CAAM 520: COMPUTATIONAL SCIENCE II
HOMEWORK 4.

WEI WU

1. Introduction

In this project, we converted our CUDA code into OpenCL. Note that since there are several issues with reduce4 (and potentially reduce3), I opted to use reduce2 for the reduction kernel. It is not the most efficient kernel, but it gets the job done at least. We expect pretty low bandwith from the this reduction kernel.

2. Jacobi and Reduction Kernels

It is fairly easy to convert our Jacobi and reduction kernels in CUDA to OpenCL. Below are the code snippets for both kernels. See jacobi.cl and reduce.cl for full details.

```
__kernel void jacobi(int N, __global float * u, __global float *f, __global float *unew){

const int i = get_local_id(0) + get_group_id(0)*get_local_size(0) + 1; // offset by 1
const int j = get_local_id(1) + get_group_id(1)*get_local_size(1) + 1;

if (i < N+1 && j < N+1){
const int Np = (N+2);
const int id = i + j*(N+2);
const float ru = -u[id-Np]-u[id+Np]-u[id-1]-u[id+1];
const float newu = .25 * (f[id] - ru);
unew[id] = newu;
}
}
}
```

To do reduction, I applied sequential addressing from our lecture notes. It is shown in the code snippet below.

```
__kernel void reduce2(int N, __global float *u, __global float *unew, __global float *res){

__local float s_x[BDIM];

const int tid = get_local_id(0);
const int i = get_group_id(0)*get_local_size(0) + tid;

// load smem
s_x[tid] = 0;
if (i < N){
```

```
const float unew1 = unew[i];
const float diff1 = unew1 - u[i];
s_x[tid] = diff1*diff1;

// update
u[i] = unew1;
}
barrier(CLK_LOCAL_MEM_FENCE);

for (unsigned int s = get_local_size(0)/2; s > 0; s /= 2){
if (tid < s){
s_x[tid] += s_x[tid+s]; // fewer bank conflicts
}
barrier(CLK_LOCAL_MEM_FENCE);
}

if (tid==0){
res[get_group_id(0)] = s_x[0];
}
}
```

3. Correctness

I compare the results of my code with the serial version in homework 1 and the GPU version in homework 4. For any given number of threads, my code finishes with a similar number of iterations and reached similar Max errors as in the serial/CUDA version. For example, for a quick comparision, when N = 100, tol = 1e-6, my OpenCL GPU implementation finishes within 4411 iterations and Max error at 6.43794e-06. My CUDA GPU implementation finishes within 4414 iterations and Max error at 6.491e-06. My CPU implementations both finish with 4395 iterations, and Max error at 6.13072e-06. Discrepancies in the number of iterations and max error might be a result of implementation details and the usage of float for GPU implementations rather than double for the serial one.

4. Computational Performance

I experimented with different N and different thread-block size, and I documented their runtime, computational throughput, and arithmetic intensity of each kernels.

Table 1. threads/block = 1024 , tol = 1e-6, reduce2

N	DMWT	DMRT	FC(float)	Time(s)	Bandwith	Throughput
100	352.69MB/s	46.016MB/s	30914	31.612ms	0.399GB/s	9.78e-4 GFLOPs/sec
150	423.72MB/s	31.064MB/s	68590	111.97ms	0.455GB/s	6.12e-4 GFLOPs/sec
200	503.20MB/s	24.067MB/s	121164	305.65ms	0.526GB/s	3.96e-4 GLOPs/sec

TABLE 2. threads/block = 1024 , tol = 1e-6, Jacobi

N	DMWT	DMRT	FC(float)	Time(s)	Bandwith	Throughput
100	14.082GB/s	193.68MB/s	50000	21.873ms	14.28GB/s	2.38e-3 GFLOPs/sec
150	14.806GB/s	475.12MB/s	112500	55.946ms	15.28GB/s	2.01e-3 GFLOPs/sec
200	19.500GB/s	119.75MB/s	200000	157.22ms	19.62GB/s	1.27e-3 GFLOPs/sec

TABLE 3. threads/block = 256 , tol = 1e-6, reduce2

N	DMWT	DMRT	FC(float)	Time(s)	Bandwith	Throughput
100	404.15MB/s	34.044MB/s	30573	46.088ms	0.44GB/s	6.63-4 GFLOPs/sec
150	532.64MB/s	22.320MB/s	67868	170.75ms	0.55GB/s	3.97-4 GFLOPs/sec
200	597.81MB/s	16.118MB/s	119873	489.41ms	0.61GB/s	2.45-4 GFLOPs/sec

TABLE 4. threads/block = 256 , tol = 1e-6, Jacobi

N	DMWT	DMRT	FC(float)	Time(s)	Bandwith	Throughput
100	15.983GB/s	309.13MB/s	50000	21.601ms	16.29GB/s	2.31e-3 GFLOPs/sec
150	14.518GB/s	424.15MB/s	112500	64.363ms	14.94GB/s	1.75e-3 GFLOPs/sec
200	19.236GB/s	114.51MB/s	200000	160.83ms	19.35GB/s	1.24e-3 GFLOPs/sec