/\* first.cl

kernel void ArrayMult( global const float \*dA, global const float \*dB, global float \*dC ){

int gid = get\_global\_id( 0 );

dC[gid] = dA[gid] \* dB[gid];

}

\*/

/\*first.cpp\*/

// 1. Program header

#include <stdio.h>

#include <math.h>

#include <string.h>

#include <stdlib.h>

#ifdef WIN32

#include <windows.h>

#else

#include <unistd.h>

#endif

#include <omp.h>

#include "CL/cl.h"

#include "CL/cl\_platform.h"

#ifndef NUM\_ELEMENTS

#define NUM\_ELEMENTS 64\*1024\*1024

#endif

#ifndef LOCAL\_SIZE

#define LOCAL\_SIZE 32

#endif

#define NUM\_WORK\_GROUPS NUM\_ELEMENTS/LOCAL\_SIZE

const char \* CL\_FILE\_NAME = { "first.cl" };

const float TOL = 0.0001f;

void Wait( cl\_command\_queue );

int LookAtTheBits( float );

int

main( int argc, char \*argv[ ] )

{

// see if we can even open the opencl kernel program

// (no point going on if we can't):

FILE \*fp;

#ifdef WIN32

errno\_t err = fopen\_s( &fp, CL\_FILE\_NAME, "r" );

if( err != 0 )

#else

fp = fopen( CL\_FILE\_NAME, "r" );

if( fp == NULL )

#endif

{

fprintf( stderr, "Cannot open OpenCL source file '%s'\n", CL\_FILE\_NAME );

return 1;

}

cl\_int status; // returned status from opencl calls

// test against CL\_SUCCESS

// get the platform id:

cl\_platform\_id platform;

status = clGetPlatformIDs( 1, &platform, NULL );

if( status != CL\_SUCCESS )

fprintf( stderr, "clGetPlatformIDs failed (2)\n" );

// get the device id:

cl\_device\_id device;

status = clGetDeviceIDs( platform, CL\_DEVICE\_TYPE\_GPU, 1, &device, NULL );

if( status != CL\_SUCCESS )

fprintf( stderr, "clGetDeviceIDs failed (2)\n" );

// 2. allocate the host memory buffers:

float \*hA = new float[ NUM\_ELEMENTS ];

float \*hB = new float[ NUM\_ELEMENTS ];

float \*hC = new float[ NUM\_ELEMENTS ];

// fill the host memory buffers:

for( int i = 0; i < NUM\_ELEMENTS; i++ )

{

hA[i] = hB[i] = (float) sqrt( (double)i );

}

size\_t dataSize = NUM\_ELEMENTS \* sizeof(float);

// 3. create an opencl context:

cl\_context context = clCreateContext( NULL, 1, &device, NULL, NULL, &status );

if( status != CL\_SUCCESS )

fprintf( stderr, "clCreateContext failed\n" );

// 4. create an opencl command queue:

cl\_command\_queue cmdQueue = clCreateCommandQueue( context, device, 0, &status );

if( status != CL\_SUCCESS )

fprintf( stderr, "clCreateCommandQueue failed\n" );

// 5. allocate the device memory buffers:

cl\_mem dA = clCreateBuffer( context, CL\_MEM\_READ\_ONLY, dataSize, NULL, &status );

if( status != CL\_SUCCESS )

fprintf( stderr, "clCreateBuffer failed (1)\n" );

cl\_mem dB = clCreateBuffer( context, CL\_MEM\_READ\_ONLY, dataSize, NULL, &status );

if( status != CL\_SUCCESS )

fprintf( stderr, "clCreateBuffer failed (2)\n" );

cl\_mem dC = clCreateBuffer( context, CL\_MEM\_WRITE\_ONLY, dataSize, NULL, &status );

if( status != CL\_SUCCESS )

fprintf( stderr, "clCreateBuffer failed (3)\n" );

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

status = clEnqueueWriteBuffer( cmdQueue, dA, CL\_FALSE, 0, dataSize, hA, 0, NULL, NULL );

if( status != CL\_SUCCESS )

fprintf( stderr, "clEnqueueWriteBuffer failed (1)\n" );

status = clEnqueueWriteBuffer( cmdQueue, dB, CL\_FALSE, 0, dataSize, hB, 0, NULL, NULL );

if( status != CL\_SUCCESS )

fprintf( stderr, "clEnqueueWriteBuffer failed (2)\n" );

Wait( cmdQueue );

// 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", fileSize, CL\_FILE\_NAME, n );

// create the text for the kernel program:

char \*strings[1];

strings[0] = clProgramText;

cl\_program program = clCreateProgramWithSource( context, 1, (const char \*\*)strings, NULL, &status );

if( status != CL\_SUCCESS )

fprintf( stderr, "clCreateProgramWithSource failed\n" );

delete [ ] clProgramText;

// 8. compile and link the kernel code:

char \*options = { "" };

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;

}

// 9. create the kernel object:

cl\_kernel kernel = clCreateKernel( program, "ArrayMult", &status );

if( status != CL\_SUCCESS )

fprintf( stderr, "clCreateKernel failed\n" );

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

status = clSetKernelArg( kernel, 0, sizeof(cl\_mem), &dA );

if( status != CL\_SUCCESS )

fprintf( stderr, "clSetKernelArg failed (1)\n" );

status = clSetKernelArg( kernel, 1, sizeof(cl\_mem), &dB );

if( status != CL\_SUCCESS )

fprintf( stderr, "clSetKernelArg failed (2)\n" );

status = clSetKernelArg( kernel, 2, sizeof(cl\_mem), &dC );

if( status != CL\_SUCCESS )

fprintf( stderr, "clSetKernelArg failed (3)\n" );

// 11. enqueue the kernel object for execution:

size\_t globalWorkSize[3] = { NUM\_ELEMENTS, 1, 1 };

size\_t localWorkSize[3] = { LOCAL\_SIZE, 1, 1 };

Wait( cmdQueue );

double time0 = omp\_get\_wtime( );

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( );

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

status = clEnqueueReadBuffer( cmdQueue, dC, CL\_TRUE, 0, dataSize, hC, 0, NULL, NULL );

if( status != CL\_SUCCESS )

fprintf( stderr, "clEnqueueReadBuffer failed\n" );

Wait( cmdQueue );

// did it work?

for( int i = 0; i < NUM\_ELEMENTS; i++ )

{

float expected = hA[i] \* hB[i];

if( fabs( hC[i] - expected ) > TOL )

{

fprintf( stderr, "%4d: %13.6f \* %13.6f wrongly produced %13.6f instead of %13.6f (%13.8f)\n",

i, hA[i], hB[i], hC[i], expected, fabs(hC[i]-expected) );

fprintf( stderr, "%4d: 0x%08x \* 0x%08x wrongly produced 0x%08x instead of 0x%08x\n",

i, LookAtTheBits(hA[i]), LookAtTheBits(hB[i]), LookAtTheBits(hC[i]), LookAtTheBits(expected) );

}

}

fprintf( stderr, "%8d\t%4d\t%10d\t%10.3lf GigaMultsPerSecond\n",

NUM\_ELEMENTS, LOCAL\_SIZE, NUM\_WORK\_GROUPS, (double)NUM\_ELEMENTS/(time1-time0)/1000000000. );

#ifdef WIN32

Sleep( 2000 );

#endif

// 13. clean everything up:

clReleaseKernel( kernel );

clReleaseProgram( program );

clReleaseCommandQueue( cmdQueue );

clReleaseMemObject( dA );

clReleaseMemObject( dB );

clReleaseMemObject( dC );

delete [ ] hA;

delete [ ] hB;

delete [ ] hC;

return 0;

}

// wait until all queued tasks have completed:

void

Wait( cl\_command\_queue queue )

{

cl\_event wait;

cl\_int status = clEnqueueMarker( queue, &wait );

if( status != CL\_SUCCESS )

fprintf( stderr, "Wait: clEnqueueMarker failed\n" );

status = clEnqueueWaitForEvents( queue, 1, &wait );

if( status != CL\_SUCCESS )

fprintf( stderr, "Wait: clEnqueueWaitForEvents failed\n" );

}

int

LookAtTheBits( float fp )

{

int \*ip = (int \*)&fp;

return \*ip;

}