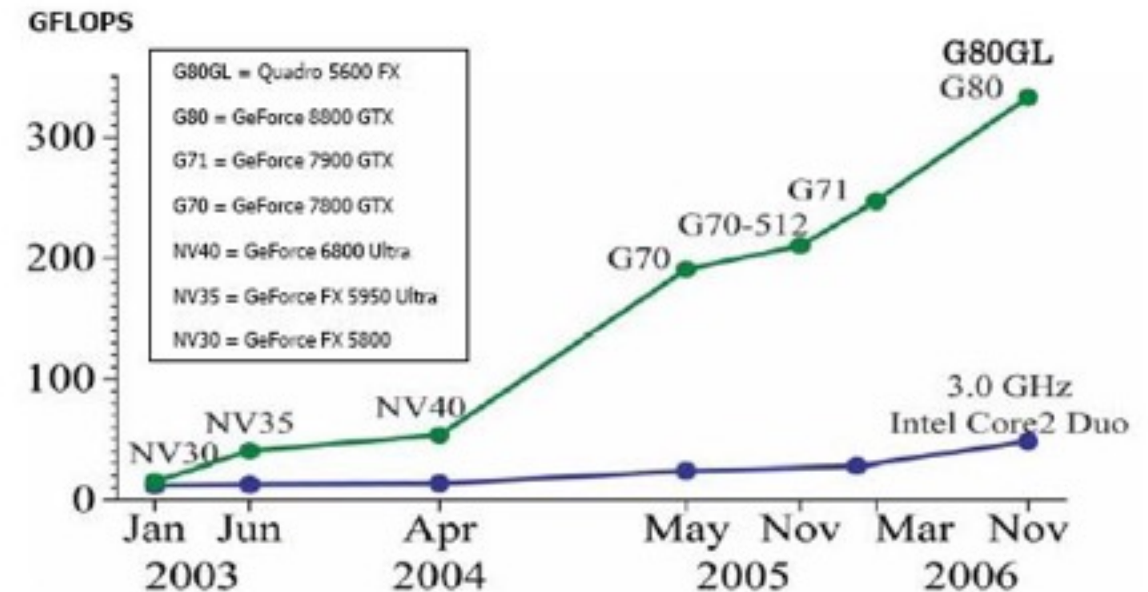
CSE 746 - Parallel and High Performance Computing
Lecture 7 - Introduction to OpenCL

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo
ppomorsk@sharcnet.ca
http://ppomorsk.sharcnet.ca/

GPU computing timeline



before 2003 - Calculations on GPU, using graphics API
2005 - Steady increase in CPU clock speed comes to a halt, switch to multicore chips to compensate. At the same time, computational power of GPUs increases
November, 2006 - CUDA released by NVIDIA
November, 2006 - CTM (Close to Metal) from ATI
December 2007 - Succeeded by AMD Stream SDK
December, 2008 - Technical specification for OpenCL1.0 released
April, 2009 - First OpenCL 1.0 GPU drivers released by NVIDIA
August, 2009 - Mac OS X 10.6 Snow Leopard released, with OpenCL 1.0 included
September 2009 - Public release of OpenCL by NVIDIA
December 2009 - AMD release of ATI Stream SDK 2.0 with OpenCL support
March 2010 - Cuda 3.0 released, incorporating OpenCL
June 2010 - OpenCL 1.1 specified
August 2011 - ATI switched to OpenCL
November 2011 - OpenCL 1.2 specified
November 2013 - OpenCL 2.0 specified

OpenCL (Open Compute Language) is an open standard for parallel programming of heterogenous systems, managed by Khronos Group.

Aim is for it to form the foundation layer of a parallel computing ecosystem of platform-independent tools

OpenCL includes a language for writing kernels, plus APIs used to define and control platforms

Targeted at GPUs, but also multicore CPUs and other hardware (Cell etc.), though to run efficiently each family of devices needs different code

OpenCL is largely derived from CUDA (no need to reinvent the wheel)

Same basic functioning : kernel is sent to the accelerator “compute device” composed of “compute units” of which “processing elements” work on “work items”.

Some names changed between CUDA and OpenCL

Thread -> Work-item
Block -> Work-group

Both CUDA and OpenCL allow for detailed low level optimization of the GPU

OpenCL device independent - really?

Actually, there is no way to write code that will run equally efficiently on both on GPU and multi-CPU architecture

However, OpenCL can detect the features of the architecture and run code appropriate for each

OpenCL code should run efficiently on many GPUs with a bit of fine tuning

On the other hand, an OpenCL code highly optimized for NVIDIA hardware will not run that efficiently on AMD hardware

As High Performance Computing code needs to be highly optimized, so OpenCL may not offer practical ability to be device independent

Advantages of running on multiple devices

The more machines can run code, the better

Some devices are much better than others for certain problems

NVIDIA and ATI GPUs can have widely different performance for certain types of problems

Eg. recent ATI GPUs had a major advantage in speed of certain types of integer operations, which made them much better for some digital coin mining computations

More fundamental differences in architecture

Two similar generation GPU cards from ATI and NVIDIA

NVIDIA GTX580 - 1544 MHz, 512 execution units (CUDA cores), in 16 multiprocessors, each with 32 cores

Radeon HD 5870 - 850 MHz, 1600 execution units (shaders), in 20 multiprocessors, each with 16 thread processors, each core with 5 execution units in VLIW (Very Large Instruction Word) architecture

GTX 580 runs 512 threads at 1544 MHz, each thread one instruction per cycle, same instruction for each 32 threads in a multiprocessor

HD 5870 runs 320 threads at 850 Mhz, each thread **five** instructions per cycle, same instructions for each of 16 threads in a multiprocessor

HD 5870 will show higher performance **if** problem fits well with the VLIW architecture, that is the problem must break down into groups of 5 completely independent instructions that can be done on a VLIW at the same time (from http://www.ece.lsu.edu/lpeng/papers/iiswc11.pdf)

CUDA vs. OpenCL

NVIDIA is fully supporting OpenCL even though it does not run exclusively on their hardware

The strategy is to increase the number of GPU users by making software more portable and accessible, which OpenCL is meant to do.   If there are more users, NVIDIA will sell more hardware.

As CUDA is fully controlled by NVIDIA, at any given time it contains more "bleeding edge" features than OpenCL, which is overseen by a consortium hence slower to incorporate new features

If you want your GPU code to run on both NVIDIA and AMD/ATI devices (still two main players at present), OpenCL is the only way to accomplish that

Available OpenCL environments

NVIDIA OpenCL
- distributed with CUDA since version 3.0
- no CPU support
- for NVIDIA GPU cards only

Apple OpenCL
- included as standard feature since Mac OS X 10.6 Snow Leopard (requires XCode)
- supports both graphics cards **and** CPUs (Apple hardware only)

AMD (ATI) OpenCL
- supports AMD GPU cards **and** CPUs

Intel OpenCL
- supports Intel CPUs and Integrated Graphics Processors (IGP)

OpenCL References

The OpenCL Programming Book - http://www.fixstars.com/en/opencl/book/

Khronos consortium: http://www.khronos.org/

CUDA OpenCL: https://developer.nvidia.com/opencl

NVIDIA GPU Computing SDK code samples

On “monk”: /opt/sharcnet/cuda/4.1/sdk/OpenCL

Apple OpenCL : http://developer.apple.com/opencl

AMD (ATI) OpenCL : http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/

OpenCL on SHARCNET

Part of CUDA, so available on same systems that CUDA is installed.

OpenCL SDK discontinued as part of CUDA SDK but it’s still available for download.

4 viz stations have ATI cards: viz7-uwo, viz9-uwo, viz10-uwo, viz11-uwo

Intel Phi machine should have OpenCL

Know your hardware

OpenCL programs should aim to be hardware agnostic, since one of the goals of OpenCL is to have programs run on multiple devices

The program should find the relevant system information at runtime

OpenCL provides methods for this

At minimum programmer must determine what OpenCL devices are available and choose which are to be used by the program

Sample program to do this follows
Source code on monk: /home/ppomorsk/CSE746_lec7/get_info/get_opencl_information.c

Program to get information - What is the platform?

```
#include <stdio.h>
#include <CL/cl.h>

int main(int argc, char** argv) {
    char dname[500];
    cl_device_id devices[10];
    cl_uint num_devices,entries;
    cl_ulong long_entries;
    int d;
    cl_int err;
    cl_platform_id platform_id = NULL;
    size_t p_size;

/* obtain list of platforms available */
    err = clGetPlatformIDs(1, &platform_id,NULL);
    if (err != CL_SUCCESS)
    {
        printf("Error: Failure in clGetPlatformIDs,error code=%d \n",err);
        return 0;
    }

/* obtain information about platform */
    clGetPlatformInfo(platform_id,CL_PLATFORM_NAME,500,dname,NULL);
    printf("CL_PLATFORM_NAME = %s\n", dname);

    clGetPlatformInfo(platform_id,CL_PLATFORM_VERSION,500,dname,NULL);
    printf("CL_PLATFORM_VERSION = %s\n", dname);
```

Program to get information cont. - What are the devices?

```
/* obtain list of devices available on platform */
    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ALL, 10, devices, &num_devices);
    printf("%d devices found\n", num_devices);

/* query devices for information */
    for (d = 0; d < num_devices; ++d) {
        clGetDeviceInfo(devices[d], CL_DEVICE_NAME, 500, dname,NULL);
        printf("Device #%d name = %s\n", d, dname);
        clGetDeviceInfo(devices[d],CL_DRIVER_VERSION, 500, dname,NULL);
        printf("\tDriver version = %s\n", dname);
        clGetDeviceInfo(devices[d],CL_DEVICE_GLOBAL_MEM_SIZE,sizeof(cl_ulong),&long_entries,NULL);
        printf("\tGlobal Memory (MB):\t%llu\n",long_entries/1024/1024);
        clGetDeviceInfo(devices[d],CL_DEVICE_GLOBAL_MEM_CACHE_SIZE,sizeof(cl_ulong),&long_entries,NULL);
        printf("\tGlobal Memory Cache (MB):\t%llu\n",long_entries/1024/1024);
        clGetDeviceInfo(devices[d],CL_DEVICE_LOCAL_MEM_SIZE,sizeof(cl_ulong),&long_entries,NULL);
        printf("\tLocal Memory (KB):\t%llu\n",long_entries/1024);
        clGetDeviceInfo(devices[d],CL_DEVICE_MAX_CLOCK_FREQUENCY,sizeof(cl_ulong),&long_entries,NULL);
        printf("\tMax clock (MHz) :\t%llu\n",long_entries);
        clGetDeviceInfo(devices[d],CL_DEVICE_MAX_WORK_GROUP_SIZE,sizeof(size_t),&p_size,NULL);
        printf("\tMax Work Group Size:\t%d\n",p_size);
        clGetDeviceInfo(devices[d],CL_DEVICE_MAX_COMPUTE_UNITS,sizeof(cl_uint),&entries,NULL);
        printf("\tNumber of parallel compute cores:\t%d\n",entries);
    }
    return 0;
}
```

Output on Apple MacBook laptop - both GPU and CPU seen

```
ppomorsk-mac:tryopencl pawelpomorski$ gcc -o test.x get_opencl_information.c  -w -m32 -lm -lstdc++ -
framework OpenCL
ppomorsk-mac:tryopencl pawelpomorski$ ./test.x
CL_PLATFORM_NAME = Apple
CL_PLATFORM_VERSION = OpenCL 1.0 (Feb 10 2010 23:46:58)
2 devices found
Device #0 name = GeForce 9400M
        Driver version = CLH 1.0
        Global Memory (MB):     256
        Global Memory Cache (MB):       0
        Local Memory (KB):      16
        Max clock (MHz) :       1100
        Max Work Group Size:    512
        Number of parallel compute cores:       2
Device #1 name = Intel(R) Core(TM)2 Duo CPU     P7350  @ 2.00GHz
        Driver version = 1.0
        Global Memory (MB):     1536
        Global Memory Cache (MB):       3
        Local Memory (KB):      16
        Max clock (MHz) :       2000
        Max Work Group Size:    1
        Number of parallel compute cores:       2
ppomorsk-mac:tryopencl pawelpomorski$
```

Getting a code to run on a GPU

Take existing serial program, separate out the parts that will continue to run on host, and the parts which will be sent to the GPU

GPU parts need to be rewritten in the form of kernel functions

Add code to host that manages GPU overhead: creates kernels, moves data between host and GPU etc.

OpenCL "Hello, World!" example

Not practical to do proper "Hello, World!" as OpenCL devices cannot access standard output directly

Our example program will pass an array of numbers from host to the GPU, square each number in the array on the GPU, then return modified array to host

This program is for learning purposes only, and will not run efficiently on a GPU

The program will demonstrate the basic structure which is common to all OpenCL programs

Will not show error checks on the slides for clarity, but having them is essential when writing OpenCL code

Source code

On monk: /home/ppomorsk/CSE746_lec7/hello/no_errorchecks_hello.c
/home/ppomorsk/CSE746_lec7/hello/hello.c

Program flow - OpenCL function calls

OpenCL function calls	Purpose
clGetPlatformIDs clGetDeviceIDs clCreateContext clCreateCommandQueue	Organize resources, create command queue
clCreateProgramWithSource clBuildProgram clCreateKernel	Compile kernel
clCreateBuffer clEnqueueWriteBuffer	Transfer data from host to GPU memory
clSetKernelArg clGetKernelWorkGroupInfo clEnqueueNDRangeKernel clFinish	Lauch threads running kernels on GPU, perform main computation
clEnqueueReadBuffer	Transfer data from GPU to host memory
clRelease...	Free all allocated memory

Kernel code - string with OpenCL C code

```
// Simple compute kernel which computes the square of an input array
//
const char *KernelSource = "\n" \
"__kernel void square(                                                  \n" \
"   __global float* input,                                              \n" \
"   __global float* output,                                             \n" \
"   const unsigned int count)                                           \n" \
"{                                                                      \n" \
"   int i = get_global_id(0);                                           \n" \
"   if(i < count)                                                       \n" \
"       output[i] = input[i] * input[i];                                 \n" \
"}                                                                      \n" \
"\n";
```

Define variables, set input data

```
int main(int argc, char** argv)
{
    int err;                            // error code returned from api calls

    float data[DATA_SIZE];              // original data set given to device
    float results[DATA_SIZE];           // results returned from device
    unsigned int correct;               // number of correct results returned

    size_t global;                      // global domain size for our calculation
    size_t local;                       // local domain size for our calculation

    cl_device_id device_id;             // compute device id
    cl_context context;                 // compute context
    cl_command_queue commands;          // compute command queue
    cl_program program;                 // compute program
    cl_kernel kernel;                   // compute kernel

    cl_mem input;                       // device memory used for the input array
    cl_mem output;                      // device memory used for the output array

    cl_platform_id platform_id = NULL;

    // Fill our data set with random float values
    //
    int i = 0;
    unsigned int count = DATA_SIZE;
    for(i = 0; i < count; i++)
        data[i] = rand() / (float)RAND_MAX;
```

Organise resources, create command queue

```
// determine OpenCL platform
err = clGetPlatformIDs(1, &platform_id,NULL);

// Connect to a compute device
//
int gpu = 1;
err = clGetDeviceIDs(platform_id, gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);

// Create a compute context
//
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

// Create a command commands
//
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

Compile kernel

```
// Create the compute program from the source buffer
//
program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource, NULL, &err);

// Build the program executable
//
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create the compute kernel in the program we wish to run
//
kernel = clCreateKernel(program, "square", &err);
```

Transfer data from host to GPU memory

```
// Create the input and output arrays in device memory for our calculation
//
input = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count, NULL, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, NULL);

// Write our data set into the input array in device memory
//
err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0, sizeof(float) * count, data, 0, NULL, NULL);
```

Launch threads running kernels on GPU, perform main computation

```
    // Set the arguments to our compute kernel
    //
    err = 0;
    err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
    err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);

    // Execute the kernel over the entire range of our 1d input data set
    // using one work item per work group (allows for arbitrary length of data array)
    //
    global = count;
    local = 1;
    err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local, 0, NULL, NULL);

    // Wait for the command commands to get serviced before reading back results
    //
    clFinish(commands);
```

Transfer data from GPU to host memory

```
// Read back the results from the device to verify the output
//
err = clEnqueueReadBuffer( commands, output, CL_TRUE, 0, sizeof(float) * count, results, 0, NULL, NULL );

// Validate our results
//
correct = 0;
for(i = 0; i < count; i++)
{
    if(results[i] == data[i] * data[i])
        correct++;
}

// Print a brief summary detailing the results
//
printf("Computed '%d/%d' correct values!\n", correct, count);
```

Free all allocated memory

```
    // Shutdown and cleanup
    //
    clReleaseMemObject(input);
    clReleaseMemObject(output);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);

    return 0;
}
```

If you are trying to use an OpenCL profiler and it crashes at the end of the run, not freeing all memory could be the cause

Compiling kernel - Online approach

In our example we have compiled code at runtime (online compile)

Kernel code can be specified as a string in the main program file

Kernel code can also be stored in a file (.cl) and loaded at runtime

```
    const char fileName[] = "./kernel.cl";

    /* Load kernel source code */
    fp = fopen(fileName, "r");
    if (!fp) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }
    source_str = (char *)malloc(MAX_SOURCE_SIZE);
    source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp );
    fclose( fp );
// ...

    /* Create Kernel program from the read in source */
    program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
```

Compiling kernel - Online approach

Code compiled at runtime may help with portability, but it makes debugging somewhat harder. If the compiler encounters an error, the build fails but error is not shown by default. Even correct code may fail if some environment variables are not set. On AMD devices, kernels using double precision variables require setting CL_KHR_FP64=1

```
if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS)
{
    printf("Error building program \n");

    printf("buildlog output \n");
    size_t len;
    char buffer[2048];
    clGetProgramBuildInfo(program, devices[device_used], CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer,&len);
    /* print error log */
    printf("%s\n", buffer);

    return 1;
}
```

Compiling kernel - Offline approach

It is possible to compile OpenCL source code (kernels) offline

This is the approach used in CUDA

Upside: need not spend time for compiling during runtime

Downside: executable not portable, need to compile separate binary for each device the code is running on

There is no freestanding compiler for OpenCL (like nvcc for CUDA)

Kernels have to be compiled at runtime and the resulting binary saved.

That binary can then be loaded in the future to avoid compiling step

Task queue

Queue is used to launch kernels, in precisely controlled order if required.

Event objects contain information about whether various operations have finished.

For example, clEnqueueTask launches a single kernel, after checking the list of provided event object whether they have completed, and returns its own event object.

```
//create quque enabled for out of order (parallel) execution
commands = clCreateCommandQueue(context, device_id, OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);
...
// no synchronization
clEnqueueTask(command_queue, kernel_E, 0, NULL,NULL);

// synchronize so that kernel E starts only after kernels A,B,C,D finish
cl_event events[4];  // define event object array
clEnqueueTask(commands, kernel_A, 0, NULL, &events[0]);
clEnqueueTask(commands, kernel_B, 0, NULL, &events[1]);
clEnqueueTask(commands, kernel_C, 0, NULL, &events[2]);
clEnqueueTask(commands, kernel_D, 0, NULL, &events[3]);
clEnqueueTask(commands, kernel_E, 4, events, NULL);
```

Queueing Data Parallel tasks

clEnqueueNDRangeKernel - used to launch data parallel tasks, i.e. copies of kernel which are identical except for operating on different data

For example, to lauch #(global) work-items (copies of kernel, i.e. threads) grouped into work-groups of size #(local), one would use:

```
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local, 0, NULL, NULL);
```

#(global) must be divisible by #(local), maximum size of #(local) dependent on device

Each work item can retrieve:
get_global_id(0) - its index among all threads
get_local_id(0) - its index among threads in its work-group
get_group_id(0) - index of its work group

This information can be used to determine which data to operate on

Organizing work-items in more than 1D

Convenient and efficient for many tasks, 2D and 3D possible

```
//2D example
// launch 256 work-items, organised in 16x16 grid
// grouped in groups of 16, organised as 4x4
global[0]=16; global[1]=16;
local[0]=4; local(1)=4;

err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL, &global, &local, 0, NULL, NULL);
```

Maximum possible values in global and local arrays are device dependent

For 2D, can retrieve global index coordinates
(x,y) = (get_global_id(0),get_global_id(1))

For 3D, can similarly retrieve global
(x,y,z) = (get_global_id(0),get_global_id(1),get_global_id(2))

Measuring execution time

OpenCL is an abstraction layer that allows the same code to be executed on different platforms

The code is guaranteed to run but its speed of execution will be dependent on the device and type of parallelism used

In order to get maximum performance, device and parallelism dependent tuning must be performed.

To do this, execution time must be measured. OpenCL provides convenient ways to do this

Event objects can also contain information about how long it took for the task associated with even to execute

clGetEventProfilingInfo can obtain start and end time of event

Taking the difference gives event duration in nanoseconds

```
    commands = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &err);
    cl_event event;
    err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local, 0, NULL, &event);

    clWaitForEvents(1, &event);
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
    printf("Time for event (ms): %10.5f  \n", (end-start)/1000000.0);
// ...
    ClReleaseEvent(event);  //important to free memory
```

Must minimize memory transfers between host and GPU as these are quite slow.

If you have a kernel which does not show any performance improvement when run on GPU, run it on GPU if doing so eliminates need for device to host memory transfer

Intermediate data structures should be created in GPU memory, operated on by GPU, and destroyed without ever being mapped or copied to host memory

Because of overhead, batching small transfers into one works better than making each transfer separately.

Higher bandwidth can be achieved when using page-locked (or pinned) memory.

Exercise: Port a serial code to use OpenCL

```
cp -rp /home/ppomorsk/CSE746_lec7/  .
```

./get_info/ - diagnostic code, has readme file, run this first
./hello/ - example code, consult, copy/paste from there as needed
./exercise/serial/ - serial code you are to convert to OpenCL
./exercise/opencl/ - the solution (made available later)

The goal is to work through the standard step of running an OpenCL application, so the kernel can be serial, and use only 1 work item (thread)

This port will be fairly generic, and could easily run on both GPU and CPU

```
void moving_average(int *values,
                    float *average,
                    int length,
                    int width)
{
   int i;
   int add_value;

   /* Compute sum for the first "width" elements */
   add_value = 0;
   for( i = 0; i < width; i++ ) {
       add_value += values[i];
   }
   average[width-1] = (float)add_value;

   /* Compute sum for the (width) to (length-1) elements */
for( i = width; i < length; i++ ) {
       add_value = add_value - values[i-width] + values[i];
       average[i] = (float)(add_value);    }

   /* Insert zeros to 0th to (width-2)th element */
   for( i = 0; i < width-1; i++ ) {
       average[i] = 0.0f;
   }

   /* Compute average from the sum */
   for( i = width-1; i < length; i++ ) {
        average[i] /= (float)width;
   }
}
```

Computes Simple Moving Average (SMA)

$$\ldots, p_M, p_{M-1}, \ldots, p_{M-(width-1)}, \ldots, p_1$$

$$SMA_M = \frac{p_M + p_{M-1} + \cdots + p_{M-(width-1)}}{width}$$

$$SMA_{\mathrm{M}} = SMA_{\mathrm{M}-1} - \frac{p_{M-n}}{width} + \frac{p_M}{width}$$

Hint: Converting this to an OpenCL kernel that runs as single thread requires very little modification

Serial host code

```
#include <stdio.h>
#include <stdlib.h>
/* Read Stock data */
int stock_array1[] = {
    #include "stock_array1.txt"
};
/* Define width for the moving average */
#define WINDOW_SIZE (13)

int main(int argc, char *argv[])
{
    float *result;

    int data_num = sizeof(stock_array1) / sizeof(stock_array1[0]);
    int window_num = (int)WINDOW_SIZE;
    int i;

    /* Allocate space for the result */
    result = (float *)malloc(data_num*sizeof(float));

    /* Call the moving average function */
    moving_average(stock_array1,result,data_num,window_num);

    /* Print result */
    for(i=0; i<data_num; i++) {
        printf( "result[%d] = %f\n", i, result[i] );
    }

    /* Deallocate memory */
    free(result);

    return 0;
}
```

OpenCL C language

OpenCL C

OpenCL C is basically standard C (C99) with some extensions and restrictions. This language is used to program the kernel.

There is a list of restrictions, for example
- recursion is not supported
- the point passed as an argument to a kernel function must be of type __global, __constant or __local
- support for double precision floating-point is currently an optional extension, and it may not be implemented

Some OpenCL functions are built in without requiring library, including math functions, geometric functions, and work-item functions

OpenCL includes some new features not available in C

Vector Data

In OpenCL can define and operate on vector data-types, which is a struct consisting of many components of same data-type (CUDA has capacity to do this but it's more limited)

float4 consists of (float,float,float,float) declared via
float4 f4;

Vector sizes are 2,4,8,16, can consist of char,int,float, double etc.

```
// vector literals used to assign values to variables
float4 g4 = (float4)(1.0f,2.0f,3.0f,4.0f);

// built in math functions can process vector types
float4 f4 = cos(g4);
// f4 = cos g(0),cos g(1), cos g(2), cos g(3)
```

Accessing vector data

```
// .xyzw will access 0 to 3, same in CUDA

int4 x = (int4)(1,2,3,4);
int4 a = x.wzyx // a=(4,3,2,1)
int2 b = x.xx // b = (1,1) */

// can access through number index using .s followed by number
// (in hex, which uses A-F for indices above 9)

int 8 c = x.s01233210 // c=(1,2,3,4,4,3,2,1)

// extract odd and even indices (.odd , .even)

int 2 d =x.odd // d=(2,4)

// extract upper half and lower half of vector (.hi,.lo)

int 2 e =x.hi // e = (3,4)
```

Other vector operations

Addition, subtraction

Comparison operation will return a vector containing TRUE or FALSE values is returned

This type of the vector values is determined by what the operands are. It's not usually char for char, and integer of same length as compared scalar values (short/int/long)

The actual values for FALSE/TRUE are 0/-1 and not 0/1, as that is more convenient for SIMD units

```
int2 tf = (float2)(1.0f,1.0f) == (float2) (0.0f,1.0f);
// tf = (int2) (0,-1)
```

Effective use of SIMD

Single Instruction obviously implies that it is essential for each thread to run exactly the same code as much as possible

Thus, if I have an if statement that will send some threads down one branch of the code and some other threads down another branch, that is not good.

Example of selection without using if

```
int a = in0[i], b = in1[i]; /*want to put smaller in out[i]*/
/*usual way with if */
if (a<b) out[i]=a;
else     out[i]=b;
// or possibly
// out[i]=a<b?a:b;
```

-- better way --

```
int a = in0[i], b = in1[i];
int cmp = a < b; //if TRUE, cmp has all bits 1, if FALSE all bits 0
// & bitwise AND
// | bitwise OR
// ~ flips all bits
out[i] = (a&cmp) | (b&~cmp);  //a when TRUE and b when FALSE

// OpenCL has shortct for this
out[i]=bitselect(cmp,b,a);
```

Works because: a&(all 1s) = a, a|(all 0s)=a
Good illustration why -1 for TRUE works better

Vector Benefits

Using vectors will only have benefits on hardware with SIMD features

If those features are not there, vectors will be handled sequentially with no speedup

The selection of vector size is important. For example, double16 may be too big to fit into some registers, so speedup will not occur

Ideally all this should be handled automatically by the compiler, but we are not yet there

OpenCL Pragmas

#pragma OPENCL FP_CONTRACT on-off-switch

Control how floating point is done (is FMA used). In contract operations multiple operations are performed as 1.

#pragma OPENCL EXTENSION <extension_name> : <behavior>

A number of extensions are available

OpenCL Memory Types

__global - memory which can be read from all work items (threads), though with relatively slow access unless the program is carefully optimized. It is the device's main memory.

__local - can be read from work items within a work group. Physically it is the shared memory on each compute unit.

__private - can only be used within each work item. Physically it is the registers used by each processing element.

__constant - memory that can be used to store constant data for read-only access by all compute units. The host is responsible for initializing and allocating this memory

OpenCL vs Cuda - Vector addition example

```
__global__ void
vectorAdd(const float * a, const float * b, float * c)
{
// Vector element index
int nIndex = blockIdx.x * blockDim.x + threadIdx.x;
c[nIndex] = a[nIndex] + b[nIndex];
}
```

CUDA

```
__kernel void
vectorAdd(__global const float * a,
__global const float * b,
__global       float * c)
{
// Vector element index
int nIndex = get_global_id(0);
c[nIndex] = a[nIndex] + b[nIndex];
}
```

OpenCL