CAAM520 Computational Science

Homework 5

AO CAI
S01255664
Earth Science Department
Rice University
April 24, 2019

Contents

---

1 Introduction

The Goal is to solve the matrix system $Au = b$ resulting from an $(N+2)\times(N+2)$ 2D finite difference method for Laplace's equation $-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x,y)$ on $[-1,1]^2$. At each point (x_i, y_i), derivatives are approximated by:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}$$

where $h = 2/(N+1)$. Assuming zero boundary conditions $u(x,y) = 0$ reduce this to an $N \times N$ system for the interior nodes.

2 OpenCL parallelism for Jacobi method

To implement the parallel computing for the Poisson's equaion, I write the OpenCL code with the main funciton to allocate arrays and move data onto the devices. In the code, one kernel is used to perform the Jacobi iteraion and a second kernel performs reduction to compute the error. The code snippets are provided in the following:

```
/*************************************************************************
        > File Name: HW5_opencl.cpp
        > Author: Ao Cai
        > Mail: aocai166@gmail.com
        > Created Time: Sun 21 Apr 2019 11:56:02 AM CDT
 ************************************************************************/

#include<iostream>
using namespace std;
// for file IO
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <time.h>

#ifdef __APPLE__
#include <OpenCL/OpenCl.h>
#else
#include <CL/cl.h>
#endif

#define PI 3.14159265359f
```

```
#define MAX(a,b) (((a)>(b))?(a):(b))

void pfn_notify(const char *errinfo, const void *private_info, size_t cb, voi
  fprintf(stderr, "OpenCL Error (via pfn_notify): %s\n", errinfo);
}

void oclInit(int plat, int dev,
             cl_context &context,
             cl_device_id &device,
             cl_command_queue &queue){

  /* set up CL */
  cl_int            err;
  cl_platform_id    platforms[100];
  cl_uint           platforms_n;
  cl_device_id      devices[100];
  cl_uint           devices_n ;

  /* get list of platform IDs (platform == implementation of OpenCL) */
  clGetPlatformIDs(100, platforms, &platforms_n);

  if( plat > platforms_n) {
    printf("ERROR: platform %d unavailable \n", plat);
    exit(-1);
  }

  // find all available device IDs on chosen platform (could restrict to CPU
  cl_uint dtype = CL_DEVICE_TYPE_ALL;
  clGetDeviceIDs( platforms[plat], dtype, 100, devices, &devices_n);

  printf("devices_n = %d\n", devices_n);
  printf("dev = %d\n", dev);

  if(dev>=devices_n){
    printf("invalid device number for this platform\n");
    exit(0);
  }

  // choose user specified device
  device = devices[dev];

  // make compute context on device, pass in function pointer for error messa
```

```
  context = clCreateContext((cl_context_properties *)NULL, 1, &device, &pfn_n

  // create command queue
  queue   = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE,
}

void oclBuildKernel(const char *sourceFileName,
                    const char *functionName,
                    cl_context &context,
                    cl_device_id &device,
                    cl_kernel &kernel,
                    const char *flags
                    ){

  cl_int            err;

  // read in text from source file
  FILE *fh = fopen(sourceFileName, "r"); // file handle
  if (fh == 0){
    printf("Failed to open: %s\n", sourceFileName);
    throw 1;
  }

  // C function, get stats for source file (just need total size = statbuf.st
  struct stat statbuf;
  stat(sourceFileName, &statbuf);

  // read text from source file and add terminator
  char *source = (char *) malloc(statbuf.st_size + 1); // +1 for "\0" at end
  fread(source, statbuf.st_size, 1, fh); // read in 1 string element of size
  source[statbuf.st_size] = '\0'; // terminates the string

  // create program from source
  cl_program program = clCreateProgramWithSource(context,
                                                  1, // compile 1 kernel
                                                  (const char **) & source,
                                                  (size_t*) NULL, // lengths =
                                                  &err);

  if (!program){
    printf("Error: Failed to create compute program!\n");
    throw 1;
```

```
  }

  // compile and build program
  err = clBuildProgram(program, 1, &device, flags, (void (*)(cl_program, void

  // check for compilation errors
  char *build_log;
  size_t ret_val_size;
  err = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL,

  build_log = (char*) malloc(ret_val_size+1);
  err = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, ret_val_

  // to be careful, terminate the build log string with \0
  // there's no information in the reference whether the string is 0 terminat
  build_log[ret_val_size] = '\0';

  // print out compilation log
  fprintf(stderr, "%s", build_log );

  // create runnable kernel
  kernel = clCreateKernel(program, functionName, &err);
  if (! kernel || err != CL_SUCCESS){
    printf("Error: Failed to create compute kernel!\n");
    throw 1;
  }
}



int main(int argc, char **argv){

        int N = atoi(argv[1]); // matrix size
        float tol = atof(argv[2]); // tolerance

  cl_int            err;

  int plat = 1;
  int dev  = 0;

  cl_context context;
```

```
  cl_device_id device;
  cl_command_queue queue;

  cl_kernel kernel1;
  cl_kernel kernel2;

  oclInit(plat, dev, context, device, queue);

  const char *sourceFileName1 = "Jacobi.cl";
  const char *functionName1 = "Jacobi";

  const char *sourceFileName2 = "reduce.cl";
  const char *functionName2 = "reduce";

  int BDIM = 512;
  char flags[BUFSIZ];
  sprintf(flags, "-DBDIM=%d", BDIM);

  oclBuildKernel(sourceFileName1, functionName1,
                 context, device,
                 kernel1, flags);

  oclBuildKernel(sourceFileName2, functionName2,
                 context, device,
                 kernel2, flags);

  // START OF PROBLEM IMPLEMENTATION
//      int N = 16;
//      float tol = 1e-6;

  /* create host array */
  int Nr = (N+2)*(N+2);
  size_t sz = (N+2)*(N+2)*sizeof(float);

  float *u = (float*) malloc(sz);
  float *unew = (float*) malloc(sz);
  float *b = (float*) malloc(sz);
  float *res = (float*) malloc(sz);
  float h = 2.0/(N+1);
  clock_t begin=0, stop;
  if(plat==0){
          begin = clock();
```

```
}

for (int ix = 0; ix < N+2; ix++){
        for (int iz = 0; iz < N+2; iz++){
                const float tmpx = -1.0 + ix*h;
                const float tmpz = -1.0 + iz*h;
                b[ix + iz*(N+2)] = sin(PI*tmpx)*sin(PI*tmpz)*h*h;
                u[ix + iz*(N+2)] = 0.f;
                unew[ix + iz*(N+2)] = 0.f;
                res[ix + iz*(N+2)] = 0.f;
        }
}

// create device buffer and copy from host buffer
cl_mem d_u = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_P
cl_mem d_unew = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOS
cl_mem d_b = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_P
cl_mem d_res = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST

// now set kernel arguments
clSetKernelArg(kernel1, 0, sizeof(int), &N);
clSetKernelArg(kernel1, 1, sizeof(cl_mem), &d_u);
clSetKernelArg(kernel1, 2, sizeof(cl_mem), &d_b);
clSetKernelArg(kernel1, 3, sizeof(cl_mem), &d_unew);

clSetKernelArg(kernel2, 0, sizeof(int), &Nr);
clSetKernelArg(kernel2, 1, sizeof(cl_mem), &d_u);
clSetKernelArg(kernel2, 2, sizeof(cl_mem), &d_unew);
clSetKernelArg(kernel2, 3, sizeof(cl_mem), &d_res);

// set thread array
int dim = 1;
int Nt = N+2;
int Ng = N+2;
size_t local_dims[3] = {Nt,1,1};
size_t global_dims[3] = {Ng*Nt,1,1};

// cl event for timing
cl_event event;
cl_ulong start,end;
double nanoSeconds1 = 0.0;
double nanoSeconds2 = 0.0;
```

```
int iter = 0;
float obj = 1.0;
while(obj > tol*tol){

      // queue up kernel
      clEnqueueNDRangeKernel(queue, kernel1, dim, 0,
                             global_dims, local_dims,
                             0, (cl_event*)NULL, // wait list events
                             &event); // queue event along with kernel

      err = clWaitForEvents(1, &event);
      clFinish(queue);

      clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_
      clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulo
      nanoSeconds1 += end - start;

  clEnqueueNDRangeKernel(queue, kernel2, dim, 0,
                             global_dims, local_dims,
                             0, (cl_event*)NULL, // wait list events
                             &event); // queue event along with kernel

      err = clWaitForEvents(1, &event);
      clFinish(queue);

      clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_
      clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulo
      nanoSeconds2 += end - start;

      clEnqueueReadBuffer(queue, d_unew, CL_TRUE, 0, sz, unew, 0, 0, 0);
      clEnqueueReadBuffer(queue, d_u, CL_TRUE, 0, sz, u, 0, 0, 0);
      clEnqueueReadBuffer(queue, d_res, CL_TRUE, 0, sz, res, 0, 0, 0);

      obj = 0.f;
      for(int j = 0; j < N+2; ++j){
              obj += res[j];
      }

      /*if(!(iter%10000)){
              printf("Iter: %d, error = %lg\n", iter, sqrt(obj));
      }*/
```

```
        clEnqueueWriteBuffer(queue, d_u, CL_TRUE, 0, sz, unew, 0, 0, 0);
        iter++;
  }

  printf("kernel Jacobi execution time is: %0.3f milliseconds \n",nanoSeconds
  printf("kernel reduce execution time is: %0.3f milliseconds \n",nanoSeconds
  printf("Final Iteration: %d, obj= %lg\n", iter, sqrt(obj));
  if(plat==0){
          stop=clock();
          printf("Total computing time CPU is: %12.11f(s)\n",((float)(stop-be
  }
  // blocking read to host

  float maxerr=0.0;
        for(int ii=0; ii<N*N; ii++){
                maxerr = MAX(maxerr,fabs(u[ii]-b[ii]/(h*h*2.0*M_PI*M_PI)));
        }
        printf("Max error: %lg\n", maxerr);

        // free memory on host
        free(u);
        free(unew);
        free(b);
        free(res);
  /* print out results */

}
```

There are two major kernels in the code, the first one to present is the Jacobi kernel, where the updated u is stored at d_{unew}. Then the data is transferred from device to host by using the command $clEnqueueReadBuffer$, the computing time of the kernel is recorded by using $clGetEventProfilingInfo$. The implementation of the Jacobi kernel is the simplest way with out using the shared memory:

```
__kernel void Jacobi(int N,
                __global float *__restrict__ d_u,
                __global float *__restrict__ d_b,
                __global float *__restrict__ d_unew){

        const int id = get_local_id(0) + get_local_size(0)*get_group_id(0);
        const int ix = get_group_id(0);
        const int iz = get_local_id(0);
```

```
        float newu = 0.0f;
        const int Np = N+2;

        if(ix > 0 && ix < N+1){
                if(iz > 0 && iz < N+1){
                float ru = -d_u[id-Np]-d_u[id+Np]-d_u[id-1]-d_u[id+1];
            newu = .25 * (d_b[id] - ru);
                }
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        if(ix > 0 && ix < N+1){
                if(iz > 0 && iz < N+1){
                        d_unew[id] = newu;
//                      printf("newu = %f\n",newu);
                }
        }
}
```

The next kernel is the reduction kernel to compute the sum of the error using the tree-based reduction model:

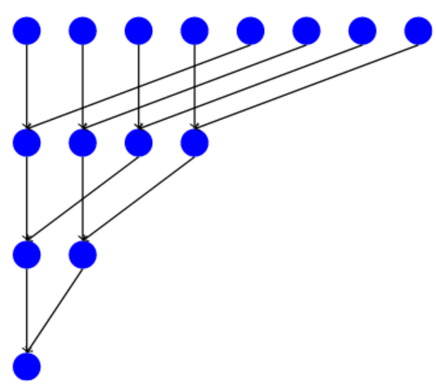


Figure 1: Tree-based GPU Reduction, cited from Yang et al., Geophysics, 2015

The reduction is achieved by reduce the number of threads at every for loop, and the code snippets are listed below.

```
__kernel void reduce(int N,
                __global const float *__restrict__ d_u,
                __global const float *__restrict__ d_unew,
```

```
        __global float *__restrict__ d_res){

    __local float s_res[BDIM+2];

    const int ii = get_local_id(0) + get_local_size(0)*get_group_id(0);
    const int tid = get_local_id(0);
    const int Nblock = get_local_size(0);
    const int bid = get_group_id(0);

    s_res[tid]=0.f;
    if(ii < N){
        const float diff = d_unew[ii] - d_u[ii];
        s_res[tid] = diff*diff;
    }

    barrier(CLK_LOCAL_MEM_FENCE);
    for(unsigned int s = Nblock/2; s > 0; s/=2){
        if(tid < Nblock){
            s_res[tid] += s_res[tid + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if(tid==0){
        d_res[bid] = s_res[0];
    }
}
```

Till now, I have presented all the implementation for the OpenCL code, the detailed documentation for the verify cation and timing of the OpenCL code is discussed in the following chapters.

3 Verification of the code

I will compared my OpenCL code with the Serial and CUDA reference code online, I made some small modification to the online code to record time and present the initial error at the first iteration. To start with, I will show the platform that I am using to write this report:

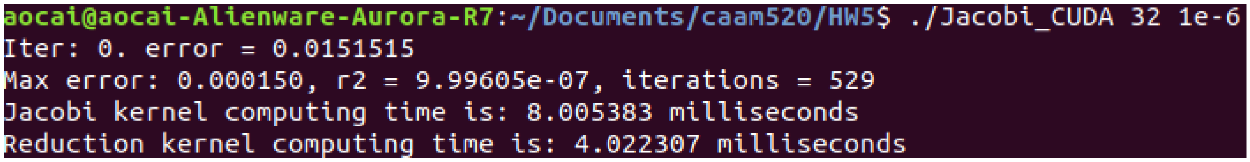
```
===================================
=== 2 OpenCL platform(s) found: ===
===================================

---------------------------------------------
PLATFORM: 0
PROFILE = FULL_PROFILE
VERSION = OpenCL 2.1 LINUX
NAME = Intel(R) CPU Runtime for OpenCL(TM) Applications
VENDOR = Intel(R) Corporation
EXTENSIONS = cl_khr_icd cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr
_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_byte_addressable_store cl_khr_d
epth_images cl_khr_3d_image_writes cl_intel_exec_by_local_thread cl_khr_spir cl_khr_fp64 cl_khr_imag
e2d_from_buffer cl_intel_vec_len_hint
*********************************************
PLATFORM: 0 DEVICE: 0
DEVICE_NAME = Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
DEVICE_VENDOR = Intel(R) Corporation
DEVICE_VERSION = OpenCL 2.1 (Build 0)
DRIVER_VERSION = 18.1.0.0920
DEVICE_MAX_COMPUTE_UNITS = 12
DEVICE_MAX_CLOCK_FREQUENCY = 3200
DEVICE_GLOBAL_MEM_SIZE = 16517259264
DEVICE_MAX_WORK_GROUP_SIZE = 8192
DEVICE_MAX_MEM_ALLOC_SIZE = 4129314816
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:     3
CL_DEVICE_MAX_WORK_ITEM_SIZES:  8192 / 8192 / 8192
---------------------------------------------

---------------------------------------------
PLATFORM: 1
PROFILE = FULL_PROFILE
VERSION = OpenCL 1.2 CUDA 10.0.292
NAME = NVIDIA CUDA
VENDOR = NVIDIA Corporation
EXTENSIONS = cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int3
2_base_atomics cl_khr_local_int32_extended_atomics cl_khr_fp64 cl_khr_byte_addressable_store cl_khr_
icd cl_khr_gl_sharing cl_nv_compiler_options cl_nv_device_attribute_query cl_nv_pragma_unroll cl_nv_
copy_opts cl_nv_create_buffer
*********************************************
PLATFORM: 1 DEVICE: 0
DEVICE_NAME = GeForce RTX 2080
DEVICE_VENDOR = NVIDIA Corporation
DEVICE_VERSION = OpenCL 1.2 CUDA
DRIVER_VERSION = 410.104
DEVICE_MAX_COMPUTE_UNITS = 46
DEVICE_MAX_CLOCK_FREQUENCY = 1710
DEVICE_GLOBAL_MEM_SIZE = 8335982592
DEVICE_MAX_WORK_GROUP_SIZE = 1024
DEVICE_MAX_MEM_ALLOC_SIZE = 2083995648
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:     3
CL_DEVICE_MAX_WORK_ITEM_SIZES:  1024 / 1024 / 64
---------------------------------------------
```

Figure 2: The platform used for CPU and GPU OpenCL implementation

For verification of the code, I am using a small problem size with N=32, the results from serial, CUDA and OpenCL code are listed below:

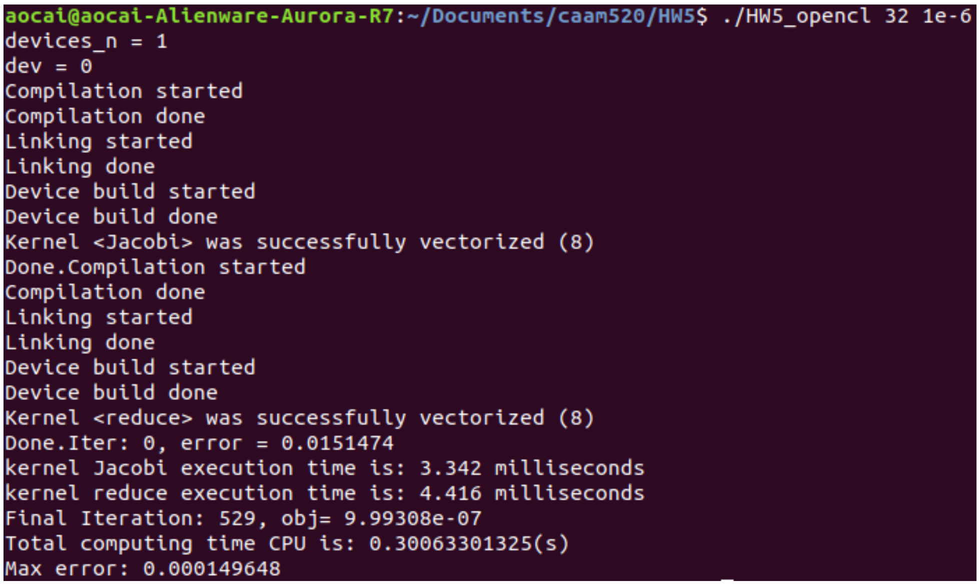
```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./Jacobi_Serial 32 1e-6
at iter 0: residual = 0.0151515
r2 = 9.96945e-07, iterations = 529
Max error: 0.000150
Serial code computing time is: 0.00868999958(s)
```

Figure 3: Serial Code Jacobi method with N=32



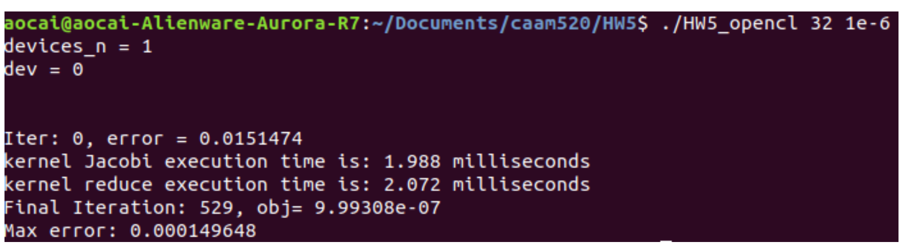
```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./Jacobi_CUDA 32 1e-6
Iter: 0. error = 0.0151515
Max error: 0.000150, r2 = 9.99605e-07, iterations = 529
Jacobi kernel computing time is: 8.005383 milliseconds
Reduction kernel computing time is: 4.022307 milliseconds
```

Figure 4: CUDA Code Jacobi method with N=32



```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./HW5_opencl 32 1e-6
devices_n = 1
dev = 0
Compilation started
Compilation done
Linking started
Linking done
Device build started
Device build done
Kernel <Jacobi> was successfully vectorized (8)
Done.Compilation started
Compilation done
Linking started
Linking done
Device build started
Device build done
Kernel <reduce> was successfully vectorized (8)
Done.Iter: 0, error = 0.0151474
kernel Jacobi execution time is: 3.342 milliseconds
kernel reduce execution time is: 4.416 milliseconds
Final Iteration: 529, obj= 9.99308e-07
Total computing time CPU is: 0.30063301325(s)
Max error: 0.000149648
```

Figure 5: OpenCL Code Jacobi method on CPU with N=32



```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./HW5_opencl 32 1e-6
devices_n = 1
dev = 0


Iter: 0, error = 0.0151474
kernel Jacobi execution time is: 1.988 milliseconds
kernel reduce execution time is: 2.072 milliseconds
Final Iteration: 529, obj= 9.99308e-07
Max error: 0.000149648
```

Figure 6: OpenCL Code Jacobi method on GPU with N=32

From the reported errors, all the implementations start with the initial error around 0.01515, there is a small difference between the initial error of OpenCL code and that of the Serial code. This might due to the small error during the data transformation. What's impressive, is that all the implementaions have exactly same numer of iterations to reach the tolerance of the misfit. The very close intial errors and exactly same number of iterations have demonstrated that the implementation of my OpenCL code is correct. The implementations on both CPU and GPU are efficient and accurate.

4 Jacobi method: Numerical results (Serial, CUDA, OpenCL)

For the OpenCL runtimes, in the N=32 case, the Jacobi and reduction execution in GPU is faster than that in CPU applications, as the following:

Kernel	CPU	GPU
Jacobi	3.342 ms	1.988 ms
Reduction	4.416 ms	2.072 ms

Table 1: Computing time (ms) using GPU and CPU OpenCl implementation with N=32

The faster kernel execution in GPU is under our expectation. In the following, I will compare the runtime between Serial code and OpenCL CPU implementation, and then compare the runtime between CUDA and OpenCL GPU implementation.

(1) Comparison between Serial code and OpenCL CPU implementation.
I am comparing the total runtime for the CPU code and OpenCL CPU implementation with N=64, 128, 256. The computing time table is listed below:

N	CPU	OpenCL CPU
64	0.0864 s	0.0510 s
128	1.275 s	0.439 s
256	18.9 s	5.457 s

Table 2: Computing time using CPU code and OpenCl CPU implementation with different N values

For $N = 64$, $N = 128$ and $N = 256$, I provide the support figures as following. Note that I am using the $< time.h >$ header to compute the total CPU time in OpenCL, which is actually the sum of computing time in different threads. Therefore, the computing time for OpenCL CPU code is calculated by the sum of computing time of Jacobi and Reduction kernel.

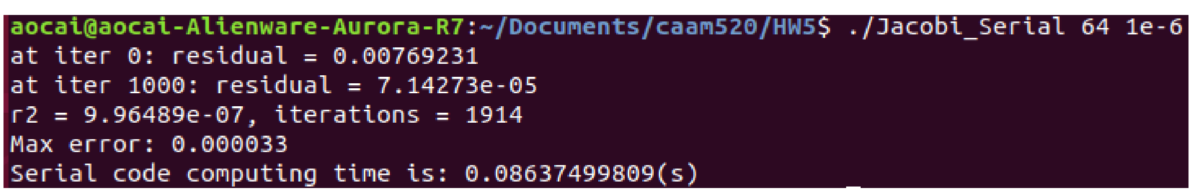

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./Jacobi_Serial 64 1e-6
at iter 0: residual = 0.00769231
at iter 1000: residual = 7.14273e-05
r2 = 9.96489e-07, iterations = 1914
Max error: 0.000033
Serial code computing time is: 0.08637499809(s)
```

Figure 7: Serial Code Jacobi method on CPU with N=64

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./Jacobi_Serial 128 1e-6
at iter 0: residual = 0.00387597
at iter 3000: residual = 0.000110234
at iter 6000: residual = 3.1351e-06
r2 = 9.99917e-07, iterations = 6964
Max error: 0.000003
Serial code computing time is: 1.27533400059(s)
```

Figure 8: Serial Code Jacobi method on CPU with N=128

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./Jacobi_Serial 256 1e-6
at iter 0: residual = 0.00194553
at iter 10000: residual = 9.79463e-05
at iter 20000: residual = 4.93104e-06
r2 = 9.99794e-07, iterations = 25340
Max error: 0.000024
Serial code computing time is: 18.92094802856(s)
```

Figure 9: Serial Code Jacobi method on CPU with N=256

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./HW5_opencl 64 1e-6
devices_n = 1
dev = 0
Compilation started
Compilation done
Linking started
Linking done
Device build started
Device build done
Kernel <Jacobi> was successfully vectorized (8)
Done.Compilation started
Compilation done
Linking started
Linking done
Device build started
Device build done
Kernel <reduce> was successfully vectorized (8)
Done.Iter: 0, error = 0.00769203
Iter: 1000, error = 7.14241e-05
kernel Jacobi execution time is: 17.876 milliseconds
kernel reduce execution time is: 33.101 milliseconds
Final Iteration: 1916, obj= 9.9663e-07
Total computing time CPU is: 1.42078900337(s)
Max error: 3.29211e-05
```

Figure 10: OpenCL Code Jacobi method on CPU with N=64

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./HW5_opencl 128 1e-6
devices_n = 1
dev = 0
Compilation started
Compilation done
Linking started
Linking done
Device build started
Device build done
Kernel <Jacobi> was successfully vectorized (8)
Done.Compilation started
Compilation done
Linking started
Linking done
Device build started
Device build done
Kernel <reduce> was successfully vectorized (8)
Done.Iter: 0, error = 0.00387595
Iter: 3000, error = 0.000110236
Iter: 6000, error = 3.14732e-06
kernel Jacobi execution time is: 121.694 milliseconds
kernel reduce execution time is: 317.355 milliseconds
Final Iteration: 7013, obj= 9.9711e-07
Total computing time CPU is: 9.18073844910(s)
Max error: 2.3461e-06
```

Figure 11: OpenCL Code Jacobi method on CPU with N=128

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./HW5_opencl 256 1e-6
devices_n = 1
dev = 0
Compilation started
Compilation done
Linking started
Linking done
Device build started
Device build done
Kernel <Jacobi> was successfully vectorized (8)
Done.Compilation started
Compilation done
Linking started
Linking done
Device build started
Device build done
Kernel <reduce> was successfully vectorized (8)
Done.Iter: 0, error = 0.00194552
Iter: 10000, error = 9.79489e-05
Iter: 20000, error = 4.96686e-06
kernel Jacobi execution time is: 1131.354 milliseconds
kernel reduce execution time is: 4325.170 milliseconds
Final Iteration: 25929, obj= 9.98343e-07
Total computing time CPU is: 94.40406036377(s)
Max error: 1.93726e-05
```

Figure 12: OpenCL Code Jacobi method on CPU with N=256

Discussion:

From the computing time table, we can find that the execution time using OpenCL CPU implementation is faster than the conventional serial code. This is because by assign workloads into different work items (threads), the OpenCL implementation is acutally functioning as a parallel programm. When the size of problem (N) is relatively small, the differce bettween CPU code and OpenCL is not obvious. The discrepancies in computing time become bigger when we are working on larger scales of the problem, but the number of iteration it takes for both applicaitons are close. The slightly larger number of iteration for

OpenCL to converge might due to the serial code is in double percision while the OpenCL is in single percision.

(2) Comparison between CUDA code and OpenCL GPU implementation.

Similar to the comparison in CPU, I am comparing the kernel runtimes for the CUDA code and OpenCL CPU implementation with N=64, 128, 256. The computing time table is listed below: I am comparing the total runtime for the CPU code and OpenCL CPU implementation with N=64, 128, 256. The computing time table is listed below:

N	CUDA	OpenCL
32 *Jacobi*	8.005 ms	1.988 ms
32 *Reduce*	4.022 ms	2.072 ms
64 *Jacobi*	22.095 ms	7.255 ms
64 *Reduce*	7.512 ms	9.299 ms
128 *Jacobi*	85.602 ms	27.324 ms
128 *Reduce*	33.716 ms	28.625 ms
256 *Jacobi*	340.782 ms	130.455 ms
256 *Reduce*	145.089 ms	162.481 ms

Table 3: Table for kernel computing time (ms) using Jacobi method with CUDA/OpenCL application and various N

For $N = 64$, $N = 128$ and $N = 256$, I provide the support figures as following. Note that I am using the $< time.h >$ header to compute the total CPU time in OpenCL, which is actually the sum of computing time in different threads. Therefore, the computing time for OpenCL CPU code is calculated by the sum of computing time of Jacobi and Reduction kernel.

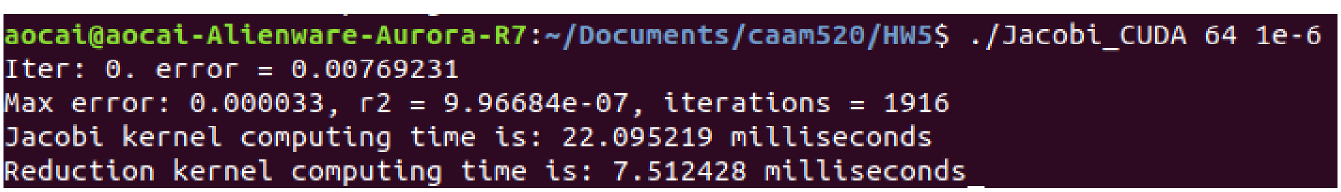


Figure 13: CUDA Code Jacobi method on GPU with N=64

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./Jacobi_CUDA 128 1e-6
Iter: 0. error = 0.00387597
Max error: 0.000002, r2 = 9.97147e-07, iterations = 7013
Jacobi kernel computing time is: 85.602097 milliseconds
Reduction kernel computing time is: 33.716309 milliseconds
```

Figure 14: CUDA Code Jacobi method on GPU with N=128

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./Jacobi_CUDA 256 1e-6
Iter: 0. error = 0.00194553
Max error: 0.000019, r2 = 9.984e-07, iterations = 25929
Jacobi kernel computing time is: 340.782440 milliseconds
Reduction kernel computing time is: 145.088638 milliseconds
```

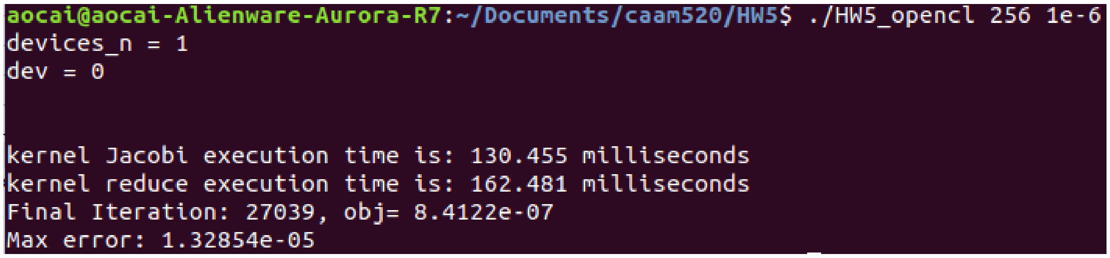
Figure 15: CUDA Code Jacobi method on GPU with N=256

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./HW5_opencl 64 1e-6
devices_n = 1
dev = 0


kernel Jacobi execution time is: 7.255 milliseconds
kernel reduce execution time is: 9.299 milliseconds
Final Iteration: 1916, obj= 9.9663e-07
Max error: 3.29211e-05
```

Figure 16: OpenCL Code Jacobi method on GPU with N=64

```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./HW5_opencl 128 1e-6
devices_n = 1
dev = 0


kernel Jacobi execution time is: 27.324 milliseconds
kernel reduce execution time is: 28.625 milliseconds
Final Iteration: 7013, obj= 9.9711e-07
Max error: 2.3461e-06
```

Figure 17: OpenCL Code Jacobi method on GPU with N=128



```
aocai@aocai-Alienware-Aurora-R7:~/Documents/caam520/HW5$ ./HW5_opencl 256 1e-6
devices_n = 1
dev = 0


kernel Jacobi execution time is: 130.455 milliseconds
kernel reduce execution time is: 162.481 milliseconds
Final Iteration: 27039, obj= 8.4122e-07
Max error: 1.32854e-05
```

Figure 18: OpenCL Code Jacobi method on GPU with N=256

Discussion:

From the computing time table, we can find that the kernel execution time using OpenCL GPU implementation is faster than the corresponding implementation using the CUDA implementation, in most of the cases. The result is a little bit out of my expectation and here is my explanation: On the one side, the $cudaEventt$ system that I used to record kernel execution time might be relatively slower than the $clGetEventProfilingInfo$. Since the time is calculated by sum at every iteration step, the timing calculation might have influence on the total computing time. On the other size, the number of threads I used in CUDA is relatively small comparing with the size of BDIM that I used in OpenCL. Due to the limitation of reduciton kernel, the size N will be prefered to be the multiples of 32. However, it is still interesting to find out the good performance of OpenCL in both CPU and GPU implementations.