CSci 5451, S'20 Programming Assignment # 1 Due Date: Feb. 29th, 2020

Background

This assignment will familiarize you with CUDA, openMP, and some of the basic issues related to the performance of parallel codes. There are two parts to this assignment.

(a) Matrix-Matrix products in openMP

In the first part, you are given a C code (will be posted) that performs a matrix-matrix product and your task is to insert `openMP` directives in order to speed up the code. This is a common situation that arises in practice. Your code will be compared with the LAPACK version that is available through BLAS. This is meant to be a simple exercise as most of the coding and set-up is already done.

Specific questions to answer:

1. Explain what you did to improve performance.
2. Run your code with 1, 4, 8, 16, 32 threads and plots the curve of the times you get versus the number of threads. On the same figure plot a horizontal line that shows the time you get with Lapack.
3. Calculate the best flops rate you get with your code.
4. When you increase the dimension to a larger matrix [double the size] does the performance drop? Can you give a possible explanation as to the reason?

What to hand-in for part (a). You need to provide on canvas your C program and a short report that answers questiosn 1 to 4 above. For this you will need to use any one of the nodes phi01 to phi08 of the phi cluster. [It is recommended to do a 'who' to see how many people are logged in. If someone else is working on the same machine this could impact the timing of your code]

(b) Testing vector operations on GPUs

In the second part you will test CUDA for performing a simple and common `saxpy` operation, which is an operation of the form

$$y := a * x + y$$

where x, y are two vectors of length n and a is a scalar. The goal of this part of the assignment is simply to vary the parameters and options provided by CUDA to achieve the best possible performance, and to measure and possibly analyze this performance.

Your first task here is to write a CUDA kernel which does a saxpy in parallel. This function (call it `saxpy_par`) should have as inputs: n, a, x, and y.

You will now also need to check the results of the saxpy on the CPU (simply because the GPU does not necessarily report errors when it fails). For example you can use the following function

```
 float saxpy_check(int n, float a, float *x, float *y, float *z)  {
 // a, x, y == original data for saxpy
 // z = result found -- with which to compare.
 float s=0.0, t = 0.0;
 for (int i=0; i<n; i++) {
      y[i] += a * x[i] ;
       s += (y[i] - z[i])*(y[i] - z[i]);
       t += z[i]*z[i];
 }
  if (t == 0.0) return(-1);
    else
  return(sqrt(s/t));
}
```

With these, your main program should do the following.

```
1. Init:
    Select two given vectors $x, y$ of length $N = 524288$ each and select $a$.
    Select $a$ (e.g., =1) and fill $x, y$ with random values
    Then select a number NITER of iterations [take NITER = 100], and set
    $a = a/\text{NITER}$ (so we know the NITER saxpys will amount to adding $x$ and $y$ when the
        original $a$ equals 1.)
2.  Loop over vector lengths:
    // Vary vector length from 1024 to 131072 by doubling it length each time:
    for (int vecLen = 1024; vecLen <= N; vecLen*=2) { ...
3       Copy vectors to GPU:
            Use cudaMemcpy(...)
            Copy only the needed portion not all $N$ entries
4       Prepare to call kernel:
            set grid-dimensions, block-dimensions, ...
5       Call saxpy_par kernel NITER times:
            for (iter = 0 ;iter<NITER; iter++)
                saxpy_par <<< ..., ... >>> (vecLen, a, x_d, y_d);
            Call saxpy_check to get the size of the error of the saxpy
4.0         Compute and print performance stats for this vector length
    } end vecLen (threads) loop
// Done
```

Your tests will be done on one of the nodes `cuda01, cuda02, ···, cuda08` of the cuda cluster. Your code will compute the performances in MFLOPS for each vector size and produce an output that looks like:

```
** vecLen =   1024, Mflops =   xx.dd, err = xx.xxe...
** vecLen =   2048, Mflops =   xx.dd  err = xx.xxe...
** vecLen =   4096, Mflops =   xx.dd  err = xx.xxe...
etc. ...
** vecLen = 524288, Mflops =   xx.dd  err = xx.xxe...
```

Note that 524288 is 2^{19} so this output should have 20 lines. The timing will be done on the host. Your flops rate should reflect the computation in the 'iter' loop (in step labeled 5. Note that this should exclude the time for the `saxpy_check` function call).
The next part aims at exploring the impact of communication. The data is sent once and then a large amount of computing is done with it – which may lead to good performance. What if we need to take into account the time it takes to transfer the data from CPU to GPU and back.. You will provide a second code, call it `testSaxpyC`, in which you time both transfer time to/GPU and computation. The iter loop becomes something like:

```
t1 = wctime();   // record time here
for (iter = 0; iter< NITER; iter++) {
  cudaMemcpy(y_d, y_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(x_d, x_h, size, cudaMemcpyHostToDevice);
  saxpy_par <<< .... ... >>> (len, a, x_d, y_d);
  cudaMemcpy(z_h, y_d, sizeof(float)*len, cudaMemcpyDeviceToHost);
 }
 t2 = wctime(); // record exit time here
 saxpy_check (...)
 // print results for this vecLen...
 ...
```

Because the data is re-read each time now you need to take $a = 1$ to get an error of zero with saxpy_check. Produce a table similar to the one before with this second code.

What to hand-in for part (b)
1.) Two source codes one called `testSaxpy` (data transfers excluded in timing) and the second `testSaxpyC` (data transfers included in timing).
2.) Two scripts showing execution results for each of these codes.
3.) A write-up (preferably in PDF format) explaining what observe.

Grading criteria. Please note: In addition to being correct, your codes should be well documented and easy to read. For both parts (a) and (b), grading will take into account things such as style and documentation.