OpenCL – The Open Standard for Parallel Programming of Heterogeneous Systems

J. Y. Xu

Institute of Information
& Mathematical Sciences
Massey University at Albany,
Auckland, New Zealand
jyxu@jamesyxu.com

OpenCL, or Open Computing Language, is an open source API that allows for the development of portable code that take advantage of the parallel computing power of modern electronic hardwares, and has the support of a number of industrial heavy weights, making it an idea starting point for any parallel application from consumer software to HPC programs.

Keywords: OpenCL, parallel computing, GPGPU

1. Introduction

Today's computers are becoming increasing reliant on parallel computing. Processors are shipping with multiple cores, and new ways of computing are being discovered such as GPGPU (General Purpose Graphics Processing Unit) programming [1][3]. Each of these new processing techniques present advantages of their own, thus making the combined use of these technologies empirical. For example GPGPU allows for massive data parallel computations with improved floating point precision [2], while CPU allows for task parallel computation. Combining these two results in applications that are capable of massively parallel vector computations, while remain interactive to end users.

While presenting unique opportunities, these emerging trends also present a number of challenges. First of all, GPGPU programming evolved from graphical APIs, and as a result the language constructs are different to those of multi-core CPU programming. This means that a program written for a dual core CPU cannot be made to run on GPGPUs without extensive modification. Worse still, different vendors have their own proprietary technologies.

OpenCL, or Open Computing Language, is a prominent project aimed to bring uniformity to this new area. It is a set of foundation level APIs aimed to abstract the underlying hardware, and to provide a framework for building parallel applications [6]. Based on the popular ISO C99 standard commonly known as the C language, OpenCL can be viewed as a variant, thus making it a widely acceptable. The current supported hardwares range from

CPUs, GPUs, DSP (Digical Signal Processors) to mobile CPUs such as ARM. Through OpenCL, multiple tasks can be configured to run in parallel on all processors in the host system, and the resulting code is portable on a number of devices.

Initially developed by Apple, the project has gained support from all the major chip vendors such as AMD, Intel and Freescale. Recently submitted to the Khronos group (the group behind OpenAL, OpenGL), the current standard specification is 1.03.

In this paper, we will explore some of the fundamentals of the OpenCL specification. The rest of the paper is organized as follow. Section 2 of the paper provides an overview of the OpenCL design. Section 3 discusses the OpenCL archtecture. And finally section 4 explore the OpenCL runtime and it's software stack.

2. OpenCL Design Overview

OpenCL follows a "close-to-the-silicon" approach [5], meaning that it is as close to the hardware implementation as possible, with just enough abstraction to make the API vendor and hardware neutral.

In a OpenCL system, all computation resources in a host system are seen as peers. These resources can include CPUs, GPUs, mobile processors, microcontrollers and DSPs. OpenCL support both data and task parallel compute models, and have clearly defined floating point representation (IEEE 754 with specified rounding and error). The resultant executable is capable of executing on a number of devices, dynamically allocating computation resources available.

OpenCL defines a set of models and a software stack. The models specify how OpenCL is constructed, and how data and tasks are handled. The software stack on the other hand, shows a general development work flow, indicating how a developer should utilize the OpenCL libraries.

Other than the standard OpenCL, there is also a Embedded version of the specification, named OpenCL Embedded Profile [8]. Compared to the full specification, there are a number of notable differences

- Embedded profile devices do not support any 64 bit data. This means that there are no double, ulong, and some vector data types
- Embedded profile devices do not have to support 3D operations.
- IEEE 754 rounding requirement is not implemented as strictly, full IEEE 754 require the rounding to be to the nearest even number, however in Embedded profile, only the basic mode of IEEE 754 is supported, meaning all rounding can be done to the nearest zero

3. Architecture, Models

The OpenCL architecture can be described using 4 models [2][8][7]:

- Platform Model
- Execution Model
- Memory Model
- Programming Model

In OpenCL, the platform model describe the overall relationship between various OpenCL elements and the host system. As Fig 1 demonstrates, the host system is seen as a number of compute devices. These devices can be CPU, GPU, or any other micro processor that support the OpenCL framework (such as ARM processors). These compute devices are not organized in hierarchy, and are seen as equals. From here, each compute device will handle a number of compute units, which are processing elements grouped together. There are a number of advantages for grouping elements together into compute units, and these are covered in the discussion of other models.

The execution model dictates how a program is run on the host. OpenCL introduces two important concepts:

- Kernel, a kernel is the smallest unit of execution. Kernels are similar to a function in C
- Host program, a host program consist of a collection of kernels. By using these kernels, the host program can perform certain tasks

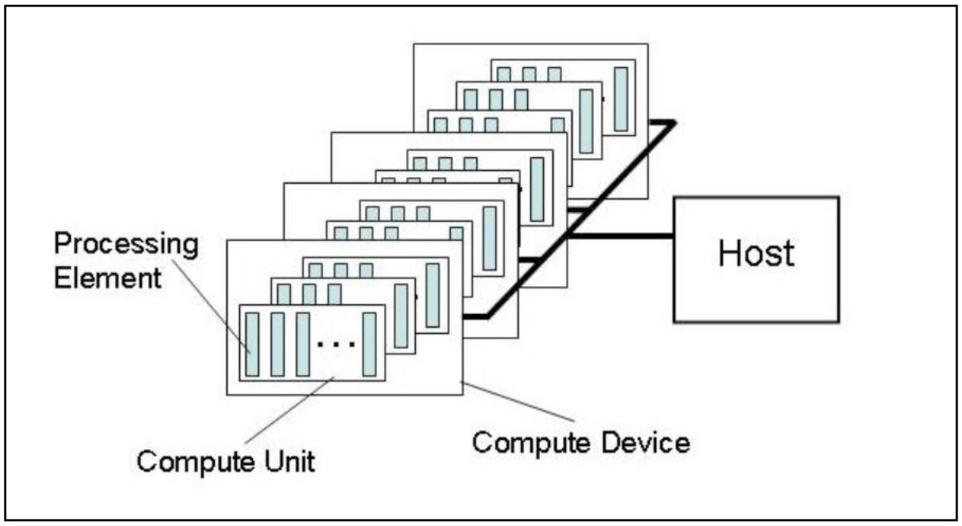
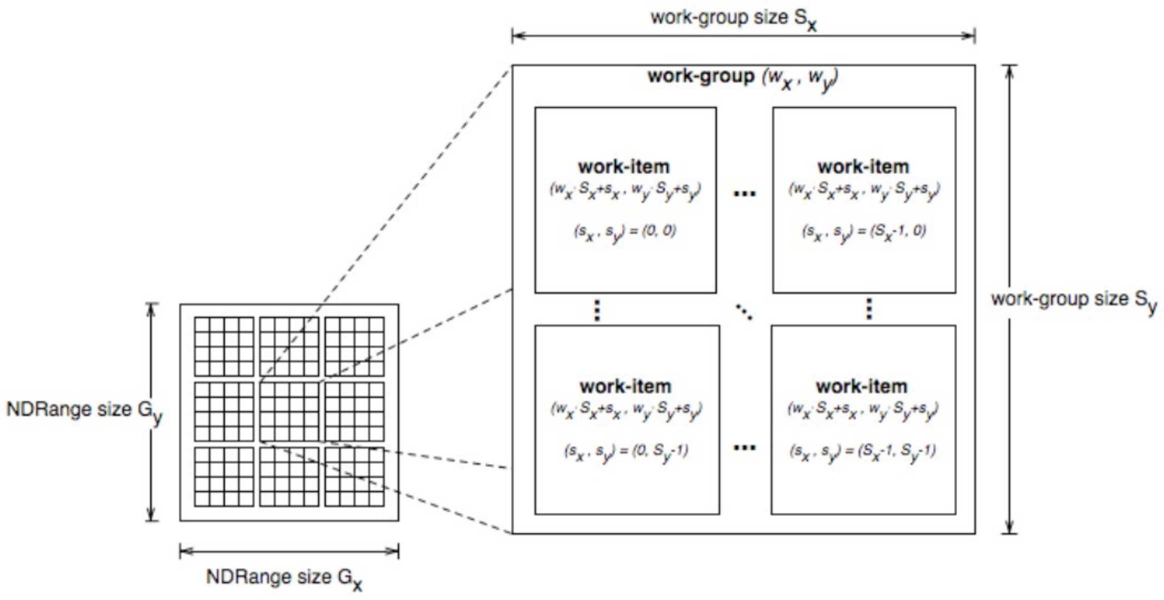


Fig 1 Platform model

When a host program is run, a number of kernels are invoked. These kernels are placed in a matrix known as the index space, or NDRange (Fig 2). Each element in the matrix is an instanced kernel, and these elements can be grouped to form work groups. Going back to the platform model, we see that each work group is assigned to a compute device, and the kernel instances in the group are the processing elements.

The grouping of the kernels logically separates the parallel tasks, and takes care of memory sharing. Elements may only communicate with other elements in the work group, and a work group has a single shared memory (memory model described later). Work groups can communicate with each other through the OpenCL framework.


work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x
Fig 2 OpenCL Execution model index space

The memory model (Fig 3) deployed in OpenCL is fairly straightforward, and is very similar to the way processes are handled in the operating system.

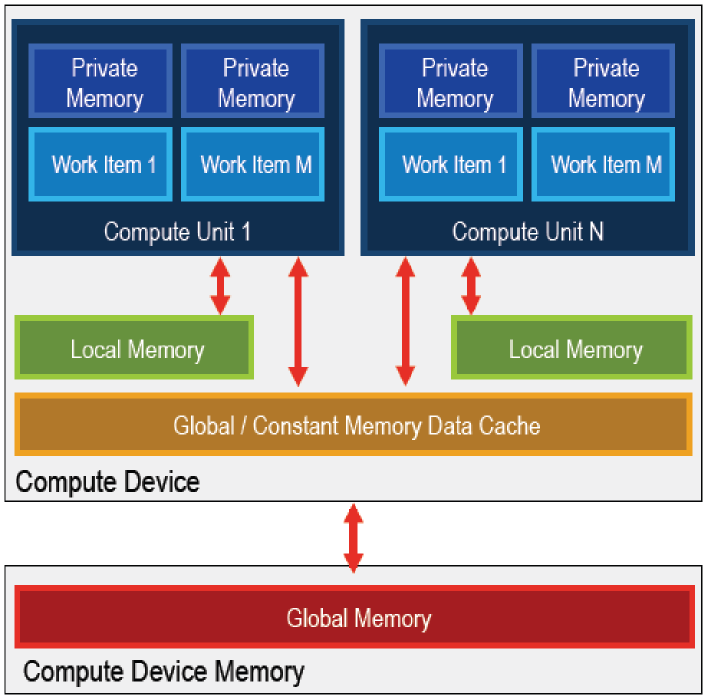


Fig 3 Memory model

Here a compute device's memory is shared by all the compute units as global memory. For a CPU this is the RAM, and for a GPU this would be the Video RAM. Within each compute unit, a local memory heap is provided, and each work item has a separate private memory. Comparing to the operating system's process memory model, we can clearly see the similarities. Where apart from the globally shared memory, each process has a local memory heap, and each thread within the process can also have their own private memory.

There are two programming models adopted in OpenCL, and they are the data parallel and task parallel programming models. Data parallel is where a sequence of data elements are fed into a work group [8]. For example, we can have a work group that consists of multiple instances of the same kernel, and map different data sets to them. Task parallel is where a work group is executed independently from all other work groups, further more, a compute unit and the work group may only contain a single instance of the kernel [8]. This means that within a compute device, we have all different kernels running in parallel, thus task parallel. According to the current specification, only some devices such as the CPU should implement the task parallel model, where as data parallel model is required on all OpenCL compliant devices.

4. OpenCL Runtime

Currently, OpenCL comes with C headers. The language is derived from ISO C99, which is the familiar standard C language. There are a few restrictions [2][5]:

- No recursion
- No function pointers

All preprocessing directives of C99 are supported, as well as all scalar and vector data types. And because of the close association between OpenCL and OpenGL, an image processing extension is included.

Apart from the programming language, the OpenCL runtime can be described using the software stack shown in Fig 4, the software stack consists of 3 layers. The bottom layer is the kernel layer, where a number of kernels are written and packaged, this is analogous to a set of C function headers. The runtime layer is responsible for creating kernel instances, mapping the correct memory space, synchronization and clean up. This layer is analogous to the functions of a DLL (dynamic link library). Finally at the top, the host program layer is where an overall context is created, and kernels are queued to run in a logical manner to solve a particular problem.

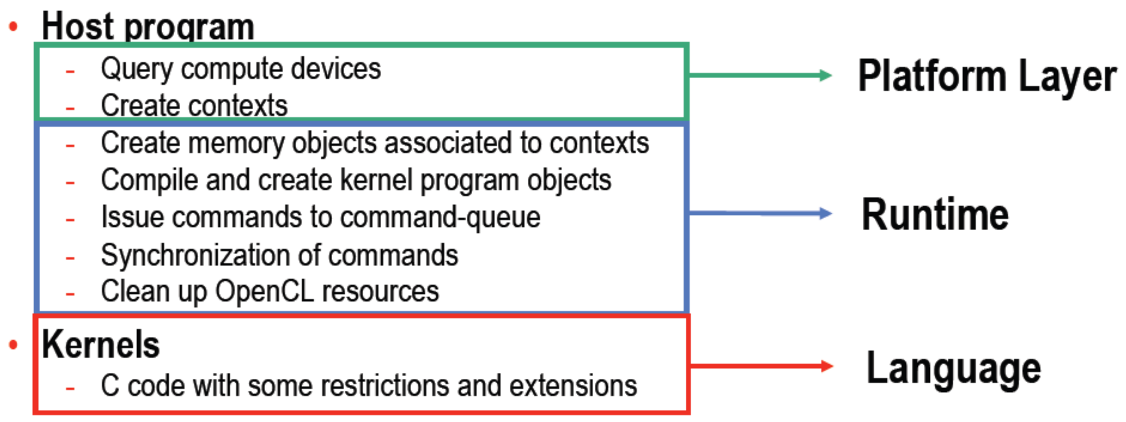

Host program
Query compute devices
Create contexts
Create memory objects associated to contexts
Compile and create kernel program objects
Issue commands to command-queue
Synchronization of commands
Clean up OpenCL resources
Kernels
C code with some restrictions and extensions
Platform Layer
Runtime
Language

Fig 4 Software stack

Overall, the basic programming flow is to:

- Query the system for available OpenCL compliant hardware
- Create a context for the program, specifying which compute devices to use
- Create memory objects that will be holding the data for computation
- Instantiate kernels
- Build a NDRange matrix that cane be used to register the kernels

- Create work groups if necessary
- Load kernels into program, map memory objects
- Execute program

To better understand the concepts of OpenCL, an analysis of a simple program is included [2][4], In this case a Fast Fourier Transform, or FFT program:

```
context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
```

- *The code above creates the context that would hold all the work units, we see that in this case, we are utilizing only the GPU*

```
queue = clCreateWorkQueue(context, NULL, NULL, 0);
```

- *A work queue is created, the queue holds kernel instances that are to be executed*

```
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(float)*2*num_entries, srcA);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(float)*2*num_entries, NULL);
```

- *Memory buffers are created to hold data*

```
program = clCreateProgramFromSource(context, 1, &fft1D_1024_kernel_src,
NULL);
clBuildProgramExecutable(program, false, NULL, NULL);
kernel = clCreateKernel(program, "fft1D_1024");
```

- *The function fft1D_1024_kernel_src kernel (function) is instanced here*

```
global_work_size[0] = n;
local_work_size[0] = 64;
range = clCreateNDRangeContainer(context, 0, 1, global_work_size,
local_work_size);
```

- *The code above creates the NDRange matrix, which is used to hold the kernel instances*

```
clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 2, NULL, sizeof(float)*(local_work_size[0]+1)*16,
NULL);
clSetKernelArg(kernel, 3, NULL, sizeof(float)*(local_work_size[0]+1)*16,
NULL);
clExecuteKernel(queue, kernel, NULL, range, NULL, 0, NULL);
```

- *Memory buffers are mapped, and the kernel is executed*

```
__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                          __local float *sMemx, __local float *sMemy) {
      int blockIdx = get_group_id(0) * 1024 + tid;
      float2 data[16];

      in = in + blockIdx;  out = out + blockIdx;
```

- *Just like CUDA programming, here we are finding out what global memory the current process is mapped to*

```
globalLoads(data, in, 64);
```

- *We load the data assigned to the current process, which is in, and the next 64 bytes. Notice that by using 4x16 byte numbers we are taking advantage of the coalesced read. GPU devices allow us to read a chunk of memory at a time without having to do multi reads*

```
fftRadix16Pass(data);
twiddleFactorMul(data, tid, 1024, 0);
localShuffle(data, sMemx, sMemy, tid,(((tid&15)*65) + (tid >> 4)));
fftRadix16Pass(data);
twiddleFactorMul(data, tid, 64, 4);
localShuffle(data, sMemx, sMemy, tid,(((tid>>4)*64) + (tid & 15)));
fftRadix4Pass(data);
fftRadix4Pass(data + 4);
fftRadix4Pass(data + 8);
fftRadix4Pass(data + 12);
```

- *The actual calculations are done here*

```
globalStores(data, out, 64);
```

- *We write the data back out to the global memory, starting from the start position of our designated block, and write 64 bytes. Again the 4x16 structure means that we do not have to write several times. The coalesced write will write the chunk of 64 byte in one cycle*

```
}
```

5. Conclusions

Through this article, we explored the architecture models of OpenCL, as well as its software stack. With more and more importance being placed on parallel computing, we need an effective framework that would present us with an uniform interface to all the computational hardwares within an end system. OpenCL allows for the development of portable code that take advantage of the parallel computing power of modern electronic hardwares, and has the support of a number of industrial heavy weights, making it an idea starting point for any parallel application from consumer software to HPC programs.

Reference

[1] – GPGPU, http://en.wikipedia.org/wiki/Gpgpu, Accessed 5th May

[2] – Munshi, A., OpenCL, parallel Computing on the GPU and CPU, *Siggraph 2008*

[3] – nVidia CUDA, http://www.nvidia.com/object/cuda_learn.html, Accessed 5th May

[4] – OpenCL, http://en.wikipedia.org/wiki/OpenCL, Accessed 5th May

[5] – OpenCL Presentation, http://www.khronos.org/developers/library/overview/opencl_overview.pdf, Accessed 5th May

[6] – OpenCL Project, http://www.khronos.org/opencl, Accessed 5th May

[7] – OpenCL Quick Reference Card, http://www.khronos.org/files/opencl-quick-reference-card.pdf, Accessed 5th May

[8] – The OpenCL Specification, http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf, Accessed 5th May