OpenCL and CUDA for GPGPU Acceleration

There are a number of options for using GPUs from your application, such as: precompiled GPU libraries or tools for particular problems, high level language and script bindings, and lower level C/C++ style languages for writing GPU programs "from scratch". On this page C/C++ style languages will be discussed. There are two main options:

- OpenCL
- CUDA

OpenCL and CUDA are very similar in their programming model and usage, and each uses different terminology to describe similar concepts. The main advantage of OpenCL is that it is a cross-platform open standard, while CUDA was limited to use on NVIDIA GPUs at the time of writing. The main advantage of CUDA is that it provides a higher level of abstraction and simplicity in the host side runtime API, while also providing a more verbose API set similar to OpenCL’s if required.

OpenCL

OpenCL (external link) is a multi-vendor supported library that at present supports GPUs from NVidia and AMD/ATI and CPUs from Intel, IBM and AMD. It is a C library, with bindings for many other languages available, including C++ (external link), Java (external link), Python (external link).

A useful reference book for understanding the structure and use of OpenCL is the "OpenCL Programming Guide" available through CSIRO library services (search link)

Installation of OpenCL

A version of OpenCL header files and libraries are available on CSIRO clusters from Intel. They are installed in standard system directories and as such there is no need to load any modules to make it available to user programs. There is also a version supplied by NVidia on the bragg-gpu cluster, accessible using “module load cuda” and using $CUDA_HOME/include and $CUDA_HOME/lib64 in make files.

Once compiled an OpenCL executable can support various devices for kernel execution (NVIDIA GPU, host CPU and Intel MIC) which OpenCL applications can select at runtime.

For desktop machines:
Download a SDK of any available target GPU (NVidia or AMD) and install as instructed.

- N.B. An OpenCL runtime system compiles the kernel code for the system it is being run on, so any OpenCL vendors SDK should allow OpenCL C/C++ source to be compiled. This means you can compile and test code on a laptop with an AMD card and then move your compiled program to a cluster with NVidia hardware and it will also run there. All you would have to do is change the vendor selection string in your OpenCL code and possibly optimise the kernel code to make full use of the hardware it is being run on.

It should be noted that OpenCL kernels written for GPUs will generally have very different performance on CPUs (read that as much worse), the main issue being that CPUs do not handle the high number of threads usually needed to keep GPUs busy.

Programming and compiling using the OpenCL library

C/C++ code has to have the following include:

```c
#include <CL/opencl.h>
```

The OpenCL runtime installed on our bragg-gpu cluster supports multiple platforms (CPU and GPU) and the OpenCL initialisation code in your application should not assume that the first available platform offers the type of device (CPU or GPU) that is required. A more reliable way to initialise your program is to get the list of all available OpenCL platforms and iterate through them until the required type of device is found. This is sample C++ code that does exactly that and should be easy to integrate into your own code:
// helper class for throwing exceptions on errors
class oclError : public runtime_error {
    public:
        oclError (const string &msg = "") : runtime_error (msg) {}
};

// This function returns first device of devType on first
// platform that supports devType type of device
//
// Input:
// devType - CL_DEVICE_TYPE_CPU/CL_DEVICE_TYPE_GPU
//CL_DEVICE_TYPE_ACCELERATOR
// CL_DEVICE_TYPE_DEFAULT or CL_DEVICE_TYPE_ALL
//
// Output:
// platform_id - first platform with devType
// device_id - first device of devType in platform_id
void getPlatformAndDevice (const cl_device_type devType,
    cl_platform_id &platform_id, cl_device_id &device_id) {

    cl_uint nplat, err;
    cl_device_id did;

    // how many platforms are there?
    err = clGetPlatformIDs(0, NULL, &nplat);
    if (err != CL_SUCCESS)
        throw (oclError("OpenCL: no platform found"));

    cout << "Found " << nplat << " OpenCL platforms" << endl;

    // get all platforms ids
    cl_platform_id *pls = new cl_platform_id[nplat];
    err = clGetPlatformIDs(nplat, pls, NULL);
    if (err != CL_SUCCESS)
        throw (oclError("OpenCL: failed to get a list of
platforms"));

    // loop over all platforms until found required type
    for (int i = 0; i < nplat; i++) {
        err = clGetDeviceIDs(pls[i], devType, 1, &did, NULL);
        if (err == CL_SUCCESS) {
            cout << "Found required device type in platform " << nplat << endl;
        
            platform_id = pls[i];
            device_id = did;
            break;
        } else if (err != CL_DEVICE_NOT_FOUND)
            throw (oclError("OpenCL: failed to create a device
group"));
    }

    if (err != CL_SUCCESS) throw (oclError("OpenCL: failed to obtain
requested device"));

    delete[] pls;
The function `getPlatformAndDevice` replaces calls to `clGetPlatformIDs` and `clGetDeviceIDs`. For example if the OpenCL initialisation code in your program looks like that:

```c
cl_platform_id cpPlatform; // OpenCL platform
cl_int devType; // type of compute device
cl_device_id devId; // compute device id
cl_int err;

err = clGetPlatformIDs(1, &cpPlatform, NULL);
if (err != CL_SUCCESS) throw (oclError("OpenCL: platform not found"));

devType = CL_DEVICE_TYPE_GPU;
err = clGetDeviceIDs(cpPlatform, devType, 1, &devId, NULL);
if (err != CL_SUCCESS) throw (oclError("OpenCL: failed to get GPU dev id"));

context = clCreateContext(0, 1, &devId, NULL, NULL, &err);
if (err != CL_SUCCESS) throw (oclError("OpenCL: failed to create context"));
```

it will fail if the first OpenCL platform found is not a GPU.

With `getPlatformAndDevice` the initialisation code will look this way and it will look for the first GPU device on the first platform that supports it:

```c
cl_platform_id cpPlatform; // OpenCL platform
cl_device_id devId; // compute device id
cl_int err;

getPlatformAndDevice (CL_DEVICE_TYPE_GPU, cpPlatform, devId);

context = clCreateContext(0, 1, &devId, NULL, NULL, &err);
if (err != CL_SUCCESS) throw (oclError("OpenCL: failed to create context"));
```

**Linux bragg-gpu and pearcey clusters**

At present, you don't need to load any module to use OpenCL. The include files and libraries are in the standard system directories (`/usr/include` and `/usr/lib64`).

To compile OpenCL code no additional compiler options are required, but to link an executable you will need to specify the OpenCL runtime library

```bash
icc -c oclexample.c
icc -o oclex.exe oclexample.o -lOpenCL
```

or, in one line:

```bash
icc oclexample.c -o oclex.exe -L/usr/lib64 oclexample.o -lOpenCL
```
Instead of the Intel C compiler (icc) you can also use GNU (gcc) or NVIDIA (nvcc).

**Windows machines**

Compilation requires the path to the CL include directory to be set:

```plaintext
set INCLUDE=<PATH_TO_SDK>\include;%INCLUDE%
```

and paths to OpenCL.lib library to be set:

```plaintext
set LIB=<PATH_TO_SDK>\lib\x64;%LIB%
```

OpenCL has two components - run time code and kernel code.

- Run time code exists in the main body of the C/C++ program and controls things such as the selection of the GPU device wanted, compilation of kernel code, memory transfers to and from the GPU and running of the kernels themselves. The run time allows concurrency on a coarse grained level through use of command queues and also on a fine grained level with (massively) parallel kernel code.
- Kernel code is C99 source code (which can be pre-compiled) that defines the compute intensive operations to be carried out on the GPU. The code looks much like CUDA based code (see below) with specific built in functions that allow the programmer to control what thread is operating:

  ```plaintext
  get_global_size()
global_id()
global_size()
local_size()
local_id()
um_groups()
group_id()
  ```

**NVIDIA CUDA**

CUDA ([external site](https://www.nvidia.com)) is an NVidia specific tool chain (set of compiler drivers) that allow modified C/C++ code to specify GPU specific code within source files. CUDA is available only on NVidia cards (such as those in the SC GPU cluster), although you can download it to your PC to compile code in GPU emulation mode.

**Windows**

On your Windows Desktop you can install the CUDA Toolkit from NVidia ([external link](https://developer.nvidia.com/cuda-toolkit)).

Also install the HPC Tool Pack 2008 R2 as mentioned in Job submission above.

Installshield wizard for nvidia gpu Computing SDK and then CUDA 4.0 and the toolkit.

Using nvcc to compile code

```plaintext
nvcc -deviceemu -I..\common\inc sourcefilename.cu
```

NVidia have useful guide to [Getting Started with CUDA](https://developer.nvidia.com/cuda-getting-started).

**Linux**
On SC systems supporting CUDA, applications can be compiled with:

```
module load cuda
nvcc foo.cu
```

Some applications may attempt to link directly to libcuda.so while others use libcudart.so.

The runtime library is found through the cuda module but the former is only installed on machines with the CUDA drivers packages.

This is not the case for the bragg cluster login node which has no attached GPUs. In those instances you should build your program either on bragg-l-test.csiro.au or on compute node (using a batch script or an interactive batch session).