CS 475/575 -- Spring Quarter 2020

Project #6

OpenCL Array Multiply, Multiply-Add, and Multiply-Reduce

100 Points

Due: May 29

```
submit-c ~/cs575/pro6 998$ ls
CL                        MultAdd      Mult.err      MultScriptAdd.sh
data.txt                  MultAdd.cl   Mult.out      MultScriptRedu.sh
DGX_MultScriptAdd.bash    MultAdd.cpp  MultRedu      MultScript.sh
DGX_MultScript.bash       MultAdd.err  MultRedu.cl   part3
DGX_MultScriptRedu.bash   MultAdd.out  MultRedu.cpp
Makefile                  Mult.cl      MultRedu.err
Mult                      Mult.cpp     MultRedu.out
submit-c ~/cs575/pro6 999$ sbatch DGX_MultScript.bash
Submitted batch job 18921
submit-c ~/cs575/pro6 1000$ ls
CL                        MultAdd      Mult.err      MultScriptAdd.sh
data.txt                  MultAdd.cl   Mult.out      MultScriptRedu.sh
DGX_MultScriptAdd.bash    MultAdd.cpp  MultRedu      MultScript.sh
DGX_MultScript.bash       MultAdd.err  MultRedu.cl   part3
DGX_MultScriptRedu.bash   MultAdd.out  MultRedu.cpp
Makefile                  Mult.cl      MultRedu.err
Mult                      Mult.cpp     MultRedu.out
submit-c ~/cs575/pro6 1001$ cat Mult.err
 1048576          2        524288         0.955 GigaMultsPerSecond
 2097152          2       1048576         1.058 GigaMultsPerSecond
 4194304          2       2097152         1.118 GigaMultsPerSecond
```

Liang Zhao

933-667-879

zhaolia@oregonstate.edu

Hi there, this project is challenging to understand and a lot of work. It took me a long time to do this project. Although the project is challenging, I can better understand and use OpenCL. My performance unit uses GigaMultsPerSecond.

1. What machine you ran this on:

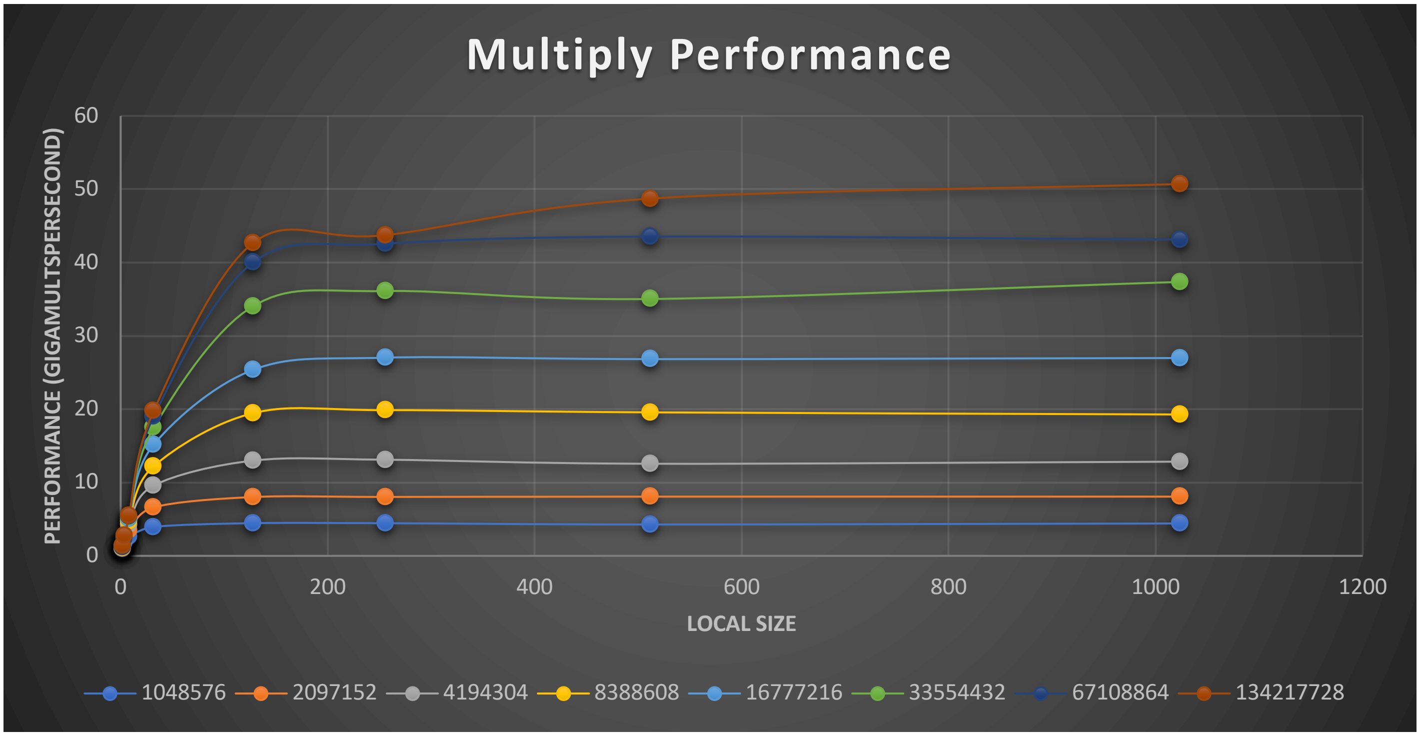
My program runs on the OSU college of Engineering DGX system.

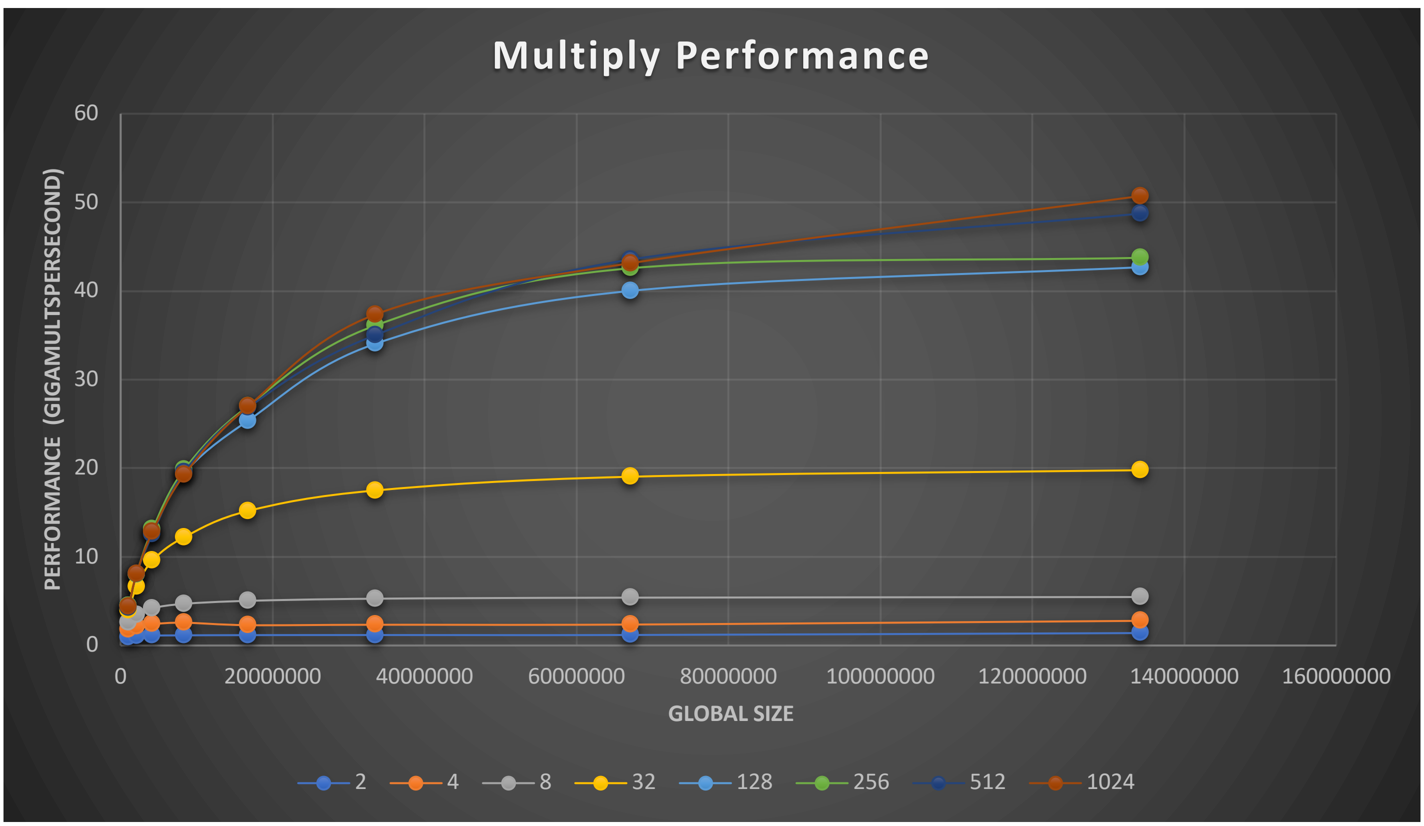
Array Multiply and the Array Multiply-Add:

2. Show the tables and graphs

   Array Multiply

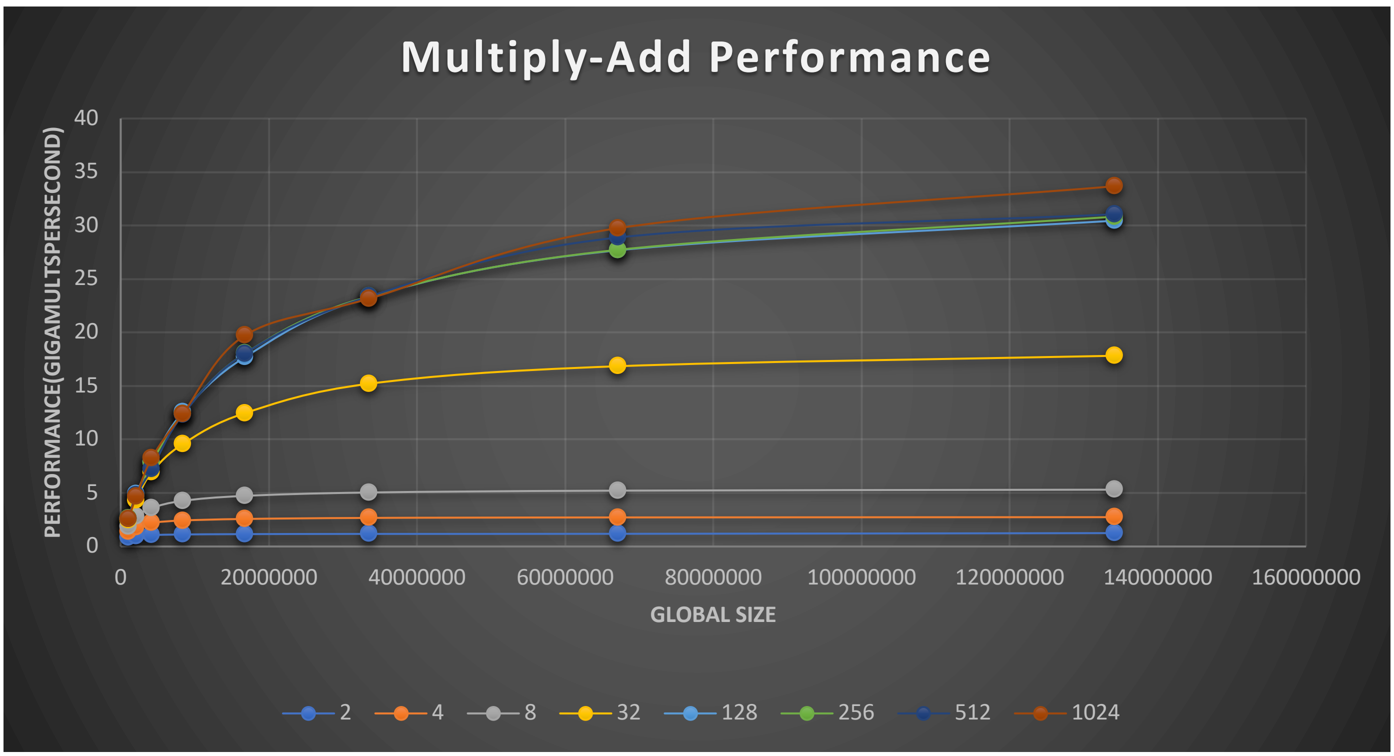
Local/Global	1048576	2097152	4194304	8388608	16777216	33554432	67108864	134217728
2	0.955	1.058	1.118	1.148	1.17	1.182	1.187	1.405
4	1.769	2.176	2.418	2.574	2.296	2.343	2.366	2.784
8	2.575	3.494	4.217	4.704	5.038	5.282	5.4	5.471
32	3.945	6.621	9.595	12.233	15.186	17.496	19.043	19.779
128	4.457	8.021	12.989	19.46	25.375	34.053	40.012	42.68
256	4.452	8.037	13.114	19.872	27.03	36.118	42.568	43.745
512	4.281	8.075	12.562	19.571	26.833	35.032	43.551	48.729
1024	4.418	8.076	12.848	19.28	26.982	37.364	43.138	50.713

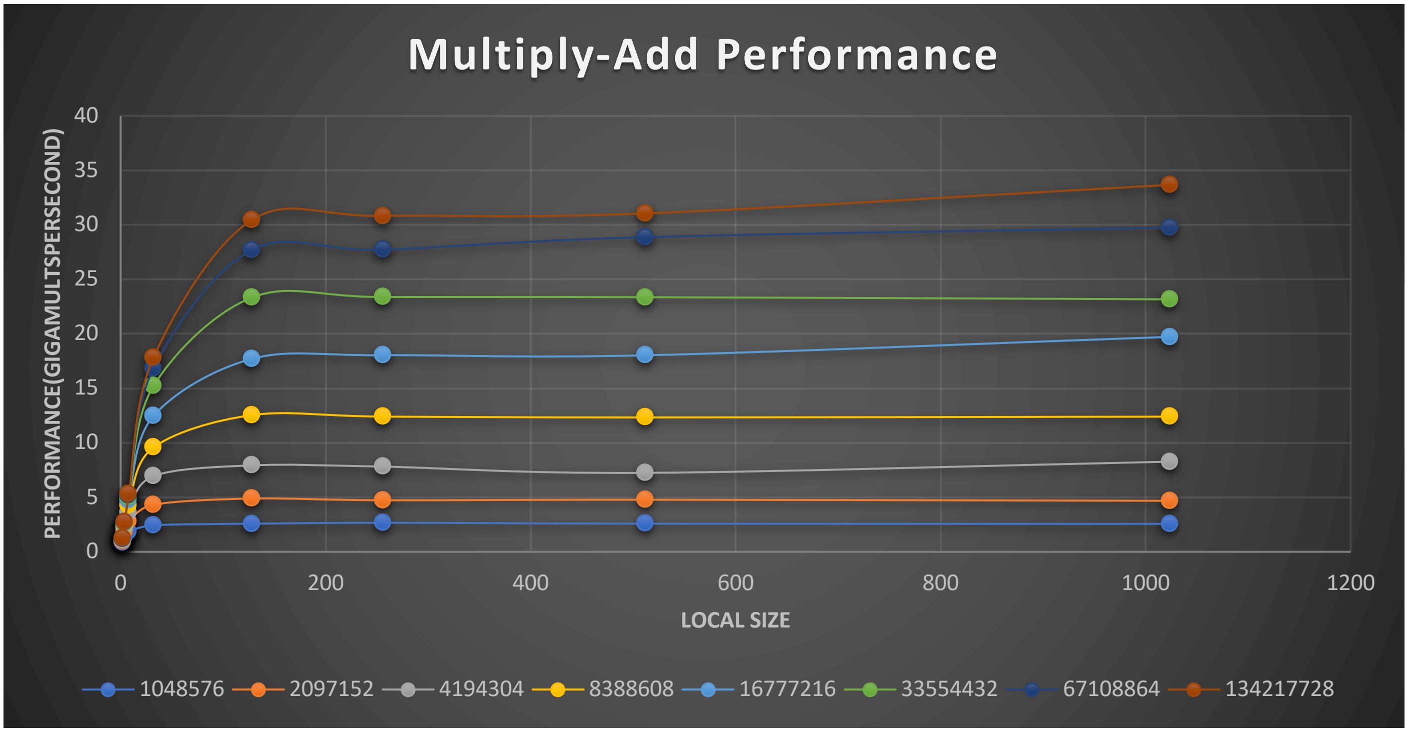

Multiply Performance
PERFORMANCE (GIGAMULTSPERSECOND)
0
10
20
30
40
50
60
0
200
400
600
800
1000
1200
LOCAL SIZE
1048576
2097152
4194304
8388608
16777216
33554432
67108864
134217728


Multiply Performance
PERFORMANCE (GIGAMULTSPERSECOND)
0
10
20
30
40
50
60
0
20000000
40000000
60000000
80000000
100000000
120000000
140000000
160000000
GLOBAL SIZE
2
4
8
32
128
256
512
1024

Array Multiply-Add:

Local/Global	1048576	2097152	4194304	8388608	16777216	33554432	67108864	134217728
2	0.84	0.987	1.067	1.122	1.156	1.176	1.185	1.251
4	1.392	1.829	2.212	2.43	2.573	2.671	2.717	2.742
8	1.874	2.803	3.602	4.269	4.731	5.046	5.219	5.311
32	2.435	4.322	6.943	9.578	12.469	15.214	16.849	17.827
128	2.583	4.89	7.933	12.543	17.707	23.331	27.688	30.453
256	2.666	4.735	7.82	12.411	18.044	23.38	27.728	30.831
512	2.589	4.783	7.247	12.335	18.035	23.368	28.877	31.057
1024	2.564	4.683	8.277	12.406	19.73	23.171	29.753	33.678





3. What patterns are you seeing in the performance curves?

   For a given global size, when the local size increases, the performance will rise; when the local size is 128, the performance will reach the highest. For a given local size, as the global workload increases, performance will grow.

4. Why do you think the patterns look this way?

   When the local workload is too small, more processing elements in the computing unit will be idle, and a lot of computing time is wasted. If the global workload is too tiny, the GPU will not be too busy, and the work done on the GPU is not enough to overcome the overhead of all settings.

5. What is the performance difference between doing a Multiply and doing a Multiply-Add?

The performance of Multiply is better than that of Multiply-Add. The core of Multiply-Add is more complicated than Multiply, so the processing time when performing Multiply-Add The GPU is longer.

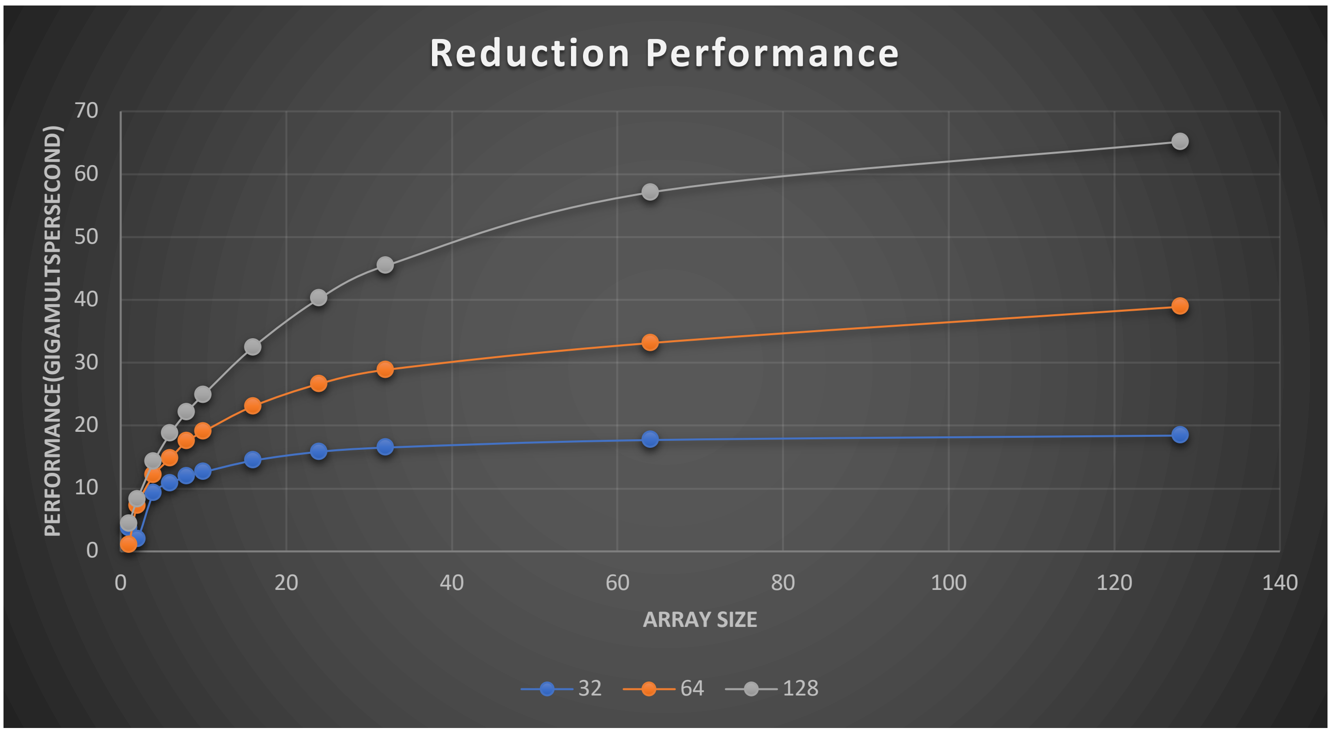
6. What does that mean for the proper use of GPU parallel computing?

According to the experiment, 128 local size is the right choice. If the data is too small, it is not worth doing on the GPU. Only when the data size is large enough, GPU parallel computing can overcome the set overhead.

Multiply+Reduce:

1. Show this table and graph

Local Size/Array Size	1	2	4	6	8	10	16	24	32	64	128
32	3.742	1.939	9.294	10.832	11.959	12.622	14.447	15.855	16.516	17.715	18.429
64	1.063	7.257	12.111	14.841	17.532	19.053	23.102	26.665	28.881	33.145	38.917
128	4.402	8.234	14.287	18.752	22.185	24.922	32.475	40.227	45.449	57.111	65.17



2. What pattern are you seeing in this performance curve?

The performance improves with the increase of the array size, and as the array size continues to increase, the performance tends to be stable.

3. Why do you think the pattern looks this way?

   When the array size is not very large, the GPU will not be busy, and the overhead may take too much time.

4. What does that mean for the proper use of GPU parallel computing?

   If the data size is too small, it is not worth doing it on GPU. Only when the data size is big enough can GPU parallel computing overcome the overhead of setting up. So for simple calculations such as sorting of small arrays, these problems do not use GPU parallel computing, but the running time will be slower.

Code:

```
// 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 NMB
// #define  NMB          64
// #endif

#ifndef NUM_ELEMENTS
#define NUM_ELEMENTS        64*1024*1024
#endif

#ifndef LOCAL_SIZE
#define LOCAL_SIZE      64
#endif

#define NUM_WORK_GROUPS     NUM_ELEMENTS/LOCAL_SIZE

const char *            CL_FILE_NAME = { "Mult.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, &stat
us );
    if( status != CL_SUCCESS )
        fprintf( stderr, "clCreateBuffer failed (1)\n" );

    cl_mem dB = clCreateBuffer( context, CL_MEM_READ_ONLY,  dataSize, NULL, &stat
us );
    if( status != CL_SUCCESS )
        fprintf( stderr, "clCreateBuffer failed (2)\n" );

    cl_mem dC = clCreateBuffer( context, CL_MEM_WRITE_ONLY, dataSize, NULL, &stat
us );
    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 d
evice buffers:

    status = clEnqueueWriteBuffer( cmdQueue, dA, CL_FALSE, 0, dataSize, hA, 0, NU
LL, NULL );
    if( status != CL_SUCCESS )
        fprintf( stderr, "clEnqueueWriteBuffer failed (1)\n" );

    status = clEnqueueWriteBuffer( cmdQueue, dB, CL_FALSE, 0, dataSize, hB, 0, NU
LL, 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 **)st
rings, 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, &s
ize );
        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, l
ocalWorkSize, 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" );

    // 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 inst
ead 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;
}


int
LookAtTheBits( float fp )
{
    int *ip = (int *)&fp;
```

```
    return *ip;
}


// wait until all queued tasks have taken place:

void
Wait( cl_command_queue queue )
{
      cl_event wait;
      cl_int      status;

      status = clEnqueueMarker( queue, &wait );
      if( status != CL_SUCCESS )
          fprintf( stderr, "Wait: clEnqueueMarker failed\n" );

      status = clWaitForEvents( 1, &wait );
      if( status != CL_SUCCESS )
          fprintf( stderr, "Wait: clWaitForEvents failed\n" );
}
```