OpenCL for programming GPUs

Sharcnet Summer School 2010

Pawel Pomorski
Overview

What is OpenCL?

Basic OpenCL
- Detecting your environment
- Basic building blocks - “Hello World!” program
- Hands on exercise - convert a “serial” CPU code to OpenCL GPU code
- OpenCL C

“Advanced” OpenCL
- Memory optimization
GPU computing timeline

before 2003 - Calculations on GPU, using graphics API
2003 - Brook “C with streams”
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
Near future - Intel may release a GPGPU (Larrabee)
Future - CPUs will have so many cores they will start to be treated as GPUs?
Why are GPUs fast?

Different paradigm, data parallelism (vs. task parallelism on CPU)

Stream processing

Hardware capable of launching MANY threads

However, GPUs also have significant limitations and not all code can run fast on them

If a significant portion of your code cannot be accelerated, your speedup will be limited by Amdahl’s Law
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.
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, it is possible that in the future it will contain 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 (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 at the moment
- for NVIDIA GPU cards only

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

**AMD (ATI) OpenCL**
- supports only AMD GPU cards
- no CPU support at the moment

**IBM OpenCL**, supports IBM Blade Center hardware

**Foxc** (Fixstars OpenCL Cross Compiler), in Beta as of June, 2010, CPU only
OpenCL References


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


NVIDIA GPU Computing SDK code samples

  On “angel”:  /opt/sharcnet/cuda/3.0/gpucomputingsdk/OpenCL

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

AMD (ATI) OpenCL :  http://ati.amd.com/technology/streamcomputing/opencl.html
OpenCL on SHARCNET

Cluster “angel” is SHARCNET’s main GPU computing cluster.

Hardware: 22 nodes, each with 8 CPU cores, each node “sees” 2 Tesla T10 GPUs

CUDA 3.0 drivers needed for OpenCL are currently available on node ang4 via a special arrangement for Summer School

Students can test OpenCL programs by connecting to ang4 via “ssh ang4” (from angel login node)

Run interactively, as ang4 is not accessible through queues right now

Compile with:
gcc -o test.x your_code.c -lOpenCL
Know your hardware

OpenCL programs should aim to be hardware agnostic

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
#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 - What is the platform?

On angel cluster: /work/ppomorsk/SUMMER_SCHOOL/get_info/get_opencl_information.c
Program to get information cont. - What are the devices?

```c
/* 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("Driver version = %s\n", dname);
    clGetDeviceInfo(devices[d], CL_DEVICE_GLOBAL_MEM_SIZE, sizeof(cl_ulong), &long_entries, NULL);
    printf("Global 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("Global 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("Local Memory (KB): \t%llu\n", long_entries/1024);
    clGetDeviceInfo(devices[d], CL_DEVICE_MAX_CLOCK_FREQUENCY, sizeof(cl_ulong), &long_entries, NULL);
    printf("Max clock (MHz): \t%llu\n", long_entries);
    clGetDeviceInfo(devices[d], CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t), &p_size, NULL);
    printf("Max Work Group Size: \t%ld\n", p_size);
    clGetDeviceInfo(devices[d], CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), &entries, NULL);
    printf("Number of parallel compute cores: %d\n", entries);
}
return 0;
```
Output on angel - 2 GPU detected, no CPUs detected

[ppomorsk@ang4:~] cd /work/ppomorsk/SUMMER_SCHOOL/get_info
[ppomorsk@ang4:/work/ppomorsk/SUMMER_SCHOOL/get_info] gcc -o test.x get_opencl_information.c -lOpenCL
[ppomorsk@ang4:/work/ppomorsk/SUMMER_SCHOOL/get_info] ./test.x
CL_PLATFORM_NAME = NVIDIA CUDA
CL_PLATFORM_VERSION = OpenCL 1.0 CUDA 3.0.1
2 devices found
Device #0 name = Tesla T10 Processor
    Driver version = 195.36.15
    Global Memory (MB): 4095
    Global Memory Cache (MB): 0
    Local Memory (KB): 16
    Max clock (MHz) : 1440
    Max Work Group Size: 512
    Number of parallel compute cores: 30
Device #1 name = Tesla T10 Processor
    Driver version = 195.36.15
    Global Memory (MB): 4095
    Global Memory Cache (MB): 0
    Local Memory (KB): 16
    Max clock (MHz) : 1440
    Max Work Group Size: 512
    Number of parallel compute cores: 30
[ppomorsk@ang4:/work/ppomorsk/SUMMER_SCHOOL/get_info]
Angel architecture

GPU server: Tesla S1070

[Image]

http://archive.electronicdesign.com/files/29/19280/fig_01.gif

4 GPUs on server

Each GPU has 30 compute units, each consisting of 8 stream processing cores

240 cores in total on one GPU

Each node “sees” 2 GPUs
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 angel cluster: /work/ppomorsk/SUMMER_SCHOOL/hello/no_errorchecks_hello.c
/work/ppomorsk/SUMMER_SCHOOL/hello/hello.c
Program flow - OpenCL function calls

- **Organize resources, create command queue**
  - `clGetPlatformIDs`
  - `clGetDeviceIDs`
  - `clCreateContext`
  - `clCreateCommandQueue`

- **Compile kernel**
  - `clCreateProgramWithSource`
  - `clBuildProgram`
  - `clCreateKernel`

- **Transfer data from host to GPU memory**
  - `clCreateBuffer`
  - `clEnqueueWriteBuffer`

- **Launch threads running kernels on GPU, perform main computation**
  - `clSetKernelArg`
  - `clGetKernelWorkGroupInfo`
  - `clEnqueueNDRangeKernel`
  - `clFinish`

- **Transfer data from GPU to host memory**
  - 

- **Free all allocated memory**
  - `clRelease...`
// Simple compute kernel which computes the square of an input array
//
const char *KernelSource = "\n" __kernel void square(  
   __global float* input,  
   __global float* output,  
   const unsigned int count)  
{"  
   int i = get_global_id(0);  
   if(i < count)  
      output[i] = input[i] * input[i];  
"}  
"\n";}
Define variables, set input data

```c
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

```c
// 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

```c
// 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);
```
Lauch threads running kernels on GPU, perform main computation

```c
// 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

```c
// 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

```c
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 - 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.

```c
//create queue 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 launch \( #(\text{global}) \) work-items (copies of kernel, i.e. threads) grouped into work-groups of size \( #(\text{local}) \), one would use:

\[
\text{err} = \text{clEnqueueNDRangeKernel} (\text{commands}, \text{kernel}, 1, \text{NULL}, \&\text{global}, \&\text{local}, 0, \text{NULL}, \text{NULL});
\]

\( #(\text{global}) \) must be divisible by \( #(\text{local}) \), maximum size of \( #(\text{local}) \) 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
Organising work-items in more than 1D

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

```c
//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 similary 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

```c
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
```
Timing “Hello World!”

Increase DATA_SIZE to 1048576

Check how computation time varies with size of work-groups

<table>
<thead>
<tr>
<th>Work-group size (local)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.191</td>
</tr>
<tr>
<td>8</td>
<td>0.803</td>
</tr>
<tr>
<td>16</td>
<td>0.414</td>
</tr>
<tr>
<td>32</td>
<td>0.209</td>
</tr>
<tr>
<td>64</td>
<td>0.127</td>
</tr>
<tr>
<td>512 (maximum)</td>
<td>0.130</td>
</tr>
</tbody>
</table>

Using largest possible work group is usually the best choice

Time for write/read to buffer is 2.51/5.14 ms.
Timing “Hello World!” with openclprof

/opt/sharcnet/cuda/3.0/cuda/openclprof/bin/openclprof
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.
Running openclprof on SHARCNET

1. ssh -X yourusername@angel.sharcnet.ca
2. ssh -X ang4
3. /opt/sharcnet/cuda/3.0/cuda/openclprof/bin/openclprof

Once you get the GUI:
4. File->New
5. Name your project, select location somewhere in your work space
6. Select OpenCL binary executable compiled previously (no special compile options needed)
7. Select your working directory
8. Click “Start” to perform profiling run

Examine output
Try various options in session and view
Exercise: Port a serial code to use OpenCL

cp -rp /work/ppomorsk/SUMMER_SCHOOL/ .
./exercise/serial - serial code you are to convert to OpenCL
./exercise/opencl_template - contains a generic “template” which you can modify, using the serial code, to develop the OpenCL version of the code
./hello - consult if needed
./exercise/opencl_solution - the solution (peek if you must)

Compile with: gcc -o test.x main.c -lOpenCL
To run: ./test.x

Template is for the case of lunching one kernel as a single 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)

\[ SM_{M, n} = \frac{p_M + p_{M-1} + \cdots + p_{M-(width-1)}}{width} \]
\[ SM_{M, n} = SM_{M-1} - \frac{p_{M-n}}{width} + \frac{p_M}{width} \]

Hint: Converting this to an OpenCL kernel that runs as single thread requires very little modification
```c
#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 a much more limited capacity to do this)

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.

---

```c
// 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

```c
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

```c
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 --

```c
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: \(a\&(\text{all 1s}) = a\), \(a\&(\text{all 0s})=a\)
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 optimised. 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

CUDA

OpenCL

```c
__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];
}
```

```c
__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];
}
```
Optimizing OpenCL

- Efficient access to global memory

- Use of local memory to increase bandwidth: Matrix Multiplication example

Code, data and figures from:
Nvidia OpenCL - Best Practices Guide
GPU memory vs. CPU memory

CPUs have a cache which for many problems can provide fast memory access, with transfers between cache and main memory handled in hardware.

The cache paradigm scales poorly as the number of cores is increased, as increased hardware overhead is needed to maintain coherency between caches attached to different cores.

In a GPU, with its hundreds of cores, caches are not practical and thus are not present.

Fast but small local (shared) memory is somewhat analogous to L1 cache on a CPU. However, there is no hardware to automatically shift data from local to global memory as needed. This must be taken care of in software, i.e. by the programmer.

In general on a CPU the requirements for memory access are not that stringent. If you are accessing memory sequentially, you are almost certainly OK.

On GPU efficient memory access requires more work by the programmer.
Coalesced Access to Global Memory

Global memory loads and stores by threads of half warp (16 threads) can be combined into as few as one transaction.

You can think of global memory as rows of 16 floats linked together into a 64 byte group, with 2 of these forming a 128 byte group.

Good access pattern results in a single 64 bit transaction.

In this example, if accesses were permuted, still one transaction would be performed on newer cards (CC 1.2), but 16 serialized transaction on older cards (CC 1.1).
Somewhat misaligned pattern
CC 1.1 devices will perform transactions in series
CC 1.2 can still do it in one

Memory to be accessed now belongs to two different 128 byte groups, so CC 1.2 devices need 2 transactions

Choosing work-group sizes to be multiples of 16 facilitates efficient access patterns
Cost of misaligned access

Even on newer cards consecutive but misaligned access can decrease bandwidth by a factor of 2
Strided accesses

```c
__kernel void strideCopy(__global float* odata,
    __global float* idata,
    int stride)
{
    int xid = get_global_id(0) * stride;
    odata[xid] = idata[xid];
}
```

Stride=2 case, CC > 1.3 devices can do this in a single 128 byte transaction, but bandwidth is wasted as half of elements not used

Effective bandwidth deteriorates as stride is increased to 16 and beyond
Note that performance deteriorates significantly as stride increases to just 16.

On CPU the cache with its fast memory is relatively large and thus can provide reasonable performance for larger stride.

GPUs have no cache, but they have __local memory (in OpenCL, same as shared memory in CUDA), which can be used to improve stride access efficiency.
Optimizing matrix multiplication with use of local memory

Consider special case of $AB=C$

A is (Mx16)
B is (16xN)
C is (MxN)

Use work-group of 16x16
Unoptimized

With this code, the number of times global memory must be accessed to provide needed entries of a and b is \( N \times M \times (16 + 16) \) times.

```c
__kernel void simpleMultiply(__global float* a,
__global float* b,
__global float* c,
int N)
{
    int row = get_global_id(1);
    int col = get_global_id(0);
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```
Partially optimized
Store A in local memory
Global memory is accessed
N*M*(1+16 times)

```cpp
__kernel void coalescedMultiply(__global float* a,
__global float* b,
__global float* c,
int N,
__local float aTile[TILE_DIM][TILE_DIM])
{
    int row = get_global_id(1);
    int col = get_global_id(0);
    float sum = 0.0f;
    int x = get_local_id(0);
    int y = get_local_id(1);
    aTile[y][x] = a[row*TILE_DIM+x];
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[y][i]* b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Fully optimized
Store A in local memory
Global memory is accessed
N*M*(1+1) times

```cpp
__kernel void sharedABMultiply(__global float* a,
__global float* b,
__global float* c,
int N,
__local float aTile[TILE_DIM][TILE_DIM],
__local float bTile[TILE_DIM][TILE_DIM])
{
    int row = get_global_id(1);
    int col = get_global_id(0);
    float sum = 0.0f;
    int x = get_local_id(0);
    int y = get_local_id(1);
    aTile[y][x] = a[row*TILE_DIM+x];
    bTile[y][x] = b[y*N+col];
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[y][i]* bTile[i][x];
    }
    c[row*N+col] = sum;
}
```
# Performance

<table>
<thead>
<tr>
<th>Optimization</th>
<th>NVIDIA GeForce GTX 280 (released June 2008)</th>
<th>NVIDIA GeForce GTX 8800 (released November 2006)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>8.8 GBps</td>
<td>0.7 GBps</td>
</tr>
<tr>
<td>Coalesced using shared memory to store a tile of A</td>
<td>14.3 GBps</td>
<td>8.2 GBps</td>
</tr>
<tr>
<td>Additionally using shared memory to store a tile of B</td>
<td>29.7 GBps</td>
<td>15.7 GBps</td>
</tr>
</tbody>
</table>

The trend is for newer cards to cope better with poor optimization.
Conclusion

Explore OpenCL to find out more of its features, as this introduction was only an overview with many aspects left out

Experiment with OpenCL on angel (full install very soon)

Contact SHARCNET HPC analysts for advice about porting your code to GPU

Keep up with new developments in the rapidly developing field of GPU computing