Documentation

Managing the election process for the new president in a parallelized way -OpenCL

Names: Mahmoud Gamal
Moustafa Elsayed
Khaled Osama

Summary

Implementation steps

File generator:

Generate a data file that contains the voter's preferences. The generated file uses a shuffle function that randomly shuffles the an array [1-C] , as C is the number of candidates , to generate a random voting preference list used in the main program. The file generator is **parallelized** using OpenMP

OpenCL environment setup :

We used OpenCL API to generate parallized programme and followed the same general procedure. Illustration of our procedure is as follows :

1. Initialization

   a. The first step was to initialize the host variables , `num_votes` , `num_candidates and data` , by scanning the data from the generated file and copying it to those variables.
   b. Then initialize the local size and the global size :

```
 localSize = 100; // this can be anything and it will work

globalSize = ceil(num_votes/(float)localSize)*localSize;

num_elements = ceil(num_votes/(float)globalSize); //data chunk
```

   c. After that, we initialized the buffer for OpenCL :

```
// Device input buffers

     cl_mem d_data;

// Device output buffer

     cl_mem d_global_sum;

     cl_mem d_round_2_variables;
```

   d. At this point there are many general steps for all opencl programmes ,so we put them in some header file called `setup.h` to accomplish the following :
      - Query for Platform IDs
      - Search for OpenCL™ compute devices in system
      - Creating Context
      - Creating Command Queue

2. Allocate resource

   In this step we put the data into the created buffers by using `clCreateBuffer and clEnqueueWriteBuffer`

3. Create Program/ Kernel
   a. Also in the `setup.h` we accomplished these typical steps :
      i. Load kernel
      ii. Create program
      iii. Build programme

b. Then , we set the arguments to compute the kernels

4. Execution

   We executed the programme by using `clEnqueueNDRangeKernel`.

   Then, we print our output. Tos see the output format please visit the Results section.

5. Tear Down

   FInally we cleaned our variables by using `clReleaseMemObject`

Round 1

a. Every thread gets its part from the data array by defining the start line and the end line that will be processed by that process from the file. Code line used :

```
int begin = id * num_elements;

int end = begin + (num_elements - 1);

begin = begin * num_candidates;

end = end * (num_candidates) + num_candidates;
```

b. For every line in its part from the file , the process gets the most preferred candidate , the first one , and makes a statistics about the number of votes for every candidate in the first position.
c. The reduce function in OpenCL collects the statistics in two steps :
   a. The first step is done locally by collecting the statistics of all threads in the same work group.
   b. The second step is to perform the same process globally for all work groups.

Find Max and check

After getting the statistics of all candidates , there should be some checkpoint to check either one of the candidates already one or we should go to the second round between the first two candidates. So , we find the candidate with the maximum percentage of votes and if it has more than 50% of the votes he wins the elections , the top two candidates are upgraded to the second round.

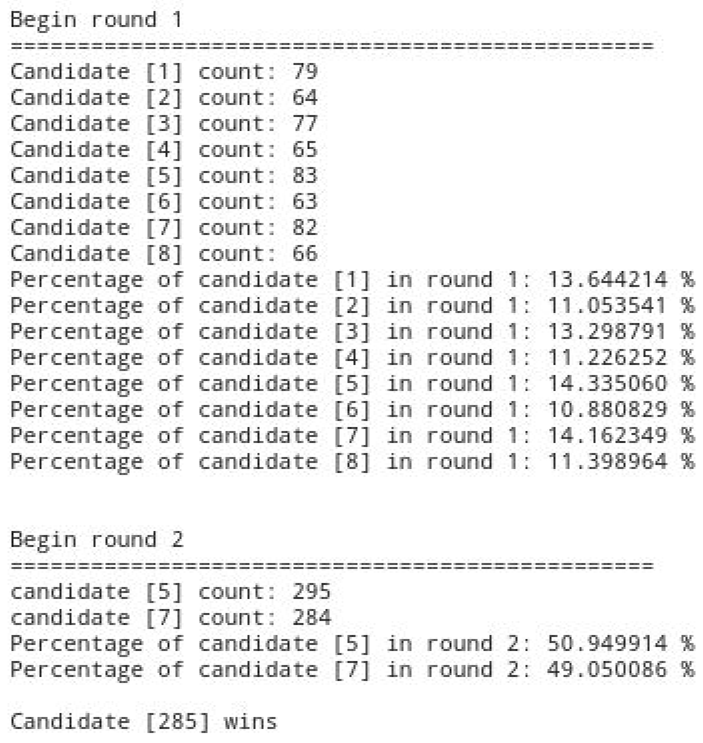
Round 2 :

Following the same procedure in the round one but :

A. If the first candidate is one of the top candidates , no action needed
B. If the candidate is not one of the top candidates , a check is made over the list of candidate preferred list for that vote and the first most preferred one from the two candidates gets the vote.
C. The candidate with a percentage of votes more than 50 % wins the elections.

Results:

To give intuition about our results and to confirm our procedure we would like to give screen shot of the output : (note: Those are the results for our random text file, it will differ if it is not the same file, we compared the results with phase1 code and it produces the same exact result)



```
Begin round 1
================================================
Candidate [1] count: 79
Candidate [2] count: 64
Candidate [3] count: 77
Candidate [4] count: 65
Candidate [5] count: 83
Candidate [6] count: 63
Candidate [7] count: 82
Candidate [8] count: 66
Percentage of candidate [1] in round 1: 13.644214 %
Percentage of candidate [2] in round 1: 11.053541 %
Percentage of candidate [3] in round 1: 13.298791 %
Percentage of candidate [4] in round 1: 11.226252 %
Percentage of candidate [5] in round 1: 14.335060 %
Percentage of candidate [6] in round 1: 10.880829 %
Percentage of candidate [7] in round 1: 14.162349 %
Percentage of candidate [8] in round 1: 11.398964 %


Begin round 2
================================================
candidate [5] count: 295
candidate [7] count: 284
Percentage of candidate [5] in round 2: 50.949914 %
Percentage of candidate [7] in round 2: 49.050086 %

Candidate [285] wins
```

Challenges :

We faced some obstacles through our development for the code :

1. A major challenge in the Jupyter notebook is that all code was crammed in one big cell, so we decided to modulate the flow, so we divided the cells into (LoadKernel - Setup - Kernel (.cl) - source code). Now everything is in place and easier to navigate.

2. One of the critical problems in OpenCL is it can suffer from race conditions at calculating the statistics of the voters in both the local and global sums.

   Solution: We used the barrier in OpenCL to overcome such a problem :

   ```
   barrier(CLK_LOCAL_MEM_FENCE);

   barrier(CLK_GLOBAL_MEM_FENCE);
   ```

3. The random generation for the file was very tricky , as in c language the same seed is held every time , so it either neglects some candidate from the first percentage or gives sharp accurate percentages.

   Solution : this forces us to use C++ and depend on `chrono` to generate a really random preference list as shown in the snapshot.

4. The number of candidates is a critical parameter in the OpenCl thread scheduling , as if it was not divisible by the number of processes there will be a reminder .

   Solution : to handle such a problem we decided to use the following equation

   ```
   int num_elements = ceil(num_votes/(float)globalSize);
   ```

Integration into github repo

The last part of this project was to integrate the project and put it into a github repo : https://github.com/moustafa-7/parallel-project and follow instructions in the README file to run the whole project from one file.

Materials & Resources

We used Jupyter Notebook Colab and AWS as our main platform for testing and developing the code. The labs and tutorials were a great reference to check.