Lyon Kee

keel@oregonstate.edu

Project #6: OpenCL

Key snippets of code

1. Program header
2. Allocate the host memory buffers
3. Create an OpenCL context
4. Create an OpenCL command queue
5. Allocate the device memory buffers
6. Write the data from the host buffers to the device buffers
7. Read the kernel code from a file
8. Compile and link the kernel code
9. Create the kernel object
10. Setup the arguments to the kernel object
11. Enqueue the kernel object for execution
12. Read the results buffer back from the device to the host
13. Clean everything up

1. Program header

```c
// 1. Program header

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <omp.h>

#include "cl.h"
#include "cl_platform.h"
```

We observe that cl.h and cl_platform.h is added which is needed for OpenCL, OpenMP for time measurement, and the rest are the standard libraries for operations.

2. Allocate the host memory buffers

```c
    // 2. xreate the host memory buffers:

    // read the data file:

    FILE* fdata;
#ifdef _WIN32
    err = fopen_s(&fdata, DATAFILE, "r");
    if (err != 0)
#else
    fdata = fopen(DATAFILE, "r");
    if (fdata == NULL)
#endif
    {
        fprintf( stderr, "Cannot open data file '%s'\n", DATAFILE );
        return -1;
    }

    float x, y;
    for( int i = 0; i < DATASIZE; i++ )
    {
#ifdef _WIN32
        fscanf_s(fdata, "%f %f", &x, &y);
#else
        fscanf( fdata, "%f %f", &x, &y );
#endif
        hX[i] = x;
        hY[i] = y;
    }
    fclose( fdata );
```

Before this, we check if kernel file is readable and also select the best OpenCL device. We read in data and store it into host device arrays of hX and hY.

3. Create an OpenCL context

```
// 3. create an opencl context:

Context = clCreateContext( NULL, 1, &Device, NULL, NULL, &status );
if( status != CL_SUCCESS )
    fprintf( stderr, "clCreateContext failed\n" );
```

Created an OpenCL context and set it to ` cl_context context` and check for CL_SUCCESS. Check for CL_SUCCESS before we proceed.

4. Create an OpenCL command queue

```
// 4. create an opencl command queue:

CmdQueue = clCreateCommandQueue( Context, Device, 0, &status );
if( status != CL_SUCCESS )
    fprintf( stderr, "clCreateCommandQueue failed\n" );
```

Create a cl_command_queue to store commands for OpenCL to quickly grab commands to execute. Check for CL_SUCCESS before we proceed.

5. Allocate the device memory buffers

```
// 5. allocate the device memory buffers:

size_t xySize = DATASIZE  * sizeof(float);

cl_mem dX     = clCreateBuffer( Context, CL_MEM_READ_ONLY,  xySize, NULL, &status );
cl_mem dY     = clCreateBuffer( Context, CL_MEM_READ_ONLY,  xySize, NULL, &status );
cl_mem dSumx2 = clCreateBuffer( Context, CL_MEM_READ_ONLY,  xySize, NULL, &status );
cl_mem dSumx  = clCreateBuffer( Context, CL_MEM_READ_ONLY,  xySize, NULL, &status );
cl_mem dSumxy = clCreateBuffer( Context, CL_MEM_READ_ONLY,  xySize, NULL, &status );
cl_mem dSumy  = clCreateBuffer( Context, CL_MEM_READ_ONLY,  xySize, NULL, &status );

if( status != CL_SUCCESS )
    fprintf( stderr, "clCreateBuffer failed\n" );
```

Used clCreateBuffer to allocate memory on the OpenCL device to allow the device to read and write to when it is doing computation. Check for CL_SUCCESS before we proceed.

6. Write the data from the host buffers to the device buffers

```
// 6. enqueue the 2 commands to write the data from the host buffers to the device buffers:

status = clEnqueueWriteBuffer( CmdQueue, dX, CL_FALSE, 0, DATASIZE, hX, 0, NULL, NULL );
if( status != CL_SUCCESS )
    fprintf( stderr, "clEnqueueWriteBuffer failed (1)\n" );

status = clEnqueueWriteBuffer( CmdQueue, dY, CL_FALSE, 0, DATASIZE, hY, 0, NULL, NULL );
if( status != CL_SUCCESS )
    fprintf( stderr, "clEnqueueWriteBuffer failed (2)\n" );

Wait( CmdQueue );
```

Writing data loaded from file -> host memory to device memory and waiting for it to finish writing.

7. Read the kernel code from a file

```
// 7. read the kernel code from a file ...

fseek( fp, 0, SEEK_END );
size_t fileSize = ftell( fp );
fseek( fp, 0, SEEK_SET );
char *clProgramText = new char[ fileSize+1 ];       // leave room for '\0'
size_t n = fread( clProgramText, 1, fileSize, fp );
clProgramText[fileSize] = '\0';
fclose( fp );
if( n != fileSize )
    fprintf( stderr, "Expected to read %d bytes read from '%s' -- actually read %d.\n", (int)fileSize, CL_FILE_NAME, (int)n );
```

Reading our OpenCL code to create a kernel program

8. Compile and link the kernel code

```
// 8. compile and link the kernel code:

char *options = { (char *)"" };
status = clBuildProgram( Program, 1, &Device, options, NULL, NULL );
if( status != CL_SUCCESS )
{
    size_t size;
    clGetProgramBuildInfo( Program, Device, CL_PROGRAM_BUILD_LOG, 0, NULL, &size );
    cl_char *log = new cl_char[ size ];
    clGetProgramBuildInfo( Program, Device, CL_PROGRAM_BUILD_LOG, size, log, NULL );
    fprintf( stderr, "clBuildProgram failed:\n%s\n", log );
    delete [ ] log;
}
```

Compile and link the kernel code that is loaded to a string in step 7

9. Create the kernel object

```
// 9. create the kernel object:

Kernel = clCreateKernel( Program, "Regression", &status );
if( status != CL_SUCCESS )
    fprintf( stderr, "clCreateKernel failed\n" );
```

Create a kernel from the compiled and linked program in step 8.

10. Setup the arguments to the kernel object

```
// 10. setup the arguments to the kernel object:

status = clSetKernelArg( Kernel, 0, sizeof(cl_mem), &dX );
status = clSetKernelArg( Kernel, 1, sizeof(cl_mem), &dY );

status = clSetKernelArg( Kernel, 2, sizeof(cl_mem), &dSumx2 );
status = clSetKernelArg( Kernel, 3, sizeof(cl_mem), &dSumx );
status = clSetKernelArg( Kernel, 4, sizeof(cl_mem), &dSumxy );
status = clSetKernelArg( Kernel, 5, sizeof(cl_mem), &dSumy );
if( status != CL_SUCCESS )
    fprintf( stderr, "clSetKernelArg failed: %d\n", status );
```

Set kernel arguments which should align with the kernel program arguments below:

```
kernel
void
Regression( IN global const float *dX,
        IN global const float *dY,
        OUT global float *dSumx2,
        OUT global float *dSumx,
        OUT global float *dSumxy,
        OUT global float *dSumy )
{
    int gid = get_global_id( 0 );

    float x = dX[gid];
    float y = dY[gid];
    dSumx2[ gid ]   = x * x;
    dSumx[ gid ]    = x;
    dSumxy[ gid ]   = x * y;
    dSumy[ gid ]    = y;
}
```

11. Enqueue the kernel object for execution

```
    // 11. enqueue the kernel object for execution:

    size_t globalWorkSize[3] = { DATASIZE,  1, 1 };
    size_t localWorkSize[3]  = { LOCALSIZE, 1, 1 };

    Wait( CmdQueue );

    double time0 = omp_get_wtime( );

    status = clEnqueueNDRangeKernel( CmdQueue, Kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL );
    if( status != CL_SUCCESS )
        fprintf( stderr, "clEnqueueNDRangeKernel failed: %d\n", status );

    Wait( CmdQueue );
    double time1 = omp_get_wtime( );
```

Enqueue the Kernel and time the execution after waiting for it to finish running.

12. Read the results buffer back from the device to the host

```
    // 12. read the results buffer back from the device to the host:

    status = clEnqueueReadBuffer( CmdQueue, dSumx2, CL_FALSE, 0, xySize, hSumx2, 0, NULL, NULL );
    status = clEnqueueReadBuffer( CmdQueue, dSumx, CL_FALSE, 0, xySize, hSumx , 0, NULL, NULL );
    status = clEnqueueReadBuffer( CmdQueue, dSumxy, CL_FALSE, 0, xySize, hSumxy , 0, NULL, NULL );
    status = clEnqueueReadBuffer( CmdQueue, dSumy, CL_FALSE, 0, xySize, hSumy , 0, NULL, NULL );

    Wait( CmdQueue );
```

Read the output of the Kernel back to host device to compute sum.

13. Clean everything up

```
    // 13. clean everything up:

    clReleaseKernel(        Kernel    );
    clReleaseProgram(       Program   );
    clReleaseCommandQueue(  CmdQueue  );
    clReleaseMemObject(     dSumx2  );
    clReleaseMemObject(     dSumx   );
    clReleaseMemObject(     dSumxy  );
    clReleaseMemObject(     dSumy  );
```

Clean up created objects relating to the OpenCL device.

Tables of data

Columns: Array Size

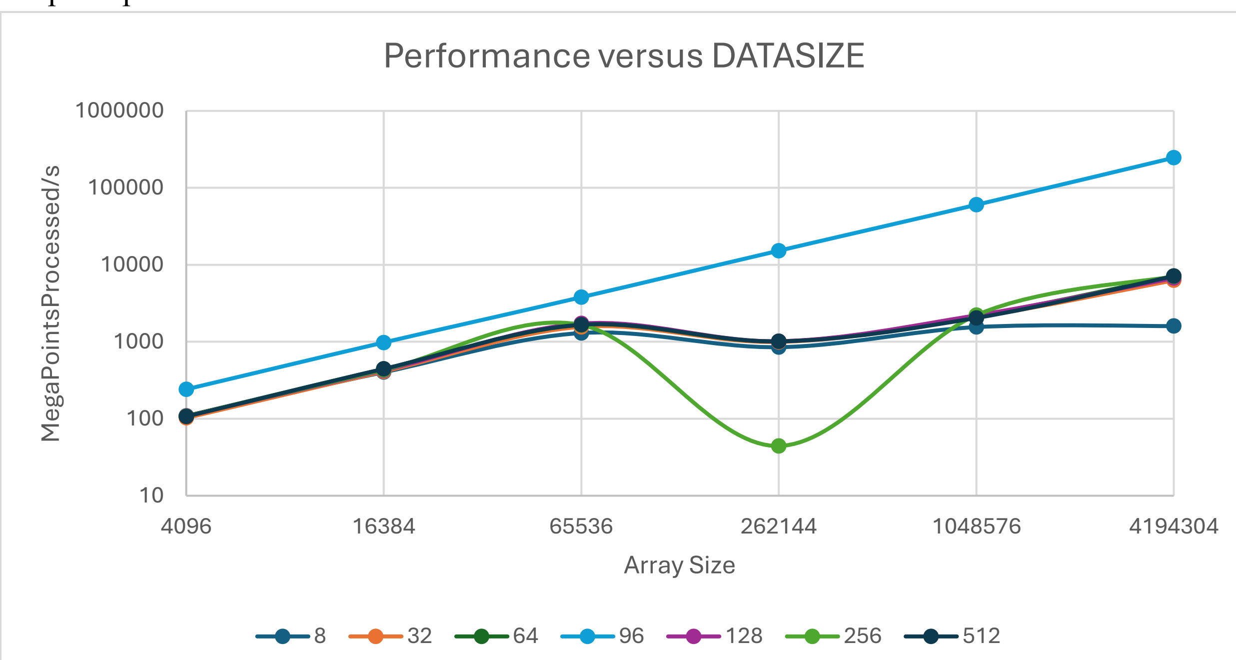
Rows: Work Elements

Values: megaPointsProcessedPerSecond

Sum of MegaPointsProcessedPerSecond	Array Size					
Work Elements	**4096**	**16384**	**65536**	**262144**	**1048576**	**4194304**
8	108.68	402.71	1301.2	851.84	1553.27	1599.62
32	102.55	411.84	1567.16	997.16	2084.37	6297.17
64	108.28	443.89	1657.76	1011.41	2162	7150.43
96	241.49	979.85	3793.46	15259.4	60221.43	246152.14
128	108.87	426.04	1725.4	1016.4	2207.97	6726.27
256	108.27	433.16	1656.98	44.35	2220.25	7087.84
512	107.01	444.34	1669.25	1014.2	2042.92	7129.99

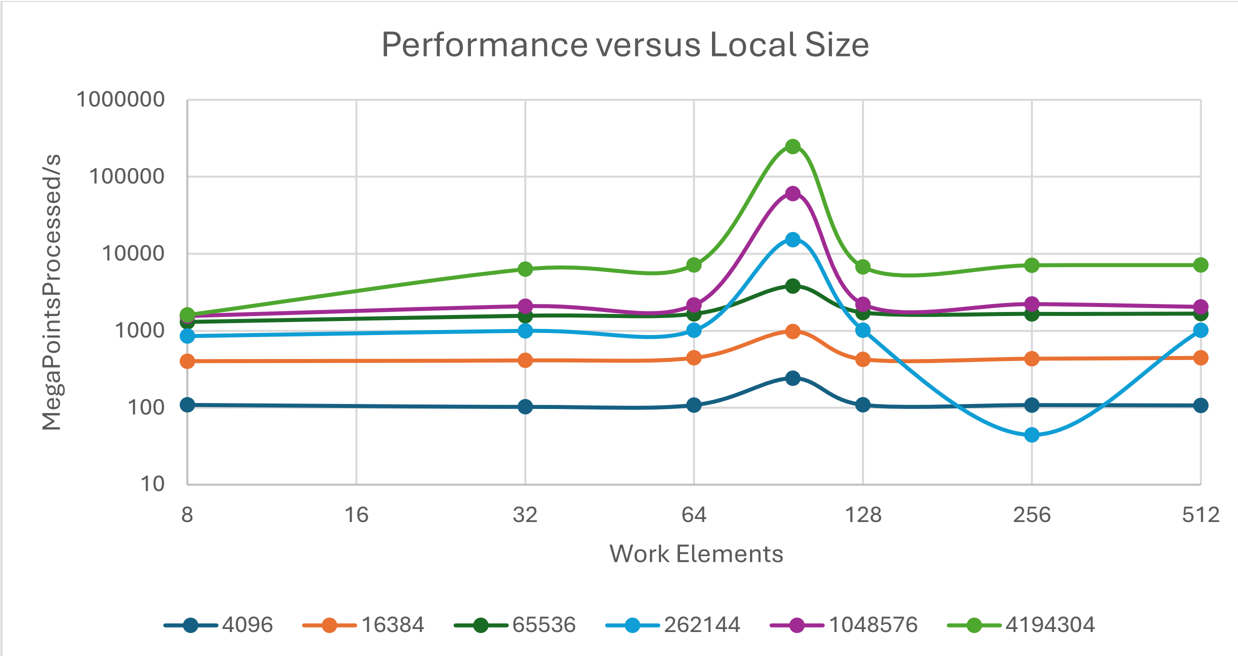
Graphs of data

Graph of performance versus DATASIZE with colored curves of constant LOCALSIZE



In the graph above, we observe that as the Array Size increases, the performance increases as well, although we do observe that given specific “Work Elements” or constant local sizes, the performance peaks and stays constant even when we increase DATASIZE, this is likely the peak performance of the device with data being abundant so there is little inefficiency.

Graph of performance versus LOCALSIZE with colored curves of constant DATASIZE



The graph above shows that there is a sweet spot in the number of work elements, and a number above or below it will yield a similar performance. We observe that the best performance is at 96 work elements, from lecture it seems that each "gripper" has 32 work elements, and at 64, we have 2 sets which allows the scheduler to swap out memory when it is performing tasks that require waiting and it just seems like in this system, 96 was the perfect number which yielded the best performance.

PDF Commentary

1. What machine you ran this on

I ran this on the one Tesla V100 of the A100s that the class has access to, therefore I get dedicated CPU&GPU time and it is very reliable.

2. Show the table and the graphs

Shown in the above sections.

3. What patterns are you seeing in the performance curves? What difference does the size of data make? What difference does the size of each work-group make?

Graph analysis and evaluations are shown in the above sections.