] T Marosevic, “Data clustering for circle detection”, *Croatian Operational Research Review*, vol 5, pp 15-24, 2014

[2] Wikipedia, “Overfitting”. [Online] Available: https://en.wikipedia.org/wiki/Overfitting