Project Report

CSE 336 Distributed Computing

Name: Mohamed Sameh Abdelhakim ID: 16p3061

Bonus Project Experiment Title: Calculating PI using openCL

Date: 19 / 5 /2019

Runtime

I ran the code on different global work-item sizes and the following results were obtained

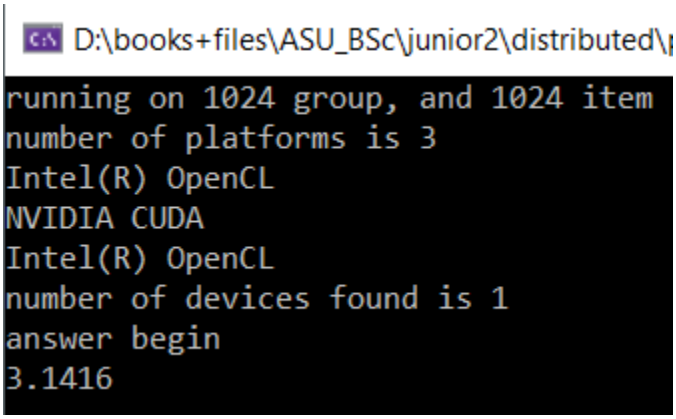
- Running using 128 work group with 128 work item per group:

```
D:\books+files\ASU_BSc\junior2\distributed\project\CalcPI\x64\Debug\CalcPI.exe
running on 128 group, and 128 item
number of platforms is 3
Intel(R) OpenCL
NVIDIA CUDA
Intel(R) OpenCL
number of devices found is 1
answer begin
3.14184
```

- Running using 512 work group with 512 work item per group:

```
running on 512 group, and 512 item
number of platforms is 3
Intel(R) OpenCL
NVIDIA CUDA
Intel(R) OpenCL
number of devices found is 1
answer begin
3.14161
```

- Running using 1024 work group with 1024 work item per group:



The result show that as number of intervals increase the accuracy of PI improves which is expected.

Source Codes

The project uses code listings from the provided book for host and kernel codes. I changed the listings to fit this experiment.

Host code

```
#define CL_USE_DEPRECATED_OPENCL_1_2_APIS
#include<fstream>
#include<iostream>
#include<CL/opencl.h>
using namespace std;
cl_int status;
cl_int ciErr;
cl_device_id* devices = 0;
cl_platform_id* platforms = 0;
cl_uint numDevices = 0;
cl_uint numPlatforms = 0;
char buffer[100000];
cl_uint buf_uint;
cl_ulong buf_ulong;
size_t bufsizet;

cl_float* srcC;

FILE* programHandle;
size_t programSize;
char* programBuffer;
cl_program cpProgram;
cl_kernel ckKernel;
//setting local and global sizes
size_t szLocalWorkSize = 256;
size_t szGlobalWorkSize = szLocalWorkSize * szLocalWorkSize;
//setting intervals to be number of all work items in all work groups
cl_long iNumIntervals = szGlobalWorkSize;

int main() {

	cout << "running on " << szLocalWorkSize << " group, and " << szLocalWorkSize << " item" <<
endl;
	// Allocate host array
	srcC = new cl_float[szGlobalWorkSize / szLocalWorkSize];


	status = clGetPlatformIDs(0, NULL, &numPlatforms);
	if (status != CL_SUCCESS) {
		cout << "Error failed to count platforms " << endl;
		return EXIT_FAILURE;
	}
	cout << "number of platforms is " << numPlatforms << endl;


	platforms = new cl_platform_id[numPlatforms];
	status = clGetPlatformIDs(numPlatforms, platforms, &numPlatforms);
	if (status != CL_SUCCESS) {
		cout << "Error failed to count platforms " << endl;
		return EXIT_FAILURE;
	}
```

```
for (int i = 0; i < numPlatforms; i++)
{
	char info[5000];
	clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, 5000, info, NULL);
	cout << info << endl;
}
auto GPU_platform = platforms[1];

//discover devices
status = clGetDeviceIDs(GPU_platform,
	CL_DEVICE_TYPE_ALL,
	0,
	NULL,
	&numDevices);
if (status != CL_SUCCESS) {
	cout << "Error failed to create device group " << endl;
	return EXIT_FAILURE;
}
cout << "number of devices found is " << numDevices << endl;

//Allocate space for each device
devices = new cl_device_id [numDevices];
//fill in devices
status = clGetDeviceIDs(GPU_platform,
	CL_DEVICE_TYPE_ALL,
	numDevices,
	devices,
	NULL);

if (status != CL_SUCCESS) {
	cout << "error failed to create a device group" << endl;
	return EXIT_FAILURE;
}


//Create a context associated to the devices
cl_context context = NULL;
context = clCreateContext(NULL,
	numDevices,
	devices,
	NULL,
	NULL,
	&status);
if (!context) {
	cout << "Error fails to create a context" << endl;
	return EXIT_FAILURE;
}

//Create command queue associated to the device we want
cl_command_queue cmdQueue;
cmdQueue = clCreateCommandQueue(context,
	devices[0], //GPU
	CL_QUEUE_PROFILING_ENABLE,
	&status);
if (!cmdQueue) {
	cout << "Error failed to create commands queue" << endl;
	return EXIT_FAILURE;
}

//Create program object for a context
std::ifstream ifs("calculatePI.cl");
std::string content((std::istreambuf_iterator<char>(ifs)),
```

```
	(std::istreambuf_iterator<char>()));
programSize = content.size();
programBuffer = new char[programSize + 1];
for (int i = 0; i < programSize; i++)
	programBuffer[i] = content[i];
programBuffer[programSize] = '\0';
cpProgram = clCreateProgramWithSource(context,
	1,
	(const char **) &programBuffer,
	&programSize,
	&ciErr);

if (!cpProgram) {
	cout << "Error failed to create compute program" << endl;
	return EXIT_FAILURE;
}
delete[] programBuffer;


//Build the pprogram
ciErr = clBuildProgram(cpProgram,
	0,
	NULL,
	NULL,
	NULL,
	NULL);

	if (ciErr != CL_SUCCESS) {
		size_t len;
		char buffer[2048];

		cout << "Error failed to build program executable" << endl;
		clGetProgramBuildInfo(cpProgram,
			devices[0],
			CL_PROGRAM_BUILD_LOG,
			sizeof(buffer),
			buffer,
			&len);
		cout << buffer << endl;
		exit(1);
	}

//create device buffers
cl_mem bufferC;
size_t datasize = sizeof(cl_float) * szGlobalWorkSize / szLocalWorkSize;
bufferC = clCreateBuffer(context,
	CL_MEM_WRITE_ONLY,
	datasize,
	NULL,
	&status);


//Create and compile the kernel
ckKernel = clCreateKernel(cpProgram,
	"CalculatePiShared",
	&ciErr);

if (!ckKernel || ciErr != CL_SUCCESS) {
	cout << "Error Failed to create compute kernel" << endl;
	exit(1);
}
```

```
//set the kernel arguments
ciErr = clSetKernelArg(ckKernel,
	0,
	sizeof cl_mem,
	(void*)& bufferC);
if (ciErr != CL_SUCCESS) {
	cout << "Erroe failed to set arguments " << ciErr << endl;
	return EXIT_FAILURE;
}

ciErr = clSetKernelArg(ckKernel,
	1,
	sizeof cl_long,
	(void*)& iNumIntervals);
if (ciErr != CL_SUCCESS) {
	cout << "Erroe failed to set arguments " << ciErr << endl;
	return EXIT_FAILURE;
}

ciErr = clEnqueueNDRangeKernel(cmdQueue,
	ckKernel,
	1,
	NULL,
	&szGlobalWorkSize,
	&szLocalWorkSize,
	0,
	NULL,
	NULL);

if (ciErr != CL_SUCCESS) {
	cout << "Erroe failed to launch kernel error code " << ciErr  << endl;
	return EXIT_FAILURE;
}
clFinish(cmdQueue);

ciErr = clEnqueueReadBuffer(cmdQueue,
	bufferC,
	CL_TRUE,
	0,
	datasize,
	srcC,
	0,
	NULL,
	NULL);

clFinish(cmdQueue);
cout << "answer begin" << endl;
float answer = 0.0;
for (int i = 0; i < szGlobalWorkSize / szLocalWorkSize; i++) {
	answer += srcC[i];
}
cout << answer << endl;

delete[] srcC;

if (ckKernel) clReleaseKernel(ckKernel);
if (cpProgram) clRetainProgram(cpProgram);
if (cmdQueue) clReleaseCommandQueue(cmdQueue);
if (context) clReleaseContext(context);

if (bufferC) clReleaseMemObject(bufferC);

cin.get();
```

```
        cin.get();
        return 0;
}
```

Kernel Code

```
__kernel void CalculatePiShared(__global float * c, ulong iNumIntervals) {
    __local float LocalPiValues[1024]; // work - group size = 256

    // work - item global index
    int glob_index = get_global_id(0);
    // work - item local index
    int local_index = get_local_id(0);
    // work - group index
    int group_index = get_group_id(0);
    // how many work - items are in WG?
    int WGsize = get_local_size(0);

    float x = 0.0;
    float y = 0.0;
    float pi = 0.0;

    while (glob_index < iNumIntervals) {
        x = (float)(1.0f / (float) iNumIntervals) * ((float) glob_index - 0.5f);
        y = (float) sqrt(1.0f - x * x);
        pi += 4.0f * (float)(y / (float) iNumIntervals);
        glob_index += get_global_size(0);
    }

    //store the product
    LocalPiValues[local_index] = pi;
    // wait for all threads in WG:
    barrier(CLK_LOCAL_MEM_FENCE);

    // Summation reduction:
    int i = WGsize / 2;
    while (i != 0) {
        if (local_index < i) {
            LocalPiValues[local_index] += LocalPiValues[local_index + i];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        i = i / 2;
    }

    // store partial dot product into global memory:
    if (local_index == 0) {
        c[group_index] = LocalPiValues[0];
    }
}
```