99

Fall 2018 CIS 620 Homework 2

(Due Nov. ~~5~~ 7)

The purpose of this assignment is to introduce you the CUDA GPU and the MPI programming environments. Use the following command

tar xvfz ~cis620s/pub/gpu620.tar.gz

to extract the necessary files under the directory `NVIDIA_CUDA_620_F18`.

Part I: deviceQuery

Use `make` to build the executable file `deviceQuery` under the subdirectory `deviceQuery` and then run it. Take a screen shot of the result. What is the GPU clock rate? How many CUDA cores?

Part II: Euclidean Distance

You are asked to use several GPUs over MPI to find the maximum Euclidean distance to the origin in parallel. First, follow the instructions in ~cis620s/pub/MPI_setup. Next, you need to modify the code under the subdirectory simpleMPI. Assume that for each point i, its coordinate is $(A[i], B[i])$. Therefore, your root node has to initialize the array A and the array B. Then, use the MPI_Scatter to dispatch the data to each node specified in the file `machinefile` for calculation on GPU. Before calling MPI_Reduce to collect the result, each node should print its local result along with its hostname. Take a screen shot of the result.

Turning it in

Each groups needs to submit this homework using the following turnin command on grail:

```
turnin -c cis620s -p hw2 NVIDIA_CUDA_620_F18
```

Each group also needs to hand in a hard-copy document which includes the description of your code, experiences in testing/debugging, experimental results, etc. The document should be typed. The cover page should contain your photo(s), name(s) and the login-id you used to turnin. Start on time and good luck. If you have any questions, send e-mail to `sang@eecs.csuohio.edu`.

(97)

100

Details can be found in
https://help.ubuntu.com/community/MpichCluster

Below is just for CIS620 students :

1. Pick up a machine (e.g. arthur) from which you want to run the MPI root node. Login to the machine.

   Use
   ```
   arthur> ssh-keygen -t rsa
   ```
   and type a passphrase to generate an RSA key pair under the default directory ~/.ssh/id_rsa.

2. Add this key to authorized keys:
   ```
   arthur> cd .ssh
   arthur> cat id_rsa.pub >> authorized_keys (if authorized_keys exists)
           or
       cp id_rsa.pub authorized_keys (if authorized_keys not exists)
   ```

(You may repeat Steps 1 and 2 by choosing another machines)

3. Edit the .cshrc file under your home directory and put the following :

```
# keychain for mpi app
if (-e /usr/bin/keychain) then
        keychain --nogui -q id_rsa
        set host=`uname -n`
        if (-f $HOME/.keychain/$host-csh) then
              source $HOME/.keychain/$host-csh
        endif
        if (-f $HOME/.keychain/$host-csh-gpg) then
              source $HOME/.keychain/$host-csh-gpg
        endif
endif
```

4. To test passwordless SSH login (from arthur),
   ```
   arthur>  ssh bach
   ```
   and type the passphrase. Exit the machine bach and login again. You won't be asked for the passphrase the second time.

5. To test MPI programs, use an editor to build the program mpi_hello.c (see below).

   Compile it:
   ```
   mpicc -o mpi_hello mpi_hello.c
   ```

   ~~Use an editor~~ to type the machine names into the file machinefile:
   ```
   arthur:1
   bach:1
   chopin:1
   degas:1
   ```
   → configuration of MPI

   Run it on a single machine:
   ```
   mpirun -n 2 ./mpi_hello
   ```
   Note that the parameter next to -n specifies the number of processes to spawn and distribute among nodes

   Run it among several machines specified in the file machinefile:
   ```
   mpirun -n 2 -f machinefile ./mpi_hello
   mpirun -n 8 -f machinefile ./mpi_hello
   ```

```
-------------------- mpi_hello.c -----------------

#include <stdio.h>
#include <mpi.h>
#includ <unistd.h>
int main(int argc, char** argv) {
   int myrank, nprocs;

   char hostname[256];
   gethostname(hostname, 256);

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   printf("Hello from %s processor %d of %d\n", hostname, myrank, nprocs);

   MPI_Finalize();
   return 0;
}
```

Use PID

mpi-procen id

2

1

2

print 1
print 2

simpleMPI.cpp

101

```
// MPI include
#include <mpi.h>

// System includes
#include <iostream>

using std::cout;
using std::cerr;
using std::endl;

// User include
#include "simpleMPI.h"

// Error handling macros
#define MPI_CHECK(call) \
   if((call) != MPI_SUCCESS) { \
      cerr << "MPI error calling \""#call"\"\n"; \
      my_abort(-1); }

// Host code
// No CUDA here, only MPI
int main(int argc, char *argv[]) {
   // Dimensions of the dataset
   int blockSize = 256;
   int gridSize = 10000;
   int dataSizePerNode = gridSize * blockSize;

   // Initialize MPI state
   MPI_CHECK(MPI_Init(&argc, &argv));

   // Get our MPI node number and node count
   int commSize, commRank;
   MPI_CHECK(MPI_Comm_size(MPI_COMM_WORLD, &commSize));
   MPI_CHECK(MPI_Comm_rank(MPI_COMM_WORLD, &commRank));
```

no change

```
   // Generate some random numbers on the root node (node 0)
   int dataSizeTotal = dataSizePerNode * commSize;
   float *dataRoot = NULL;

   if (commRank == 0)  // Are we the root node?
```

this is the root node

```
   {
      cout << "Running on " << commSize << " nodes" << endl;
      dataRoot = new float[dataSizeTotal];
      initData(dataRoot, dataSizeTotal);
   }
```

Duplicate

Two diff array data Root A / data Root B

```
   // Allocate a buffer on each node
   float *dataNode = new float[dataSizePerNode];
```

dup? ✓

```
   // Dispatch a portion of the input data to each node
   MPI_CHECK(MPI_Scatter(dataRoot, dataSizePerNode, MPI_FLOAT, dataNode, dataSizePerNode,
                MPI_FLOAT, 0, MPI_COMM_WORLD));
```

Dup ✓ Root A / Root B

```
   if (commRank == 0) {
      // No need for root data any more
      delete [] dataRoot;
   }

   // On each node, run computation on GPU
   computeGPU(dataNode, blockSize, gridSize);
```

dup ✓

2

```
   // Reduction to the root node, computing the sum of output elements
   float sumNode = sum(dataNode, dataSizePerNode);
   float sumRoot;
```

SUM Function

use CPU → change

MPI-MAX

chang name

```
   MPI_CHECK(MPI_Reduce(&sumNode, &sumRoot, 1, MPI_FLOAT,
                 MPI_SUM, 0, MPI_COMM_WORLD));

   if (commRank == 0) {
      float average = sumRoot / dataSizeTotal;
      cout << "Average of square roots is: " << average << endl;
   }
```

No need average

Just MAX

```
   // Cleanup
   delete [] dataNode;
   MPI_CHECK(MPI_Finalize());

   if (commRank == 0) {
      cout << "PASSED\n";
   }
   return 0;
}

// Shut down MPI cleanly if something goes wrong
void my_abort(int err)
{
   cout << "Test FAILED\n";
   MPI_Abort(MPI_COMM_WORLD, err);
}
```

simpleMPI.cu

```
#include <iostream>
using std::cerr;
using std::endl;

#include "simpleMPI.h"

// Error handling macro
#define CUDA_CHECK(call) \
   if((call) != cudaSuccess) { \
      cudaError_t err = cudaGetLastError(); \
      cerr << "CUDA error calling \""#call"\", code is " << err << endl; \
      my_abort(err); }

// Device code
// Very simple GPU Kernel that computes square roots of input numbers
__global__ void simpleMPIKernel(float *input, float *output) {
   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   output[tid] = sqrt(input[tid]);
}
```

predefined variable

You need block # so you know which block you are

// give the position in the array

modify

```
// Initialize an array with random data (between 0 and 1)
void initData(float *data, int dataSize) {
   for (int i = 0; i < dataSize; i++) {
      data[i] = (float)rand() / RAND_MAX;
   }
}

// CUDA computation on each node
// No MPI here, only CUDA
void computeGPU(float *hostData, int blockSize, int gridSize) {
   int dataSize = blockSize * gridSize;

   // Allocate data on GPU memory
   float *deviceInputData = NULL;
   CUDA_CHECK(cudaMalloc((void **)&deviceInputData, dataSize * sizeof(float)));
```

A → dup ✓

B

```
   float *deviceOutputData = NULL;
   CUDA_CHECK(cudaMalloc((void **)&deviceOutputData, dataSize * sizeof(float)));
```

→ dup X

IN

IN

OUT

```
   // Copy to GPU memory
   CUDA_CHECK(cudaMemcpy(deviceInputData, hostData, dataSize * sizeof(float), cudaMemcpyHostToDevice));
```

→ Transfer from CPU To GPU

```
   // Run kernel
   simpleMPIKernel<<<gridSize, blockSize>>>(deviceInputData, deviceOutputData);
```

(A, B, Data Big - ?

```
   // Copy data back to CPU memory
   CUDA_CHECK(cudaMemcpy(hostData, deviceOutputData, dataSize *sizeof(float), cudaMemcpyDeviceToHost));
```

→ dup ✓

GPU

```
   // Free GPU memory
   CUDA_CHECK(cudaFree(deviceInputData));
   CUDA_CHECK(cudaFree(deviceOutputData));
}

float sum(float *data, int size) {
   float accum = 0.f;

   for (int i = 0; i < size; i++) {
      accum += data[i];
   }
   return accum;
}
```

SUMATION

Change to Max finder

cuda thread

__ global __ void simple MPI Kernel (

(_ A , _ B , _ out)

OUT