CAAM 520: COMPUTATIONAL SCIENCE II
HOMEWORK 5.

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, 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 implementation 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 instead of double for the serial one.

4. Computational Performance

I experimented on NOTS using Tesla K80 with different N and different thread-block size, and I documented their runtimes. For both GPU implementations, I documented the runtime for each kernel. For CPU code, I documented the total runtime. See log file for full details. Below is a sample where BDIM and pNthreads are set to 32.

When N = 100, tol = 1e-6: For OpenCL, jacobi takes 55.692 ms in total, and reduce2 takes 61.618 ms in total. The code finishes in 4411 iterations. Hence on average, jacobi takes 12.626 s, and reduces2 takes 13.969 s. For CUDA, jacobi takes on average 4.985 s, and reduces2 takes on average 7.24s. The CPU code takes 0.66 s in total, and 149.626 s per iteration.

When N = 300, tol = 1e-6: For OpenCL, jacobi takes 769.590 ms, and reduce2 takes 1539.974 ms. The code finishes in 35570 iterations. On average, jacobi takes 21.636 s, and redcue2 takes 43.294 s. For CUDA, jacobi takes on average 38.011 s, and reduce2 takes 16.933 s. The CPU code takes 55.14 s in total, and 1550.183 s per iteration.

5. Performance Comparison

Obviously the GPU implementations are much faster than the CPU version for a large problem size. The CUDA implementation is also faster than the OpenCL implementation, observed across several problem sizes and threads-per-block.