Scientific Computing
Course SCICOMP 9502B

Assignment 4

GPU – Poisson Equation using Conjugate Gradient Method

Instructor

Dr. Colin Denniston

Submitted by

Tuntun Gaurav

Chemical and Biochemical Engineering

Introduction

The Conjugate gradient algorithm as explained in previous problems was solved using GPU using the CUDA library. Advantage of using GPU is vastly dependent on the size of the grid size as a number of threads perform the calculation at the same time. Here, the grid size has been carried from 50 to 2000 and a huge difference in time for calculation has been observed for the case of higher grid sizes as was expected. To solve the conjugate gradient algorithm using CUDA, first a CUDA qualifier is used to identify as a CUDA kernel. Further, shared memory is declared with minimum size requirements. These requirements are then checked with the device available. In the main function, the arrays and variables are declared for both the host and the device. After checking for error, the best possible CUDA device is selected. The boundary conditions are then filled in with the corner values. Once, this is done, the cublasSetVector is set to begin the GPU calculations. A handler is created as CUBLAS context to be used to perform inbuilt functions on the arrays. The first value of d is initialized as required and the main loop begins there after as per the algorithm. Ad is calculated using GPUs for the grid and used to calculate the lambda and the variables are substituted values using the cublasSaxpy and cublasSdot functions. The iteration was performed till the required level of accuracy is achieved and the results are printed.

Results

The code was compiled and run in the developer mode with the commands as below:

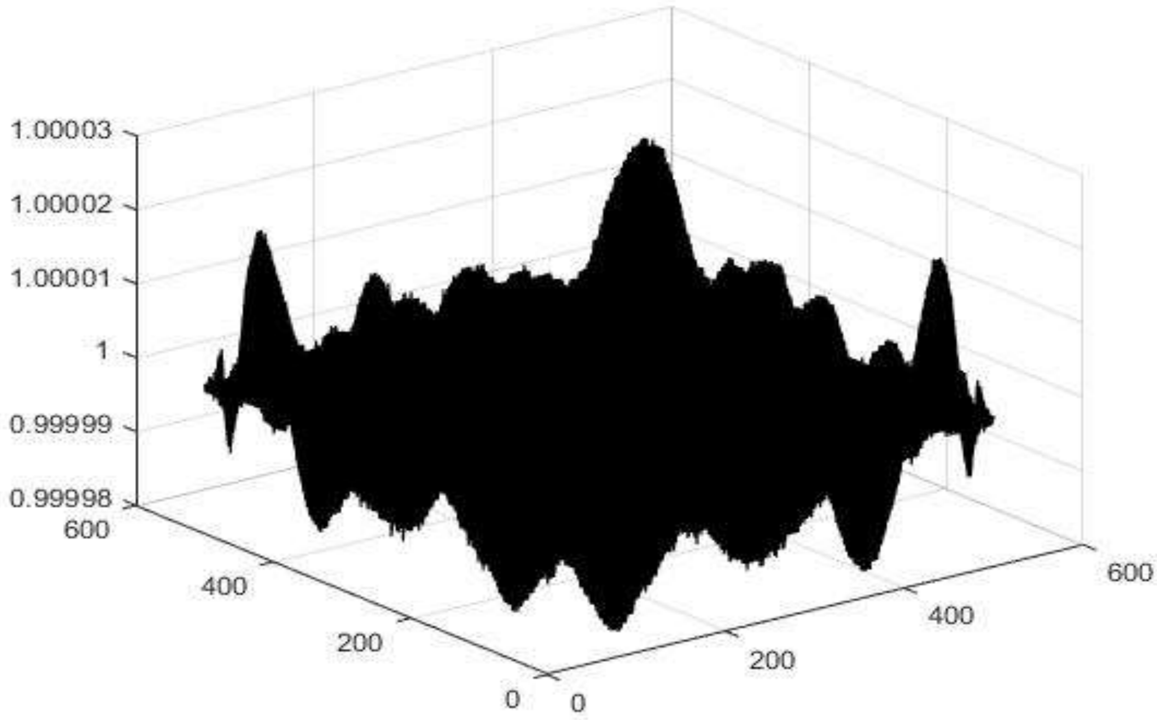
comp:

```
nvcc -I/opt/sharcnet/cuda/7.5.18/samples/common/inc ./cg_gpu.cu -lcublas
```

run:

```
sqsub -q gpu --gpp=1 -r 10m -o CUDA_TEST ./a.out
```

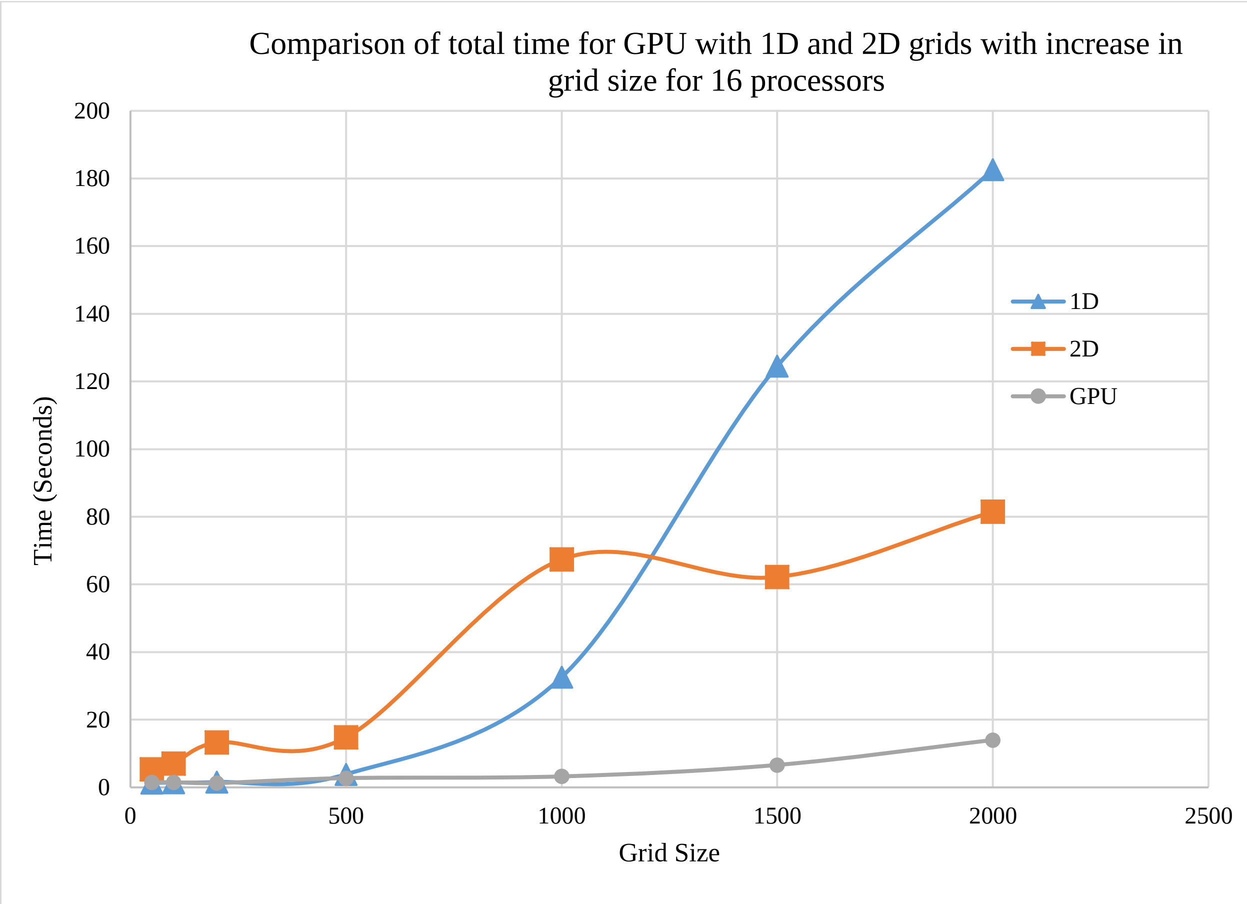
Figure 1 shows the contour for the calculation done using GPUs for 500 grid size.



The program was used to solve for a variety of grid sizes. The respective error and time taken are given in Table 1.

Grid Size	Error	Time (Seconds)
50	0.000095	1.45
100	0.000089	1.53
200	0.00009	1.27
500	0.000099	2.74
1000	0.000099	3.24
1500	0.000099	6.63
2000	0.000099	14

A comparison of GPU speed with those of 1D and 2D MPI using 16 processors for a range of grid sizes each from 50 – 2000 are given in Figure 2. As evident GPU takes very less time for same calculations as 16 processors even with a 2D grid. Also, this difference increases with the grid size.



Appendix

```
#include <stdio.h>
#include <stdlib.h>
#include <cassert>
#include "time.h"
#include <math.h>

//Including CUDA library
#include <cuda_runtime.h>
#include "cublas_v2.h"



// Including CUDA library functions
#include <helper_functions.h>
```

```
#include <helper_cuda.h>

#define tol  0.0001
#define MAXITER 1024

const int BLOCK_SIZE_X = 500;  //interior + 2 boundary layers, ie. 16x16 + boundary
layers=18x18
const int BLOCK_SIZE_Y = 500;

float **matrix(int m, int n)
{	int i;
	float **ptr;
	ptr = (float **)calloc(m, sizeof(float *));
	ptr[0] = (float *)calloc(m * n, sizeof(float));
	for (i = 1; i < m ; i++)
		ptr[i] = ptr[i - 1] + n;
	return (ptr);
}

__global__ void AD_gpu(float* u, float* d_d, int ArraySizeX, int ArraySizeY, float h)
{
	int tx = threadIdx.x;
	int ty = threadIdx.y;
	int bx = blockIdx.x * (BLOCK_SIZE_X - 2);
	int by = blockIdx.y * (BLOCK_SIZE_Y - 2);
	int x = tx + bx;
	int y = ty + by;

	__shared__ float d_sh[BLOCK_SIZE_X][BLOCK_SIZE_Y];

	d_sh[tx][ty] = d_d[x + y * ArraySizeX];

	__syncthreads();

	if (tx > 0 && tx < BLOCK_SIZE_X - 1 && ty > 0 && ty < BLOCK_SIZE_Y - 1) {
		u[x + y * ArraySizeX] = -(d_sh[tx + 1][ty] + d_sh[tx - 1][ty] + d_sh[tx][ty + 1] +
d_sh[tx][ty - 1] - 4.0f * d_sh[tx][ty]);
	}

}

void myGpuInfo(int argc, char** argv, int dev, cudaDeviceProp deviceProp) {
	// Device queries and error checks
	char msg[256];
```

```
        SPRINTF(msg, "Total amount of global memory: %.0f MBytes (%llu bytes)\n",
              (float)deviceProp.totalGlobalMem / 1048576.0f, (unsigned long long)
deviceProp.totalGlobalMem);
        printf("%s", msg);

        printf(" (%2d) Multiprocessors, (%3d) CUDA Cores/MP: %d CUDA Cores\n",
              deviceProp.multiProcessorCount,
              _ConvertSMVer2Cores(deviceProp.major, deviceProp.minor),
              _ConvertSMVer2Cores(deviceProp.major, deviceProp.minor) *
deviceProp.multiProcessorCount);

        printf(" Total amount of shared memory per block: %lu bytes\n",
deviceProp.sharedMemPerBlock);

        printf("  Maximum number of threads per block: %d\n",
deviceProp.maxThreadsPerBlock);
        printf("  Max dimension size of a thread block (x,y): (%d, %d)\n",
              deviceProp.maxThreadsDim[0],
              deviceProp.maxThreadsDim[1]);
        printf("  Max dimension size of a grid size (x,y): (%d, %d)\n",
              deviceProp.maxGridSize[0],
              deviceProp.maxGridSize[1]);
  }

  main(int argc, char** argv)
  {
        float *u_h, *r_h, *d_h, *x_h;    // pointers to host memory
        float *u, *r_d, *r_d_old, *d_d, *x_d;    // pointers to device memory
        float **x, **r;
        float *init;
        int ArraySizeX = 258; // Note these, minus boundary layer, have to be exactly divisible
by the (BLOCK_SIZE-2) here
        int ArraySizeY = 258;
        int size_side = ArraySizeX;
        size_t size = ArraySizeX * ArraySizeY * sizeof(float);
        int array_size = ArraySizeX * ArraySizeY;
        int itr;
        // int i, j, k;
        char str[20];
        // cudaError_t error ;
        cudaDeviceProp deviceProp;
        FILE *fp;

        /* Runtume arrays initlized to zero */
```

```
	x = matrix(ArraySizeX, ArraySizeY);
	r = matrix(ArraySizeX, ArraySizeY);

	// Time variable
	clock_t t_global_start, t_global_end;
	double t_global;

	t_global_start = clock();

	int deviceCount = 0;
	cudaError_t error_id = cudaGetDeviceCount(&deviceCount);

	if (error_id != cudaSuccess)
	{
		printf("cudaGetDeviceCount returned %d\n-> %s\n", (int)error_id,
cudaGetErrorString(error_id));
		printf("Result = FAIL\n");
		exit(EXIT_FAILURE);
	}

	// Pick the best possible CUDA device
	int dev = findCudaDevice(argc, (const char **)argv);
	cudaSetDevice(dev);
	cudaGetDeviceProperties(&deviceProp, dev);
	cublascubhan_t cubhan; // CUBLAS context

	//Allocate arrays on host and initialize to zero
	u_h = (float *)calloc(ArraySizeX * ArraySizeY, sizeof(float));
	x_h = (float *)calloc(ArraySizeX * ArraySizeY, sizeof(float));
	d_h = (float *)calloc(ArraySizeX * ArraySizeY, sizeof(float));
	init = (float *)calloc(ArraySizeX * ArraySizeY, sizeof(float));
	r_h = (float *)calloc(ArraySizeX * ArraySizeY, sizeof(float));
	// d = (float *)calloc(ArraySizeX*ArraySizeY,sizeof(float));

	//Allocate arrays on device
	cudaMalloc((void **) &u, size);
	cudaMalloc((void **) &r_d, size);
	cudaMalloc((void **) &r_d_old, size);
	cudaMalloc((void **) &d_d, size);
	cudaMalloc((void **) &x_d, size);


	/* fill in compute block beside boundaries as available */
	for (int i = 1; i < size_side - 1; i++) {
```

```
            r[i][1] -= 1;
            r[i][size_side - 2] -= 1;
            x[i][0] = 1;
            x[i][size_side - 1] = 1;
        }
        for (int j = 1; j < size_side - 1; j++) {
            r[1][j] -= 1;
            r[size_side - 2][j] -= 1;
            x[0][j] = 1;
            x[size_side - 1][j] = 1;
        }
    //Fill in the corners
        x[0][0] = x[0][size_side - 1] = x[size_side - 1][0] =
                                    x[size_side - 1][size_side - 1] = 1;


        //GPU Calculations
        cublasSetVector(array_size, sizeof(*d_h), x_h, 1, x_d , 1);//setting the cublasSetVector
        cublasSetVector(array_size, sizeof(*u_h), u_h, 1, u , 1);
        cublasSetVector(array_size, sizeof(*r_h), r_h, 1, r_d , 1);

        int nBlocksX = (ArraySizeX - 2) / (BLOCK_SIZE_X - 2);
        int nBlocksY = (ArraySizeY - 2) / (BLOCK_SIZE_Y - 2);

        dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
        dim3 dimGrid(nBlocksX, nBlocksY);

    // initialize CUBLAS cubhanr
        cublasCreate(&cubhan);

        /* Compute first delta */
        for (int i = 0; i < array_size; i++)
                d_h[i] = -r[0][i];

        cublasSetVector(array_size, sizeof(*d_h), d_h, 1, d_d , 1);

        /* Compute first delta */
        float delta;
        cublasSdot(cubhan, array_size, r_d, 1, r_d, 1, &delta);

        float lamda = 0.0;
        float minus = -1.0;
        float alpha = 0.0;
        float olddelta = 0.0;
```

```
    float temp_dot_du = 0.0;
    float error = 0.0;

    //   Main loop
    for (itr = 0; itr < MAXITER; itr++) {

        /* do Ad for interior of compute block excluding beside boundaries */
        AD_gpu <<< dimGrid, dimBlock>>>(u, d_d, ArraySizeX, ArraySizeY, 1);

        cublasSdot(cubhan, array_size, d_d, 1, u, 1, &temp_dot_du);


        lamda = delta / temp_dot_du;

        // x(m+1) = x(m) + lamda(m) * d(m)
        cublasSaxpy(cubhan, array_size, &lamda, d_d, 1, x_d, 1);

        // r(m+1) = r(m) + lamda(m) * u
        cublasSaxpy(cubhan, array_size, &lamda, u, 1, r_d, 1);

        olddelta = delta;
        cublasSdot(cubhan, array_size, r_d, 1, r_d, 1, &delta);

        // Compute error and check for tolerance
        cublasSnrm2(cubhan, array_size, r_d, 1, &error);

        if (error < tol)
            break;

/* find global error */
        alpha = delta / olddelta;

        cublasScopy(cubhan, array_size, r_d, 1, r_d_old, 1);

        cublasSscal(cubhan, array_size, &minus, r_d_old, 1);

        cublasSaxpy(cubhan, array_size, &alpha, d_d, 1, r_d_old, 1);
        cublasScopy(cubhan, array_size, r_d_old, 1, d_d, 1);

    }

    printf("It took %d iterations and error is %f \n", itr, error);
    cublasGetVector(array_size, sizeof(float), x_d, 1, x_h, 1);
```

```
	t_global_end = clock();
	t_global = (double)(t_global_end - t_global_start) / CLOCKS_PER_SEC;

	/* Output results */
	sprintf(str, "results_blk_%d.txt", BLOCK_SIZE_X);
	fp = fopen(str, "wt");
	fprintf(fp, "%d \n", BLOCK_SIZE_X);
	fprintf(fp, "%lf \n", t_global);
	fclose(fp);

	// Output results
	fp = fopen("solution.txt", "wt");
	for (int i = 0; i < ArraySizeX; i++) {
		for (int j = 0; j < ArraySizeY; j++)
			fprintf(fp, " %f", x_h[j * ArraySizeX + i]);
		fprintf(fp, "\n");
	}
	fclose(fp);

	// free;
	cublasDestroy(cubhan); // CUBLAS handler free
	free(u_h);
	free(r_h);
	free(x_h);
	free(d_h);
	free(init);
	cudaFree(u);
	cudaFree(r_d);
	cudaFree(x_d);
	cudaFree(d_d);
}
```