145

CIS 620 Project 4 Fall 2018

(Due: Dec. 5)

Project Description

The purpose of this project is to write a distributed application using the Remote Procedure Call (RPC) protocol the pthreads library, and the CUDA toolkit. In this project you are expected to use the ONC RPC Protocol to distribute computation from a Linux workstation to two Linux workstations with the lowest two loads.

This project can be divided into two parts. The client program running on a workstation provides the interface to the users. The servers running on the pre-selected five Linux workstations provide the machine load reporting service and another two computational services. You are required to follow the eight steps of RPC distributed application implementation discussed in the class and use the protocol compiler rpcgen to implement the client/server programs.

The client program keeps a list of five available Linux workstations in our lab. It firstly invokes a remote procedure `getload()` to collect the load average (over the last 1 minute) from each workstation in the list and selects two workstations with the lowest load averages. Then, based on the command line option (-gpu or -lst), the client invokes the computation function `sumqroot_gpu()` or `sumqroot_lst()` on the selected two workstations. Note that each workstation will get equal amount of data size. Finally, the client adds the results from the two servers and prints it out.

Two examples are shown below. The first example, with the command line arguments -gpu N M S1 S2, computes the sum of the quadruple root of an array. The whole array has 2^N (i.e. 2^{20}) elements. Each of the selected servers will initialize half of the array elements using the exponential distribution with the mean value M (i.e. 5). The two servers will use the values S1 and S2, respectively (i.e. 17 and 23), as the initial seed. The computation is executed on the machines chopin and degas with their GPUs. The client combines the results from the two servers and prints it out. The second example calculates the sum of the quadruple root of the doubles 5.0, 7.0, 19.0, and 23.0 stored in the file `datafile`. The task is distributed to the machines arthur and degas. Each server gets a linked list of doubles from the client.

```
haydn % ldshr -gpu 20 5 17 23
arthur: 2.30  bach: 3.3  brahms: 3.8  chopin: 0.50  degas: 1.27
(executed on chopin and degas)
[... result...]

haydn % ldshr -lst datafile
arthur: 1.30  bach: 5.5  brahms: 2.8  chopin: 2.27  degas: 0.9
(executed on arthur and degas)
[... result...]
```

In your client program, you need to use the POSIX threads to mask the communication latency. That is, create a thread when you make a remote procedure call.

The server provides three services (i.e functions). To get the load average (over the last 1 minute) on Linux, you can call the system function `getloadavg()`. The function `sumqroot_gpu()` firstly initializes the very large array using the values N, M, and S1 (or S2) passed from the client. It then launches a kernel function on GPU to calculate the quadruple root of each element in the array. Next, it invokes the reduction kernel function on GPU to get the summation of the quadruple roots and returns the value back to the client. You are required to modify the code in `reduction_kernel.cu` discussed in the class to implement above. Similarly, the function `sumqroot_lst()` gets a linked list of doubles from the client. It then utilizes higher-order functions (i.e. `map()` along with a formula function and `reduce()` along with a summation function) to compute the summation of the quadruple roots in the list.

Note that you should run the server program on each of the five Linux workstations before you start the client.

Turnin

Each group (at most two students) needs to submit this project using the following turnin command:

```
turnin -c cis620s -p proj4 makefile ldshr.x ldshr.c ldshr_svc_proc.c reduction.cu
```

You also need to hand in a hard-copy document which includes the description of your code, data structures, and experiences in debugging. The document should be typed. The cover page should contain your names, pictures, and your login-id you turnined.

cp ~cs620s/pub/reduction_kernel.cu

141

```
/*
 To compile:
   nvcc -arch=sm_60 reduction_kernel.cu
 To run with the array size 2^20, expo dist mean 5, and init seed 17:
   ./a.out 20 5 17
 */
#include <stdio.h>
#include <stdlib.h>
template<class T>
struct SharedMemory
{  __device__ inline operator       T *() {
      extern __shared__ int __smem[];
      return (T *)__smem;
   }
   __device__ inline operator const T *() const {
      extern __shared__ int __smem[];
      return (T *)__smem;
   }
};
```

Too large

sum=sum+ a[1]

this is ignored do ~~to~~ precision

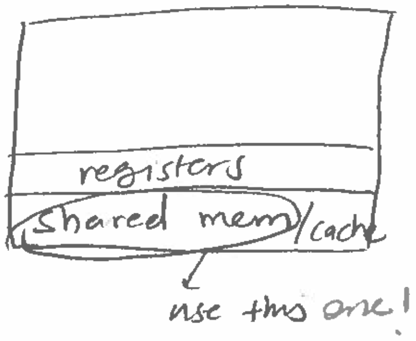
solution put the loss precision into a seprat variable and add in once the value is large enough.

```
template<class T>
T reduceCPU(T *data, int size) {
   T sum = data[0];
   T c = (T)0.0;

   for (int i = 1; i < size; i++){
      T y = data[i] - c;
      T t = sum + y;
      c = (t - sum) - y;
      sum = t;
   }
   return sum;
 }
```

CPU sequential algorithm

block of thread



```
 __global__ void
~~reduce(double~~ *g_idata, double *g_odata, unsigned int n) {
   double *sdata = SharedMemory<double>();

   // load shared mem
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   sdata[tid] = (i < n) ? g_idata[i] : 0;

   __syncthreads();

   // do reduction in shared mem
   for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
      if (tid < s) {
         sdata[tid] += sdata[tid + s];
      }
      __syncthreads();
   }

   // write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
 }

 // CUDA Runtime
 #include <cuda_runtime.h>

 #define checkCudaErrors(ans) { gpuAssert((ans), __FILE__, __LINE__); }
 inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true) {
   if (code != cudaSuccess)  {
      fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
      if (abort) exit(code);
```

in GPU side

load to SM

thread id

Block Dim / 2

next run → shift one bit right → $S = S/2$

$128/2 = 6$'

$\frac{64}{2} = 3$?

⋮

```
57        }
58    }

59    int main(int argc, char **argv){
```

double reduction(N,M,S)

```
60          int n = atoi(argv[1]);
```

N

```
61          int mean = atoi(argv[2]);
```

M

```
62          int seed = atoi(argv[3]);
```

S

```
63          int size = 1<<n;    // number of elements to reduce
64          int maxThreads = 256;  // number of threads per block

65          // create random input data on CPU
66          unsigned int bytes = size * sizeof(double);

67          double *h_idata = (double *) malloc(bytes);

68          srand48(seed);
69          for (int i=0; i<size; i++) {
70                // h_idata[i] = 1.0; // for testing
71                // expo dist with mean 5.0
72                h_idata[i] = -mean * log(drand48());
73          }

74          int numBlocks = size / maxThreads;
75          int numThreads = size;

76          int smemSize = maxThreads * sizeof(double);

77          // allocate mem for the result on host side
78          double *h_odata = (double *) malloc(numBlocks*sizeof(double));

79          // allocate device memory and data
80          double  *d_idata = NULL;
81          double *d_odata = NULL;

82          checkCudaErrors(cudaMalloc((void **) &d_idata, bytes));
83          checkCudaErrors(cudaMalloc((void **) &d_odata, numBlocks*sizeof(double)));

84          // copy data directly to device memory
85          checkCudaErrors(cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice));
```

tid = ○×○+○

a[tid] = sqrt(sqrt(a[tid]))

out the sar as in

85.5: mapqroot<<< >>> (in, out)

do quadroot first

```
86          reduce<<<numBlocks,maxThreads,smemSize>>>(d_idata, d_odata, numThreads);
```

reduce function

generate intermediate array

```
87          int s=numBlocks;
```

Now keep reducing ✓

```
88          while (s > 1) {
89             reduce<<<(s+maxThreads-1)/maxThreads,maxThreads,smemSize>>>(d_odata, d_odata, s);
90             s = (s+maxThreads-1)/maxThreads;
91          }
```

Keep calling reduce function

```
92          checkCudaErrors(cudaMemcpy(h_odata, d_odata, sizeof(double), cudaMemcpyDeviceToHost));

93          printf("GPU sum : %f\n\n", h_odata[0]);

94          checkCudaErrors(cudaFree(d_idata));
95          checkCudaErrors(cudaFree(d_odata));

96          double cpu_result = reduceCPU<double>(h_idata, size);
97
98          printf("CPU sum : %f\n", cpu_result);
99
100       return true;
101   }
```