OpenCL: Programming Heterogeneous Architectures

Porting BigDFT to OpenCL

Brice Videau

October 20, 2011

Introduction

Needs in computing resources are infinite

Benefits for physicists and chemists

More computing power means :

- Bigger systems,
- Fewer approximations,
- Improved accuracy.

Numerical experimentation.

CEA's hybrid cluster Titane, built by Bull

Current and future architectures

2 trends are seen in current calculators :

Bigger systems

- Number of nodes in clusters and grids increase.
- Number of processors in supercomputer increase.

More powerful components

- Increased frequency,
- Increased number of processors and cores,
- Specialized co-processors : GPU, PPU, MIC...

Exploiting those architectures

- Middlewares are available to program those machines.
- Each middleware covers a range of usage.

Some examples

Distributed machines :

- MPI, ompss...

Multicore architectures :

- MPI, OpenMP, CILK, OpenCL...

GPU :

- OpenCL, NVIDIA Cuda...

Talk Outline

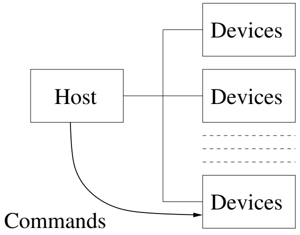
1. OpenCL : a Standard for Parallel Computing
2. Writing Kernels
3. BigDFT
4. Conclusions and perspectives
5. Coding tutorial

OpenCL : a Standard for Parallel Computing

OpenCL Architecture Model

Host-Devices model

- 1 host and several devices.
- Devices are connected to the host.
- Host issues commands to the devices.
- Data transport is done via memory copy.



All major actors support OpenCL

- NVIDIA and ATI GPU.
- IBM CELL processor.
- Intel CPU, AMD CPU.
- ARM platforms (CPU+GPU).

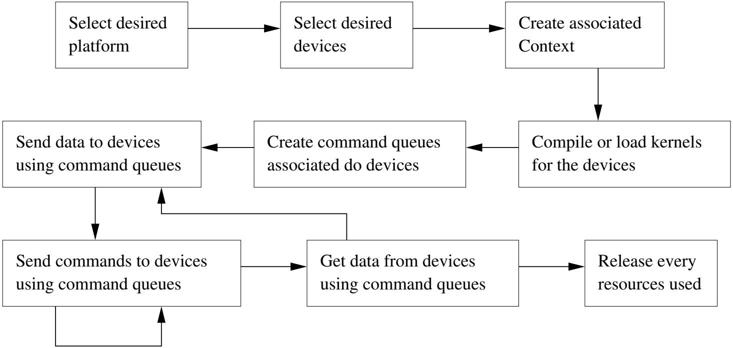
Context and Queues

- Contexts aggregate resources, programs and devices belonging to a common platform (ie NVIDIA, or ATI).
- Host and devices communicate via buffers defined in a context.
- Commands are sent to devices using command queues.
- Commands are called kernels.

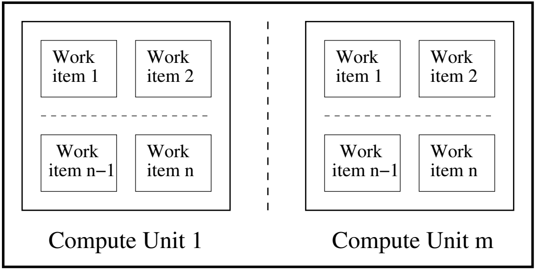
Command queues

- Can be synchronous or asynchronous.
- Can be event driven.
- Several queues can point to the same device, allowing concurrent execution.

OpenCL General Workflow



OpenCL Processing Model



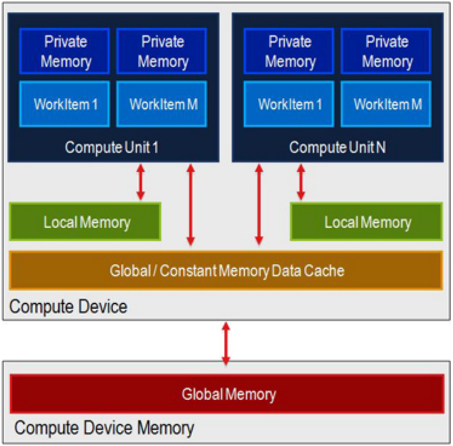
Compute Device

- Kernels are split into uni, two or three-dimensional ranges called work groups.
- Work groups are mapped to compute units.
- Individual item are processed by work items.

OpenCL Memory Model

4 different memory space defined on an OpenCL device :

- Global memory : corresponds to the device RAM, input data are stored there.
- Constant memory : cached global memory.
- Local memory : high speed memory shared among work items of a compute unit.
- Private memory : registers of a work item.



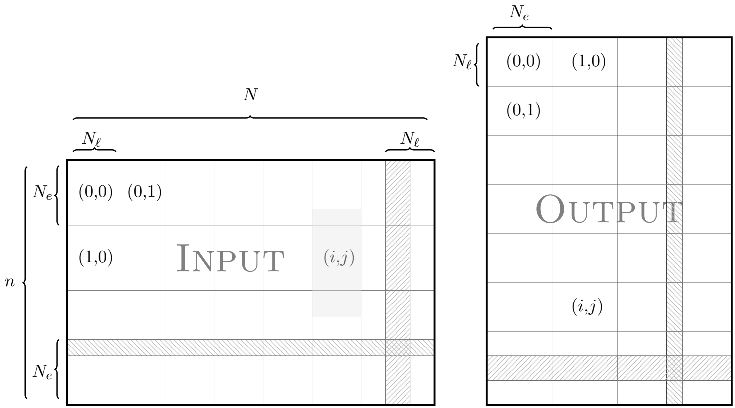
Writing Kernels

OpenCL Language : a Subset of C

Kernels are written using a C-like language

- Recursion is prohibited
- Helper functions are defined
  - Barriers
  - Work item indexes
  - Atomic operations
  - Vector operations
- New keywords :
  - `__kernel`
  - `__global`, `__local`, `__constant`, `__private`
  - `__read_only`, `__write_only`, `__read_write`

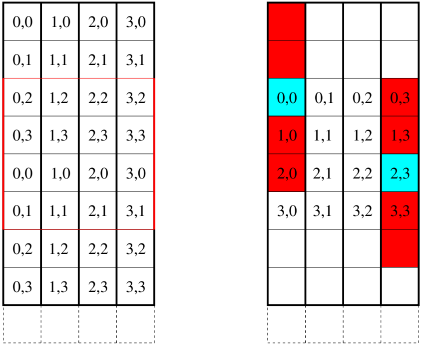
Example : Unidimensional Convolutions



One unidimensional convolution with transposition, simple but not too much. Magicfilter code used in BigDFT.

OpenCL memory management

Example using a 4*4 block processing a filter of length 5.



1 work item processes one element of the final matrix.

Benefits from this approach

- 3D convolutions can be expressed as a succession of 3 1D convolution/transposition.
- Memory access are coalesced while reading input matrix and writing output matrix.
- Bank conflicts are *almost* avoided by padding the buffer to the number of bank $+1$.
- Maps easily to current architectures.
- Extends to more complex convolutions found in BigDFT.
- Almost every convolution in BigDFT have been ported, including free boundary and semi-periodic boundary.

Kernel Declaration

Kernel Declaration

```
/* Activate double precision support */
#pragma OPENCL EXTENSION cl_khr_fp64: enable
#define FILT_W 16
#define WG_S 16
__kernel void
__attribute__((reqd_work_group_size(WG_S,WG_S,1)))
magicfilter1dKernel_d(uint n, uint ndat,
                      __global const double *psi,
                      __global double *out){
//padded local buffer size : 33*16
__local double tmp[WG_S*(WG_S+FILT_W+1)];
```

- Works on double precision floats
- Kernel expects work group size of 16 x 16
- `n` and `ndat` are in `__local` memory
- `tmp1` is a storage buffer in local memory, shared among work items

Work with Indexes

Get Indexes and Load Data

```
//get our position in the result matrix
const size_t ig = get_global_id(0);
const size_t jg = get_global_id(1);
//get our position in the local work group
const size_t i = get_local_id(0);
const size_t j = get_local_id(1);
//transpose indexes in the work group in order to read transposed data
ptrdiff_t igt = ig - i + j - FILT_W/2;
ptrdiff_t jgt = jg - j + i;
//if we are on the outside, select a border element to load, wrapping around
//we will be loading 2 elements each
if ( igt < 0 )
  tmp[i * (WG_S+FILT_W+1) + j] = psi[jgt + ( n + igt ) * ndat];
else
  tmp[i * (WG_S+FILT_W+1) + j] = psi[jgt + igt * ndat];
igt += FILT_W;
if ( igt >= n )
  tmp[i * (WG_S+FILT_W+1) + j + FILT_W] = psi[jgt + ( igt - n ) * ndat];
else\n\
  tmp[i * (WG_S+FILT_W+1) + j + FILT_W] = psi[jgt +  igt * ndat];
```

Compute Convolution and Write Output

Performing Computations NVIDIA

```
//initialize result
double tt = 0.0;
//rest position in the buffer to first element involved in the convolution
tmp += j2*(WG_S+FILT_W+1) + i2;
//wait for buffer to be full
barrier(CLK_LOCAL_MEM_FENCE);

//apply filter
tt += *tmp++ * FILT0;
tt += *tmp++ * FILT1;
/* ... */
tt += *tmp++ * FILT15;
//store the result
out[(jg*n+ig)]=tt;
};
```

Compute Convolution and Write Output

Performing Computations AMD

```
//initialize result
double2 tt = (double2)(0.0, 0.0);
//rest position in the buffer to first element involved in the convolution
tmp += j2*(WG_S+FILT_W+1) + i2;
//wait for buffer to be full
barrier(CLK_LOCAL_MEM_FENCE);

//apply filter
__local double2 *tmp2= (__local double2 *)tmp;
tt += *tmp2++ * (double2)(FILT0,FILT1);
/* ... */
tt += *tmp2++ * (double2)(FILT14,FILT15);
//store the result
out[(jg*n+ig)]=tt.x+tt.y;
};
```

BigDFT OpenCL Port Evaluation

Motivations

Part of BigDFT was already ported to GPU architectures using CUDA, why a new port ?

- Only part of the program was ported.
- OpenCL can target several platforms while CUDA can't.
- OpenCL is a standard, while CUDA is a vendor provided solution.
- This time most of BigDFT is expected to run on GPU.
- Code ported is relatively simple and well suited to GPU, thus the performance loss is hoped to be minimal.

Objectives

- Most of BigDFT operations can be expressed as unidimensional convolutions.
- There are several dozens convolutions to implement.
- Convolutions of BigDFT share common traits.

Objectives

- Find a common parallelization technique fitting most convolutions.
- Be as efficient as CUDA.

Test System Setup

GPU 2 :

- Tesla C2070 (Fermi)
- 6 GB of RAM
- Driver version : 260.14

GPU 2 :

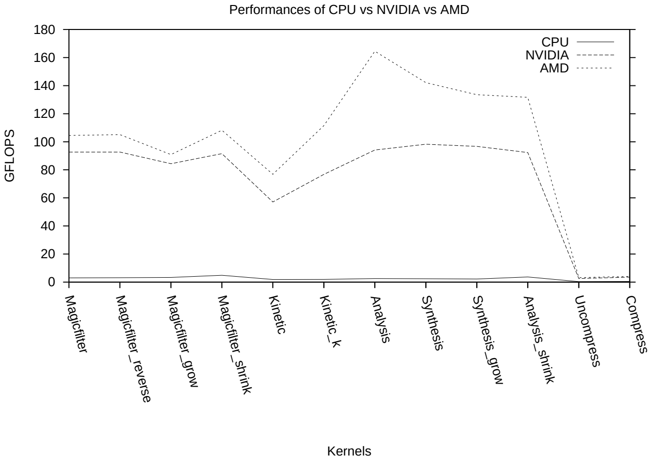
- Radeon HD6970
- 2 GB of RAM
- Driver version : 11.6

Test System Setup

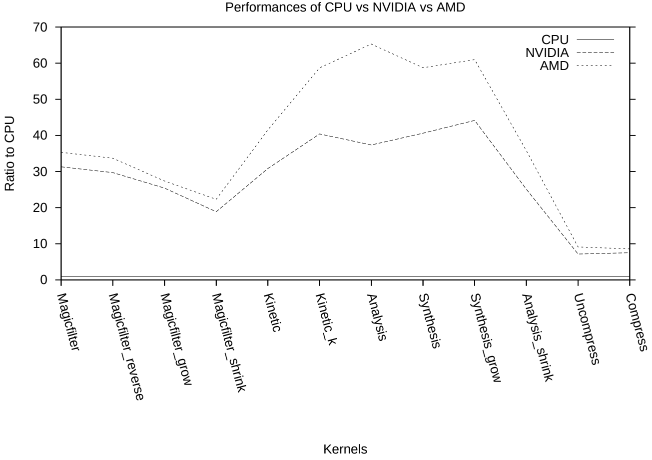
Host :

- Lenovo D20
- 1 Xeon 5550 @ 2.83 GHz (4 Nehalem cores)
- 8 GB of RAM
- Linux 2.6.38-11 x86_64
- icc 11.1

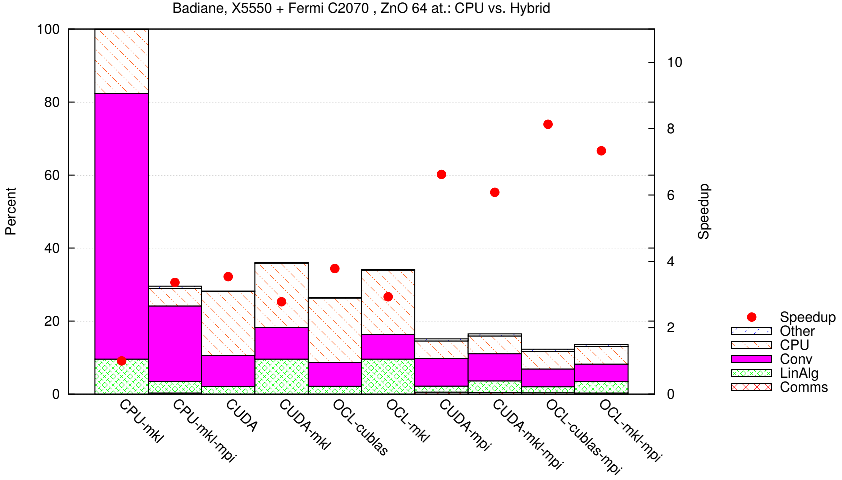
Comparison CPU,Fermi,HD6970



Comparison CPU,Fermi,HD6970



Comparison CUDA, OpenCL, CPU



Hybrid and heterogeneous runs with OpenCL

Graphene, 4 C atoms, 52 kpts

MPI+NVIDIA/AMD	Execution Time (s)	Speedup
1	6020	1
4	1660	3.6
1 + NVIDIA	300	20
4 + NVIDIA	160	38
1 + AMD	347	17
4 + AMD	197	30
(4 + NV) + (4 + AMD)	109	55

TABLE: Performance results for different configuration of BigDFT, using MPI + GPUs

Conclusions

OpenCL

- OpenCL proved easy to use.
- Performance is on-par with previous CUDA implementation.
- Kernels have been shown to run on other architectures : ATI and CPU.

BigDFT

- Full port of BigDFT convolutions on OpenCL.
- Part of the code was rewritten during this work.
- Complexity reduced compared to the CUDA support.
- Portability across GPUs.

Perspectives

OpenCL

- Some OpenCL implementations are still buggy.
- Best way to do multi-GPU, GPU+OpenCL CPU ?
- Optimizing kernels for CPU architectures ? Intel, ARM.
- Automated kernel generation.
- OpenCL : an easy way to write vectorized code ?

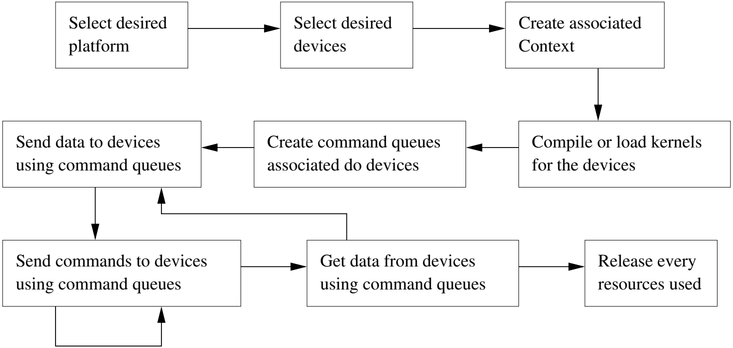
Questions ?

Thanks for your attention.

Life and Death of OpenCL in a Program

The Host Side of OpenCL

General Workflow



Platform Selection

In a near future every platform will support OpenCL, but the user may not be interested in all of them : select an appropriate platform

Get Platforms

```
#include <CL/cl.h>
cl_uint num_platforms;
clGetPlatformIDs( NULL, NULL, &num_platforms);
cl_platform_id *platforms = malloc(sizeof(cl_platform_id) * num_platforms);
clGetPlatformIDs(num_platforms, platforms, NULL);
/*...*/
for(int i=0; i<num_platforms; i++){
  /*...*/
  clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, ... );
  /*...*/
}
```

Device Selection

Several device from the same vendor is also common : one device for the screen and one device for computations

Get Devices

```
#include <CL/cl.h>
cl_uint num_devices;
clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, NULL, NULL, &num_devices);
cl_device_id *devices = malloc(sizeof(cl_device_id) * num_devices);
clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, num_devices, devices, NULL);
/*...*/
for(int i=0; i<num_devices; i++){
  /*...*/
  clGetDeviceInfo(devices[i], CL_DEVICE_NAME , ... );
  /*...*/
}
```

Context Creation

Context gather devices from the same platform. Those devices will be able to share resources.

Create Context

```
cl_context_properties properties[] =
   { CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id, 0 };
cl_device_id devices[] = {device_id_1, device_id_2};
cl_context context =
   clCreateContext(properties, 2, devices, NULL, NULL, NULL);
```

A shortcut exists, skipping device selection :

Create Context from Type

```
cl_context_properties properties[] =
   { CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id, 0 };
cl_context context =
   clCreateContextFromType(properties, CL_DEVICE_TYPE_GPU , NULL, NULL, NULL);
```

Building Program from Source

Once the context is created, the program is to be built (or loaded from binary).

Building Program

```
/* strings is an array of string_count NULL terminated strings */
cl_program program =
   clCreateProgramWithSource(context, string_count, strings, NULL, NULL);
/* if device_list is NULL, program is built
 * for all available devices in the context */
clBuildProgram(program, num_devices, device_list, options, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "kernel_name", NULL);
```

Kernels are extracted from the built program using their name.

Creating Command Queues

A command queue is used to send commands to a device. They have to be associated with a device.

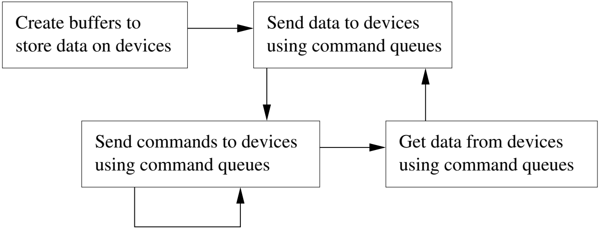
Creating Command Queues

```
cl_command_queue queue =
   clCreateCommandQueue(context, devices[chosen_device], 0, NULL);
```

Options can be specified instead of 0, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE allows for out of order execution for instance.

Using OpenCL

Using OpenCL is (hopefully) easier than setting it up.



Buffer Creation

In OpenCL buffers creation and deletion are explicitly managed. As can be noted buffers are tied to a context and not a particular command queue. The implementation is free to transfer buffers from devices to host memory or to another device.

Creating Simple Buffers

```
cl_mem read_buffer =
   clCreateBuffer(context, CL_MEM_READ_ONLY, buffer_size, NULL, NULL);
cl_mem write_buffer =
   clCreateBuffer(context, CL_MEM_WRITE_ONLY, buffer_size, NULL, NULL);
```

Pinned Buffer Creation

Pinned buffer creation can offer premium performances. Here is a code sample that can be used on NVIDIA devices. The finals pointers obtained can be used to transfer data between the host and the device.

Creating Pinned Simple Buffers

```
cl_mem pinned_read_buffer =
   clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY,
                  buffer_size, NULL, NULL);
cl_mem pinned_write_buffer =
   clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_WRITE_ONLY,
                  buffer_size, NULL, NULL);
unsigned char *data_in =
   clEnqueueMapBuffer(queue, pinned_read_buffer, CL_TRUE, CL_MAP_WRITE, 0,
                      buffer_size, 0, NULL, NULL, NULL);
unsigned char *data_out =
   clEnqueueMapBuffer(queue, pinned_write_buffer, CL_TRUE, CL_MAP_READ, 0,
                      buffer_size, 0, NULL, NULL, NULL);
```

Transferring Data

The implementation is free to move buffers in memory. But nonetheless, memory is often kept on the device associated to the command queue used to transfer the data.

Data Transfer

```
clEnqueueWriteBuffer(queue, read_buffer, CL_TRUE, 0,
                      buffer_size, data_in, 0, NULL, NULL);
/* Processing that reads read_buffer and writes write_buffer */
/*...*/
clEnqueueReadBuffer(queue, write_buffer, CL_TRUE, 0,
                     buffer_size, data_out, 0, NULL, NULL);
```

Performing Calculations

Once data is transfered, kernels are used to perform calculations.

Kernel Usage

```
/* Place kernel parameters in the kernel structure. */
clSetKernelArg(kernel, 0,sizeof(data_size), (void*)&data_size);
clSetKernelArg(kernel, 1,sizeof(read_buffer), (void*)&read_buffer);
clSetKernelArg(kernel, 2,sizeof(write_buffer), (void*)&write_buffer);
/* Enqueue a 1 dimensional kernel with a local size of 32 */
size_t localWorkSize[] = { 32 };
size_t globalWorkSize[] = { shrRoundUp(32, data_size) };
clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
                       globalWorkSize, localWorkSize, 0, NULL, NULL);
```

Event Management

Almost all functions presented end with :

```
        ..., 0, NULL, NULL);
```

These 3 arguments are used for event management, and thus asynchronous queue handling. Functions can wait for a number of events, and can generate 1 event.

```
event_t event_list[] = {event1, event2};
event_t event;
clEnqueueReadBuffer(queue, write_buffer, CL_FALSE, 0,
                    buffer_size, data_out, 2, event_list, &event);
```

Previous buffer read waits for 2 events and generate a third that will happen when the read is completed.

Release Resources

OpenCL uses reference counts to manage memory. In order to exit cleanly from an OpenCL program all allocated resources have to be freed :

- buffers (`clReleaseMemObject`)
- events (`clReleaseEvent`)
- kernel (`clReleasekernel`)
- programs (`clReleaseProgram`)
- queues (`clReleaseCommandQueue`)
- context (`clReleaseContext`)
- etc...