Australian National University

The OpenCL C++ API

Host programs can be verbose

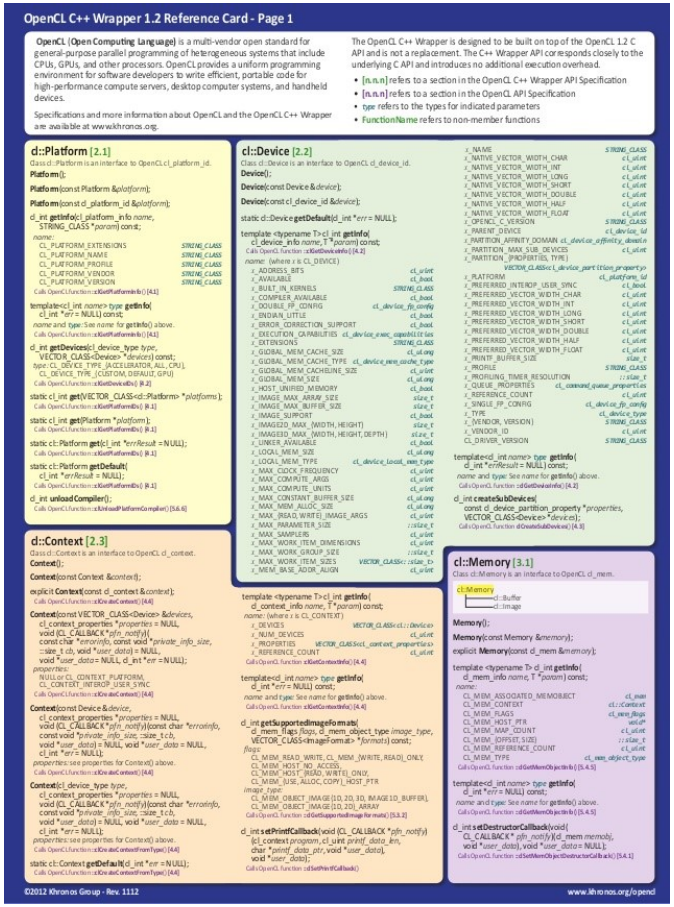
- OpenCL's goal is extreme portability, so it exposes ***everything***
- Most of the host code is the same from one application to the next – this **re-use** makes the verbosity a non-issue.
- Common API combinations can be packaged into functions, C++ or Python classes, or libraries to make the reuse more convenient.

The C++ Interface

- Khronos has defined a common C++ header file containing a high level interface to OpenCL, **cl2.hpp**
- This interface is dramatically easier to work with[1]
- **Key OpenCL C++ API features:**
  - Uses common defaults for the platform and command-queue, saving the programmer from extra coding for the most common use cases
  - Simplifies the basic API by bundling key parameters with the objects rather than requiring verbose and repetitive argument lists
  - Ability to "call" a kernel from the host, like a regular function
  - Error checking can be performed with C++ exceptions

[1] especially for C++ programmers…

The OpenCL C++ API ref card



- Useful to have the OpenCL C++ reference card to hand
- Download it from the Khronos website:
- http://www.khronos.org/files/OpenCLPP12-reference-card.pdf
- Doxygen available here: http://github.khronos.org/OpenCL-CLHPP/

1. Create a context and queue

- Grab a context using a device type:

```
cl::Context
  context(CL_DEVICE_TYPE_DEFAULT);
```

- Create a command queue for the first device in the context:

```
cl::CommandQueue queue(context);
```

2. Create and Build the program

- Define source code for the kernel-program either as a string literal (great for toy programs) or read it from a file (for real applications).

- Create the **program object and compile** to create a "dynamic library" from which specific kernels can be pulled:

"true" tells OpenCL to build (compile/link) the program object

```
cl::Program program(context, KernelSource, true);
```

KernelSource is a string … either statically set in the host program or returned from a function that loads the kernel code from a file.

Compiler error messages

- For most real programs we will want to catch and report kernel compilation errors. To do this, we add an additional catch clause:

```
catch (cl::BuildError error)
{
   // Recover compiler messages for first device
   // .first is the device, .second is the log
   std::string log = error.getBuildLog()[0].second;
   std::cerr << "Build failed:" << std::endl
             << log
             << std::endl;
}
catch (cl::Error error)
{
  ...
}
```

3. Setup Memory Objects

- For vector addition we need 3 memory objects, one each for input vectors A and B, and one for the output vector C

- Create input vectors and assign values **on the host**:

```
std::vector<float> h_a(N), h_b(N), h_c(N);
for (i = 0; i < N; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
}
```

- Define **OpenCL device buffers** and copy from **host buffers**:

```
cl::Buffer d_a(context, h_a.begin(), h_a.end(), true);
cl::Buffer d_b(context, h_b.begin(), h_b.end(), true);
cl::Buffer d_c(context, CL_MEM_WRITE_ONLY,
                sizeof(float)*N);
```

Creating and manipulating buffers

- Buffers are declared on the host as object type:
  `cl::Buffer`

- Arrays in host memory hold your original host-side data:

```
std::vector<float> h_a, h_b;
```

- Create the device-side **buffer** (d_a), assign read only memory to hold the host array (h_a) and copy it into device memory:

```
cl::Buffer d_a(context, h_a.begin(), h_a.end(), true);
```

Start_iterator and end_iterator for the container-holding host-side object

Stipulates that this is a read-only buffer

Creating and manipulating buffers

- The last argument sets the device's read/write access to the Buffer. **true** means “read only” while **false** (the default) means “read/write”.

- Can use explicit copy commands to copy from the device buffer (in global memory) to host memory:

```
cl::copy(queue, d_c, h_c.begin(), h_c.end());
```

- Can also copy from host memory to global memory:

```
cl::copy(queue, h_c.begin(), h_c.end(), d_c);
```

4. Define the kernel

- Create a ***kernel functor*** for the kernels you want to be able to call in the **program**:

```
cl::KernelFunctor
        <cl::Buffer, cl::Buffer, cl::Buffer>
        vadd(program, "vadd");
```

Must match the pattern of arguments to the kernel.

Variable name

A previously created "program object" serving as a dynamic library of kernels

The name of the function used for the kernel

- This means you can 'call' the kernel as a 'function' in your host code to enqueue the kernel.

Create a kernel (advanced)

- If you want to query information about a kernel, you will need a **kernel object** too:

```
cl::Kernel ko_vadd(program, "vadd");
or
cl::Kernel ko_vadd
  = vadd.getKernel();
```

If we set the local dimension ourselves or accept the OpenCL runtime's, we don't need this step

- Get the maximum size for a work-group:

```
::size_t local = ko_vadd.getWorkGroupInfo
 <CL_KERNEL_WORK_GROUP_SIZE>(Device::getDefault());
```

We can use any work-group-info parameter from table 5.15 in the OpenCL 1.1 specification. The function will return the appropriate type.

5. Enqueue commands

- Specify *global* (and optionally *local)* dimensions
  - cl::NDRange global(1024)
  - cl::NDRange local(64)
  - If you don't specify a local dimension, it is assumed as cl::NullRange, and the runtime picks a size for you

- Enqueue the kernel for execution (note: non-blocking):

```
vadd(cl::EnqueueArgs(queue, global), d_a, d_b, d_c);
```

- Read back result (as a blocking operation). We use an in-order queue to ensure the previous commands are completed before the read can begin

```
cl::copy(queue, d_c, h_c.begin(), h_c.end());
```

C++ Interface: setting up the host program

- Enable OpenCL API Exceptions. Do this before including the header file

```
#define CL_HPP_ENABLE_EXCEPTIONS
```

- Specify the version of OpenCL that you wish to target. Do this before including the header file

```
#define CL_HPP_TARGET_OPENCL_VERSION 120
#define CL_HPP_MINIMUM_OPENCL_VERSION 120
```

- Include key header files … both standard and custom

```
#include <CL/cl2.hpp>   // Khronos C++ Wrapper API
#include <cstdio>       // For C style IO
#include <iostream>     // For C++ style IO
#include <vector>       // For C++ vector types
```

C++ interface: The vadd host program

```
#define N 1024

using namespace cl;

int main(void) {

vector<float> h_a(N), h_b(N), h_c(N);
// initialize these host vectors…

Buffer d_a, d_b, d_c;

Context context(CL_DEVICE_TYPE_DEFAULT);

CommandQueue queue(context);

Program program(context,
            loadprogram("vadd.cl"), true);

// Create the kernel functor
KernelFunctor<Buffer, Buffer, Buffer>
   vadd(program, "vadd");

// Create buffers
// True indicates CL_MEM_READ_ONLY
// False indicates CL_MEM_READ_WRITE
d_a = Buffer(context, h_a.begin(), h_a.end(), true);
d_b = Buffer(context, h_b.begin(), h_b.end(), true);
d_c = Buffer(context, CL_MEM_WRITE_ONLY,
                sizeof(float) * N);

// Enqueue the kernel
vadd(EnqueueArgs(queue, NDRange(N)),
     d_a, d_b, d_c);

copy(queue, d_c, h_c.begin(), h_c.end());

}
```

Note: The default context and command queue are used when we do not specify one in the function calls. The code here also uses the default device, so these cases are the same.

The C++ Buffer Constructor

- The API definition:
  - Buffer(startIterator, endIterator, bool readOnly, bool useHostPtr)
- The readOnly boolean specifies whether the memory is CL_MEM_READ_ONLY (true) or CL_MEM_READ_WRITE (false)
  - You must specify a true or false here
- The useHostPtr boolean is false by default
  - Therefore the array defined by the iterators is implicitly copied into device memory
  - If you specify true:
    - The memory specified by the iterators must be contiguous
    - The context uses the pointer to the host memory, which becomes device accessible - this is the same as CL_MEM_USE_HOST_PTR
    - The array **might not be** copied to device memory
- We can also specify a context to use as the first argument in this API call

The C++ Buffer Constructor

- When using the buffer constructor which uses C++ vector iterators, remember:
  - This is a blocking call
  - The constructor will enqueue a copy to the first Device in the context (when useHostPtr == false)
  - The OpenCL runtime will automatically ensure the buffer is copied across to the actual device you enqueue a kernel on later if you enqueue the kernel on a different device within this context

The Python Interface

- A Python library by Andreas Klöckner from University of Illinois at Urbana-Champaign
- This interface is dramatically easier to work with[1]
- Key features:
  - Helper functions to choose platform/device at runtime
  - getInfo() methods are class attributes – no need to call the method itself
  - Call a kernel as a method
  - Multi-line strings – no need to escape new lines!

[1] not just for Python programmers…

Setting up the host program

- Import the pyopencl library

```
import pyopencl as cl
```

- Import numpy to use arrays etc.

```
import numpy
```

- Some of the examples use a helper library to print out some information

```
import deviceinfo
```

```
N = 1024
# create context, queue and program
context = cl.create_some_context()
queue   = cl.CommandQueue(context)
kernelsource = open('vadd.cl').read()
program = cl.Program(context, kernelsource).build()

# create host arrays
h_a = numpy.random.rand(N).astype(float32)
h_b = numpy.random.rand(N).astype(float32)
h_c = numpy.empty(N).astype(float32)

# create device buffers
mf  = cl.mem_flags
d_a = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_a)
d_b = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_b)
d_c = cl.Buffer(context, mf.WRITE_ONLY, h_c.nbytes)

# run kernel
program.vadd(queue, h_a.shape, None, d_a, d_b, d_c)

# return results
cl.enqueue_copy(queue, h_c, d_c)
```

We have now covered the basic platform runtime APIs in OpenCL