Scientific Computing
Course SCICOMP 9502B

Assignment 5

OpenCL and Stochastic Processes

Instructor

Dr. Colin Denniston

Submitted by

Tuntun Gaurav

Chemical and Biochemical Engineering

1. **OpenCL**

The conjugate gradient algorithm for solving the Poisson equation was programmed using OpenCl libraries. OpenCL libraries from CUDA database in sharcnet were included. The OpenCL file is saved in a different file referenced by a character saved as a character pointer. This .cl program file contains the program to execute OpenCL and is read and saved as a character pointer in the program. In the main function the variables are declared and allocated only for the host. The boundary condition is given as the last programs along with the corner points. The d array is initialized as –r. The number of platforms for CUDA calculation are seen through the function clGetPlatformIDs and space is allocated. Now, the devices are retrieved, allocated space and filled. A context is created with the GPU device and create a command queue to associate. The buffer memory objects are allocated. The computation on GOU begins with copying the data from host to device. A program is created and built for the device. Among the two kernels which are calculated using GPU the first is the matrix multiplication Ad. The kernel fro Ad multiplication is created and the input and output variables are associated. Now the saxpy kernel is also created and the input and output variables are associated. The main loop for calculation begins for the number of maxiter steps and the calculations are performed using the Ad kernel for matrix multiplication. Further, calculating lambda, the saxpy inbuilt cblas function is used to perform the calculations as per the conjugate gradient algorithm. The program runs for maxiter iterations. The data is retrieved from device to host and stored in an array. And the results are printed from there on. The total time is calculated using the clock function. The program is run for 2*N to perform the same amount of calculations as the MPI programs.

Results

The sample output for a run of 100 grid size is as given:

Grid Size = 100
Number of iterations=144 and Error = 0.000092
Time = 0.020000s econds

Table 1 contains the time taken for grid sizes ranging from 50 to 2000.

Grid Size	Error	Time (Seconds)
50	0.000092	0
98	0.000092	0.02
210	0.16	0.000099
498	0.000095	1.66
994	0.000099	20.03
1506	0.00011	67.54
2002	0.00275	113.59

Figure 1. show the error contour with grid points.

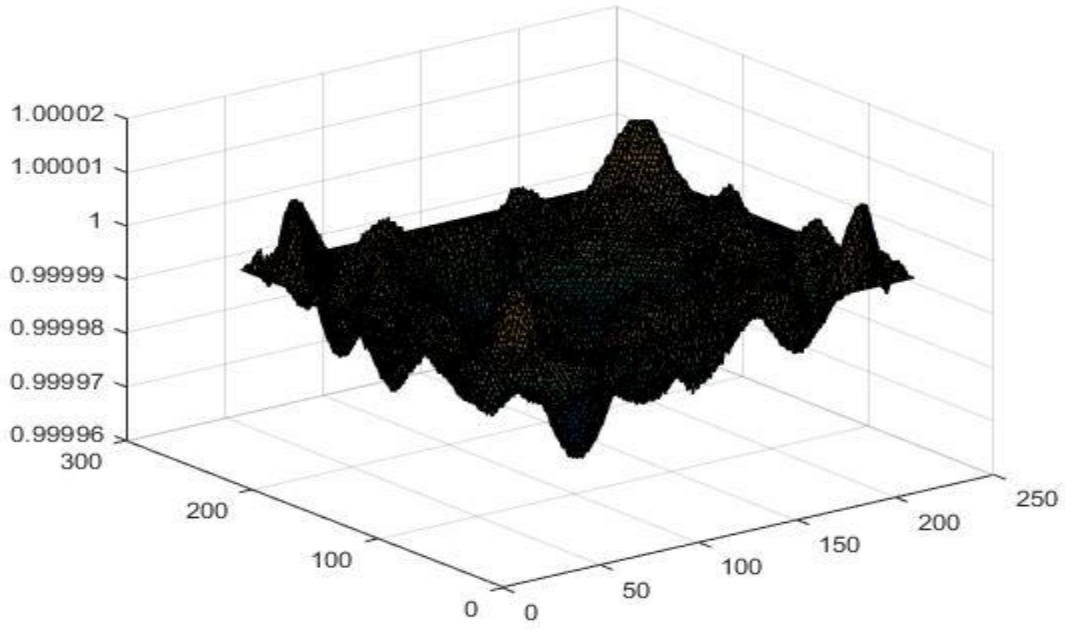
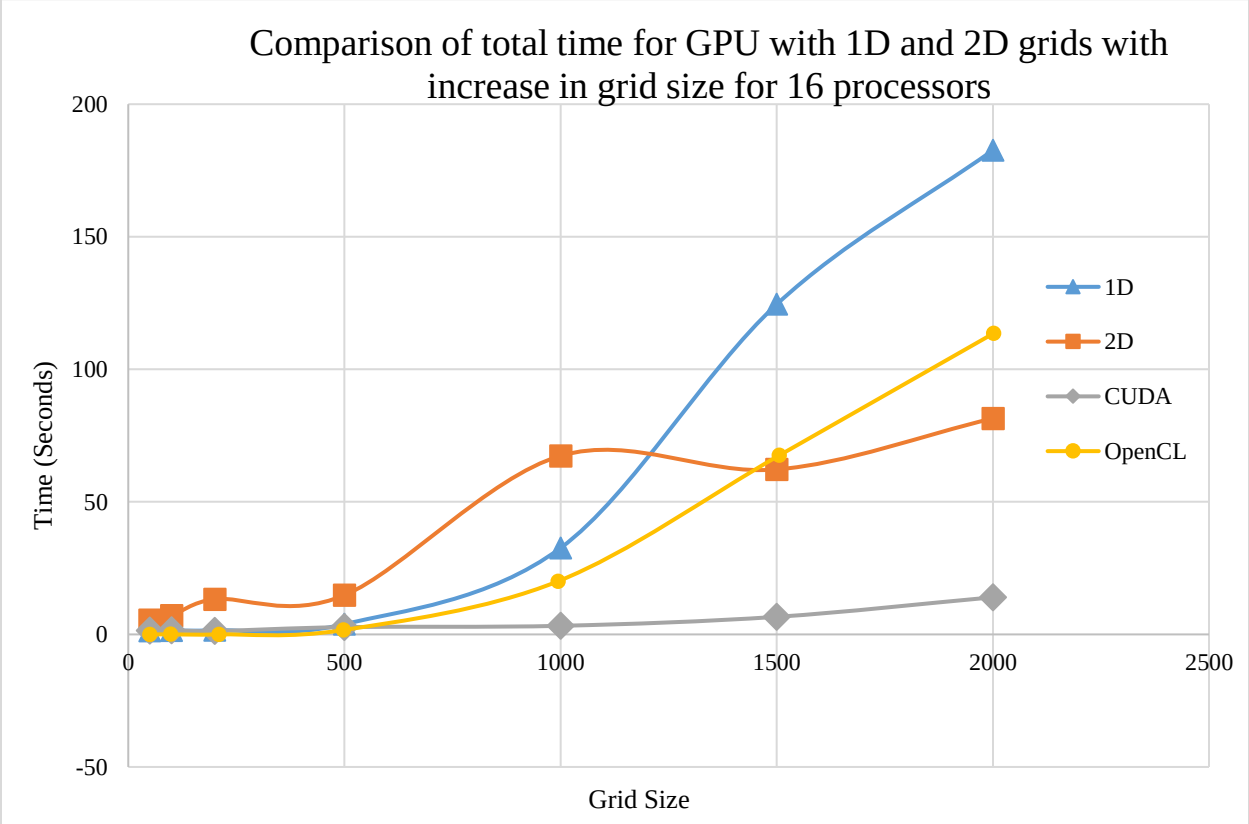


Figure 2 shows comparison of time taken with grid size in comparison with GPU using CUDA and MPI with 2D and 1D processor arrangements with 16 processors



2. Monte Carlo Simulation

The problem given with the probability of choice of door in choice a and b was calculated using the Monte Carlo simulation. The program was run using CUDA library. For the given case, the number of trails were 10000. The probability of choice a winning and choice b winning was determined by running the simulation for a large number of trials as is done for a Monte Carlo Algorithm. Based on the results a mean probability was determined for the cases won using either of the choices. Further, the given size of 10000 was divided into sample sizes of n = 1, 2, 4, 8, 16, 32, 64, 128, 256. To demonstrate that the size of sample size does not effectively affects the probabilities pa and pb, the mean error was derived for all the sizes of n. The variance and standard deviation which was plotted as function of sample size n. The random function inbuilt in cuda curand is used. After including the libraries and defining the global kernels for both random number generation and montecarlo simulations. The montecarlo simulation kernel takes in random value and makes the choice based on another random value for winning in choice a and choice b. One door is eliminated after the choice a is made. The main function includes the loop to select a variety for sample size and calculate the mean of the probability. There are 10000 trial runs for each sample and the probability of choice a winning or choice b winning is calculated as a percentage of the wins in the random trials. The array is initialized with host array and copied to the CUDA device using cudaMemcpy function. The calculation is then performed by running the kernels setupkernel and montecarlo kernel. The data is copied back to host from device using cudaMemcpyDeviceToHost in cudaMemcpy function. The probability is calculated as the fraction of winnings from choice a and those from winnings of choice b are printed for this sample size and the next loop is executed for the next sample size. The whole set of results are printed in the file as a set of mean of probabilities.

Results

The application was run for a 10000 trials for a sample size of 2. The program was compiled and run using the following commands:

Compiling:

nvcc -I/opt/sharcnet/cuda/7.5.18/samples/common/inc ./montecarlo_cuda.cu –lcublas

Run

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

The results are as given:

Number of trials was = 10000
Winnings with choice a 2556
times
Pa = 0.255600
Winnings with choice b 3743
times
Pb = 0.374300

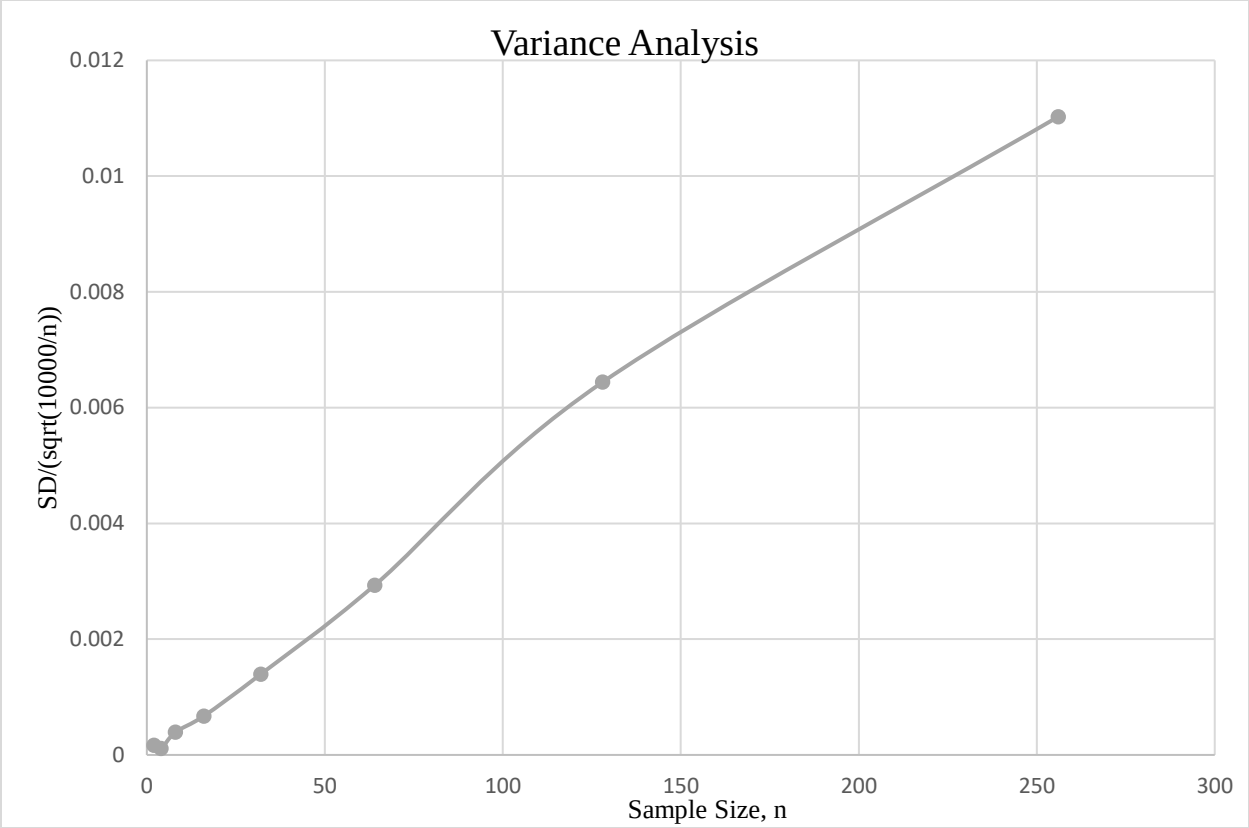
Number of trials was = 5000
Winnings with choice a 1275 times
Pa = 0.255000
Winnings with choice b 1903 times
Pb = 0.380600

Number of trials was = 5000
Winnings with choice a 1249 times
Pa = 0.249800
Winnings with choice b 1812 times
Pb = 0.362400

To study the variation of mean and hence the variation of error in the probability, the trials were divided in sample sizes of n = 1, 2, 4, 8, 16, 32, 64, 128 and 256. Table 1 gives the mean, variance and standard deviation of for these values of n.

Sample Size	Variance	Standard Deviation
2	0.011879	0.003111
4	0.005575	0.004542
8	0.01385	0.016272
16	0.016781	0.022864
32	0.024663	0.025912
64	0.036642	0.042428
128	0.056916	0.062036
256	0.068907	0.083204

Figure 1 shows the variation of Standard deviation/(sqrt(1000/n)) with n. It can be seen that for huge changes in sample size the standard deviation term varies. Hence, for a given range at for a range of sample sizes would not be affected adversely with sample size.



Appendix

1. OpenCL with conjugate gradient

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

// For the CUDA runtime routines (prefixed with "cuda_")
#include <math.h>
#include "mkl.h"

// OpenCL library
#include <CL/cl.h>

#define tol  0.0001
#define MAXITER 3000

//Blocksize = 16+ 2 boundary layers
const int BLOCK_SIZE_X = 18;
const int BLOCK_SIZE_Y = 18;

// This function reads in a text file and stores it as a char pointer
```

```
char* readSource(char* kernelPath) {
	cl_int status;
	FILE *fp;
	char *source;
	long int size;
	printf("OpenCL program file is: %s\n", kernelPath);
	fp = fopen(kernelPath, "rb");
	if (!fp) {
		printf("Could not open kernel file\n");
		exit(-1);
	}
	status = fseek(fp, 0, SEEK_END);
	if (status != 0) {
		printf("Error seeking to end of file\n");
		exit(-1);
	}
	size = ftell(fp);
	if (size < 0) {
		printf("Error getting file position\n");
		exit(-1);
	}
	rewind(fp);
	source = (char *)malloc(size + 1);
	int i;
	for (i = 0; i < size + 1; i++) {
		source[i] = '\0';
	}
	if (source == NULL) {
		printf("Error allocating space for the kernel source\n");
		printf("Error allocating space for the kernel source\n");
		exit(-1);
	}
	fread(source, 1, size, fp);
	source[size] = '\0';
	return source;
}

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);
}

void chk(cl_int status, const char* cmd) {

	if (status != CL_SUCCESS) {
		printf("%s failed (%d)\n", cmd, status);
		exit(-1);
	}
}

int main(void)
{
	float *u_h, *r_h, *d_h, *x_h;    // pointers to host memory
	float **x, **r;
	float h = 1.0;
	int ArraySizeX = 502; // Note these, minus boundary layer, have to be exactly divisible by the
(BLOCK_SIZE-2) here
	int ArraySizeY = 502;
	int bytes_per_float = 4;
	int size_side = ArraySizeX;
	size_t size = ArraySizeX * ArraySizeY * sizeof(float);
	int array_size = ArraySizeX * ArraySizeY;
	int thread_size = BLOCK_SIZE_X * BLOCK_SIZE_Y;
	int itr,  i, j, k;
	char str[20];
	FILE *fp;

	float lamda, minus, alpha, delta, olddelta, temp_dot_du, error;

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

	const char* programSource = readSource("gpuOpenCL.cl");

	/* create arrays and initialize to all zeros */
	x = matrix(ArraySizeX, ArraySizeY);
	r = matrix(ArraySizeX, ArraySizeY);

	//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));
```

```
	r_h = (float *)calloc(ArraySizeX * ArraySizeY, sizeof(float));

	/* Subtract off b from a particularly boring set of boundary conditions where x=1 on the boundary
*/
	for (i = 1; i < size_side - 1; i++) {
		r[i][1] -= 1;
		r[i][size_side - 2] -= 1;
		x[i][0] = 1;   /* we won't use the x on the boundary but this will be useful for the output at
the end*/
		x[i][size_side - 1] = 1;
	}
	for (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;
	}
	x[0][0] = x[0][size_side - 1] = x[size_side - 1][0] =
				x[size_side - 1][size_side - 1] = 1; /* fill in the corners */

	for (i = 0; i < array_size; i++)
	{
		r_h[i] = r[0][i];
		x_h[i] = x[0][i];
	}

	/* Fill in d */
	for (i = 0; i < array_size; i++)
	{
		d_h[i] = -r_h[i];
	}
	printf("Grid Size = %d\n", ArraySizeX);

	// Use this to check the output of each API call
	cl_int status;

	// Retrieve the number of platforms
	cl_uint numPlatforms = 0;
	status = clGetPlatformIDs(0, NULL, &numPlatforms);
	chk(status, "clGetPlatformIDs0");

	// Allocate enough space for each platform
	cl_platform_id *platforms = NULL;
	platforms = (cl_platform_id*)malloc(numPlatforms * sizeof(cl_platform_id));
```

```
// Fill in the platforms
status = clGetPlatformIDs(numPlatforms, platforms, NULL);
chk(status, "clGetPlatformIDs1");

// Retrieve the number of devices
cl_uint numDevices = 0;
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices);
chk(status, "clGetDeviceIDs0");

// Allocate enough space for each device
cl_device_id *devices;
devices = (cl_device_id*)malloc(numDevices * sizeof(cl_device_id));

// Fill in the devices
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, numDevices, devices, NULL);
chk(status, "clGetDeviceIDs1");

// Create a context and associate it with the devices
cl_context context;
context = clCreateContext(NULL, numDevices, devices, NULL, NULL, &status);
chk(status, "clCreateContext");

// Create a command queue and associate it with the device
cl_command_queue cmdQueue;
cmdQueue = clCreateCommandQueue(context, devices[0], 0, &status);
chk(status, "clCreateCommandQueue");

// allocate the buffer memory objects
cl_mem u_d, r_d, r_d_old, d_d, x_d;
u_d = clCreateBuffer(context, CL_MEM_READ_WRITE, size, NULL, &status);
chk(status, "clCreateBuffer");
r_d = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, &status);
chk(status, "clCreateBuffer");
r_d_old = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, &status);
chk(status, "clCreateBuffer");
d_d = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, &status);
chk(status, "clCreateBuffer");
x_d = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, &status);
chk(status, "clCreateBuffer");


//Perform computation on GPU
// Copy data from host to device
```

```
	status = clEnqueueWriteBuffer(cmdQueue, u_d, CL_FALSE, 0, size, u_h, 0, NULL, NULL);
	chk(status, "clEnqueueWriteBuffer");
	status = clEnqueueWriteBuffer(cmdQueue, r_d, CL_FALSE, 0, size, r_h, 0, NULL, NULL);
	chk(status, "clEnqueueWriteBuffer");
	status = clEnqueueWriteBuffer(cmdQueue, x_d, CL_FALSE, 0, size, x_h, 0, NULL, NULL);
	chk(status, "clEnqueueWriteBuffer");
	status = clEnqueueWriteBuffer(cmdQueue, d_d, CL_FALSE, 0, size, d_h, 0, NULL, NULL);
	chk(status, "clEnqueueWriteBuffer");

	// Create a program with source code
	cl_program program = clCreateProgramWithSource(context, 1, (const char**)&programSource,
NULL, &status);
	chk(status, "clCreateProgramWithSource");

	// build the compute program executablefor the device
	status = clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);

	if (status != CL_SUCCESS) {
		printf("clBuildProgram failed (%d)\n", status);

		// Determine the size of the log
		size_t log_size;
		clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG, 0, NULL,
&log_size);

		// Allocate memory for the log
		char *log = (char *) malloc(log_size);

		// Get the log
		clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG, log_size,
log, NULL);

		// Print the log
		printf("%s\n", log);

		exit(-1);
	}

	// Create the AD_gpu compute kernel
	cl_kernel kernel_AD;
	kernel_AD = clCreateKernel(program, "AD_gpu", &status);
	chk(status, "clCreateKernel");

	// Associate the input and output buffers with the kernel_AD
```

```
	status = clSetKernelArg(kernel_AD, 0, sizeof(cl_mem), &u_d);
	status |= clSetKernelArg(kernel_AD, 1, sizeof(cl_mem), &d_d);
	status |= clSetKernelArg(kernel_AD, 2, sizeof(int), &ArraySizeX);
	status |= clSetKernelArg(kernel_AD, 3, sizeof(int), &ArraySizeY);
	status |= clSetKernelArg(kernel_AD, 4, sizeof(float), &h);
	chk(status, "clSetKernelArg");

	// Set the work item dimensions

	size_t localWorkSize[2] = {BLOCK_SIZE_X, BLOCK_SIZE_Y};
	int nBlocksX = (ArraySizeX - 2) / (BLOCK_SIZE_X - 2);
	int nBlocksY = (ArraySizeY - 2) / (BLOCK_SIZE_Y - 2);

	// Global thread index has not direct equivalent in CUDA
	size_t globalWorkSize[2] = {nBlocksX * BLOCK_SIZE_X, nBlocksY * BLOCK_SIZE_Y};

	delta = cblas_sdot(array_size, r_h, 1, r_h, 1);


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


	t_global_start = clock();

	int nsteps;
    // Main Loop
	for (nsteps = 0; nsteps < MAXITER; nsteps++) {
    // Call kernel_AD with execution configuration
		status = clEnqueueNDRangeKernel(cmdQueue, kernel_AD, 2, NULL, globalWorkSize,
localWorkSize, 0, NULL, NULL);
		status = clEnqueueReadBuffer(cmdQueue, u_d, CL_TRUE, 0, size, u_h, 0, NULL,
NULL);

		temp_dot_du = cblas_sdot(array_size, d_h, 1, u_h, 1);
		lamda = delta / temp_dot_du;

		cblas_saxpy(array_size, lamda, d_h, 1, x_h, 1);
```

```
                cblas_saxpy(array_size, lamda, u_h, 1, r_h, 1);

                olddelta = delta;
                delta = cblas_sdot(array_size, r_h, 1, r_h, 1);

                error = sqrt(delta);
                if (error < tol)
                        break;

                alpha = delta / olddelta;
                for (i = 0; i < array_size; i++)
                        d_h[i] = -r_h[i] + alpha * d_h[i];
                status = clEnqueueWriteBuffer(cmdQueue, d_d, CL_FALSE, 0, size, d_h, 0, NULL,
NULL);

        }

        printf("Number of iterations= %d and Error = %f \n", nsteps, error);

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

        printf("Time= %f seconds\n", t_global);

    // Retrieve result from device and store in array
        status = clEnqueueReadBuffer(cmdQueue, u_d, CL_TRUE, 0, size, u_h, 0, NULL, NULL);
        chk(status, "clEnqueueReadBuffer");

        status = clEnqueueReadBuffer(cmdQueue, r_d, CL_TRUE, 0, size, r_h, 0, NULL, NULL);
        chk(status, "clEnqueueReadBuffer");



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

    //Cleanup
    // Free OpenCL resources
```

```
        clReleaseKernel(kernel_AD);
        clReleaseProgram(program);
        clReleaseCommandQueue(cmdQueue);
        clReleaseMemObject(u_d);
        clReleaseMemObject(r_d);
        clReleaseMemObject(r_d_old);
        clReleaseMemObject(x_d);
        clReleaseMemObject(d_d);
        clReleaseContext(context);

    // Free host resources
        free(u_h);
        free(r_h);
        free(x_h);
        free(d_h);
        free(platforms);
        free(devices);

        printf("Done everything \n");

        return 0;
    }
```

Kernel function: gpuOpenCL.cl

```
__kernel void AD_gpu(__global float* u_d, __global float* d_d, int ArraySizeX, int ArraySizeY, float h)
{
  /* Compute Ad= A*d using the auxiliary layer around the outside that is all zero for d to account for
boundary */
  int blk_size_x = get_local_size(0);
  int blk_size_y = get_local_size(1);
  int tx = get_local_id(0);
  int ty = get_local_id(1);
  int bx = get_group_id(0) * (blk_size_x - 2);
  int by = get_group_id(1) * (blk_size_y - 2);
  int x = tx + bx;
  int y = ty + by;

  __local float d_sh[18][18];

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

  barrier(CLK_LOCAL_MEM_FENCE);

  if (tx > 0 && tx < blk_size_x - 1 && ty > 0 && ty < blk_size_y - 1) {
```

```
   u_d[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]);
  }

}
```

2. Monte Carlo Simulation with Random numbers

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
#include <limits.h>
#include "time.h"


// CUDA libraries
#include <cuda.h>
#include <curand.h>          //CUDA random number library
#include <curand_kernel.h>


const int N = 10000;

// Seed the random number generator (call only once)

// Kernel to runon the CUDA device
__global__ void setup_kernel( curandState * state, unsigned long seed )
{
  int id = threadIdx.x + blockDim.x * blockIdx.x;
  curand_init ( seed, id, 0, &state[id] );
}

__global__ void montecarlo( curandState* globalState, int *win_a, int *win_b )
{
  int ind = threadIdx.x + blockDim.x * blockIdx.x;

  if ( ind < N ) {
    int count = 4;
    int i, win, choice, choice_b, eliminated;
    int Door[4] = {0};
    int remaining[2] = {0}; // Remaining two doors for choice b

    curandState localState = globalState[ind];
```

```
globalState[ind] = localState;

// Random value generated using built-in CUDA function

float r = curand_uniform( &localState );

//Conditions for the winning door selection

if (r < 0.25) {
  Door[0] = 1;
  win = 1;
}
else if (r > 0.25 && r < 0.5) {
  Door[1] = 1;
  win = 2;
}
else if ( r > 0.5 && r < 0.75) {
  Door[2] = 1;
  win = 3;
}
else  {
  Door[3] = 1;
  win = 4;
}

float m = curand_uniform( &localState );
// printf("m = %f \n", m);

if (m < 0.25) {
  choice = 1;
}
else if (m > 0.25 && m < 0.5) {
  choice = 2;
}
else if ( m > 0.5 && m < 0.75) {
  choice = 3;
}
else  {
  choice = 4;
}

for (i = 0; i < count; ++i)
{
  if (Door[i] == 0 && choice != (i + 1))  {
```

```
        eliminated = i + 1;
        break;
      }
    }

    int j = 0;
    for (i = 0; i < count; ++i)
    {
      if (eliminated != (i + 1) && choice != (i + 1))
      {
        remaining[j] = (i + 1);
        j++;
      }
    }


    // Choice of b
    float b_choice = curand_uniform( &localState );
    if (b_choice < 0.5) {
      choice_b = remaining[0];
    }
    else  {
      choice_b = remaining[1];
    }

    __syncthreads();

    // Choice a
    if (win == choice)
      win_a[ind] = 1;

    // Choice b
    if (win == choice_b)
      win_b[ind] = 1;
  }
}

template <typename T> // this is the template parameter declaration
T sum(T *a, int N)
{
  T sum = 0;
  for (int i = 0; i < N; ++i)
  {
    sum += a[i];
```

```
  }
  return sum;
}

int main(int argc, char** argv)
{
  int *win_a;
  int *win_b;   // Pointer to host & device arrays
  int group_n[9] = {1, 2, 4, 8, 16, 32, 64, 128, 256};
  char str[20];
  FILE *fp;
  int k;

  for (int l = 0; l < 9; ++l)
  {
    int trial_num = N / group_n[l];
    size_t size = sizeof(int) * trial_num;

    // int list_length = trial_num / group_n[0];
    float *Pa_array;
    float *Pb_array;

    // Device variables
    int *win_a_d;
    int *win_b_d;

    int blockdimx = 1024;
    int griddimx = ceil(trial_num / blockdimx) + 1;

    Pa_array = (float *)calloc(group_n[l], sizeof(float));
    Pb_array = (float *)calloc(group_n[l], sizeof(float));

      // Run kernel

    dim3 block(blockdimx, 1);
    dim3 grid(griddimx, 1);
    curandState* devStates;

    // Monte Carlo simulation

    for (int i = 0; i < group_n[l]; ++i)
      {
        int seed = (int)time(NULL) * i * l;
        win_a = (int *)calloc(trial_num, sizeof(int));
```

```
        win_b = (int *)calloc(trial_num, sizeof(int));

// Initialize host array and copy it to CUDA device

        cudaMalloc((void **)&win_a_d, size);
        cudaMalloc((void **)&win_b_d, size);
        cudaMalloc(&devStates, trial_num * sizeof( curandState ) );

        cudaMemcpy(win_a_d, win_a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(win_b_d, win_b, size, cudaMemcpyHostToDevice);

// Do calculation on device

        setup_kernel <<< grid, block >>> ( devStates, seed );
        montecarlo <<< grid, block >>> ( devStates, win_a_d, win_b_d );

        cudaDeviceSynchronize();

// Retrieve result from device and store it in host array

        cudaMemcpy(win_a, win_a_d, size, cudaMemcpyDeviceToHost);
        cudaMemcpy(win_b, win_b_d, size, cudaMemcpyDeviceToHost);

        printf("\nNumber of trials was = %d \n", trial_num);

        int sum_a = sum(win_a, trial_num);
        int sum_b = sum(win_b, trial_num);

        float pa = (float)sum_a / (float)trial_num;
        float pb = (float)sum_b / (float)trial_num;

        Pa_array[i] = pa;
        Pb_array[i] = pb;

        printf("Winnings with choice a %d times \n", sum_a);
        printf("Pa = %f \n", pa);
        printf("Winnings with choice b %d times \n", sum_b);
        printf("Pb = %f \n", pb);

        for (int s = 0; s < trial_num; ++s)
        {
          win_a[s] = 0;
          win_b[s] = 0;
        }
```

```
    }


    // Output results
    sprintf(str, "pa_%d.txt", group_n[l]);
    fp = fopen(str, "wt");
    for (k = 0; k < group_n[l]; k++) {
      fprintf(fp, " %f \n", Pa_array[k]);
    }
    fclose(fp);

    sprintf(str, "pb_%d.txt", group_n[l]);
    fp = fopen(str, "wt");
    for (k = 0; k < group_n[l]; k++) {
      fprintf(fp, " %f \n", Pb_array[k]);
    }
    fclose(fp);

    //   // Cleanup
    cudaFree(&win_a_d);
    cudaFree(&win_b_d);
    cudaFree(devStates);

    // CPU free space
    free(win_a);
    free(win_b);
    free(Pa_array);
    free(Pb_array);
  }
}
```