codeplay®

OpenCL as a back-end for compiler tools

(and C++ on GPUs)

Andrew Richards

CEO, Codeplay

Visit us at
www.codeplay.com

2nd Floor
45 York Place
Edinburgh
EH1 3HP
United Kingdom

Why?

- Performance: GPUs (& FPGAs) are fast!
- OpenCL lets you run on a wide variety of parallel devices with high performance
  - GPU/CPU/FPGA/Cell BE, etc...
  - Open standard, widely supported, lots going on
- A higher-level language may have domain-specific knowledge about software to enable parallelization
  - Can make GPU acceleration easy ?

What are the problems?

- A GPU (or FGPA) is not a 100% general-purpose programmable device
  - May change in the future, but for now:
    - Data-parallel, with limited task-parallel support
    - No recursion, no globals, no function pointers
    - Only access data in buffers
    - Limits on types available (double may be, bytes might be limited etc)
- Shipping source for unknown platforms
  - Need to be able to handle run-time compilation

A simple example

```
void offloadCLExample (int width, int height, float *myFloatArray )
{
    GpuBuffer<float, 2> myGpuBuffer (width, height, myFloatArray);

    myGpuBuffer.unmap (); // move the data from host mem onto device

    parallel_for (width, height,
        processBuffer (myFloatArray) // call processBuffer in parallel
    );

    myGpuBuffer.map (); // now myFloatArray is accessible again
}
```

- This is what it looks like in our C++ - to OpenCL tool
- Your language may look different, but this is a good reference

A simple example

```
void offloadCLExample (int width, int height, float *myFloatArray )
{
    GpuBuffer<float, 2> myGpuBuffer (width, height, myFloatArray);

    myGpuBuffer.unmap (); // move the data from host mem onto device

    parallel_for (width, height,
        processBuffer (myFloatArray) // call processBuffer in parallel
    );

    myGpuBuffer.map (); // now myFloatArray is accessible again
}
```

- We are going to process an array of data
  - Has a *width*, a *height*, and we get a pointer to the data, *myFloatArray*
- We need to *unmap* it from host (CPU) onto the device (GPU), then process it in parallel on the device, then *map* it back to host

A simple example – the buffer

```
void offloadCLExample (int width, int height, float *myFloatArray )
{
    GpuBuffer<float, 2> myGpuBuffer (width, height, myFloatArray);

    myGpuBuffer.unmap (); // move the data from host mem onto device

    parallel_for (width, height,
        processBuffer (myFloatArray) // call processBuffer in parallel
    );

    myGpuBuffer.map (); // now myFloatArray is accessible again
}
```

- We wrap the buffer in a C++ class (‘`GpuBuffer`’)
  - (but you could create a special array type for your language)
- You have to explicitly map and unmap the buffer
- You may decide to automatically handle map & unmap, but that may limit optimization opportunities for programmer

A simple example – parallel-for

```
void offloadCLExample (int width, int height, float *myFloatArray )
{
    GpuBuffer<float, 2> myGpuBuffer (width, height, myFloatArray);

    myGpuBuffer.unmap (); // move the data from host mem onto device

    parallel_for (width, height,
        processBuffer (myFloatArray) // call processBuffer in parallel
    );

    myGpuBuffer.map (); // now myFloatArray is accessible again
}
```

- The kernel needs to be called as a "parallel-for" operation
  - In C++0x, we can use a lambda function, but I use a *functor* here
- The parallel-for operation has a 2D range in this example
- We will call 'processBuffer' across the *width* × *height* range

A simple example – the kernel

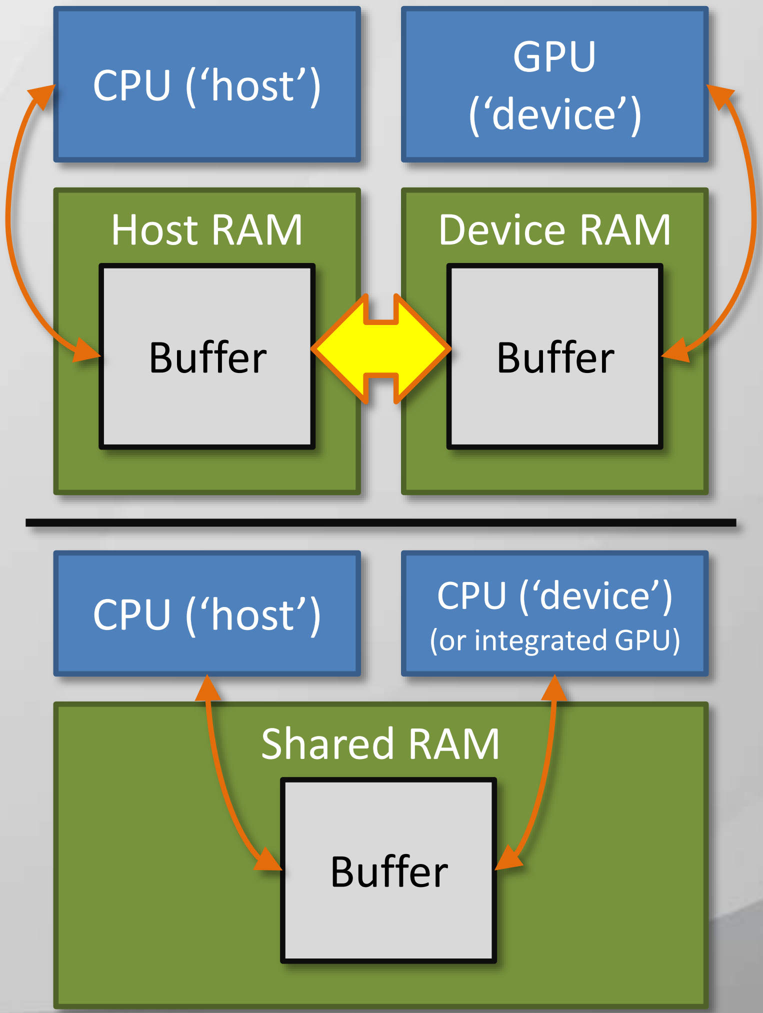
```
class processBufferKernel
{
    GpuKernelBuffer<float, 2> myData;

public:
    void operator() (int i, int j)
    { ... work goes here
    }
}
```

- This is how to write a kernel function as a *functor* in C++
  - In C++0x, we can use lambda functions, which makes it easier
- Our kernel is a function that receives:
  - The iterators as integer parameters: indices into 0..width & 0..height
  - The original parameters to the kernel
- Notice how we use 'GpuKernelBuffer' here, and 'GpuBuffer' in the host: the big issue is how to handle these parameters

Handling buffer types

- In OpenCL, a ‘buffer’ is a way of handling the fact that data shared between host and device might be:
  - In a different memory chip
  - At a different address
  - Copied or shared
  - Use different pointer sizes (32-bit vs 64-bit, etc.)
- On the host side, we use a ‘buffer’ object
- On the device side, we use a ‘global’ or ‘constant’ pointer

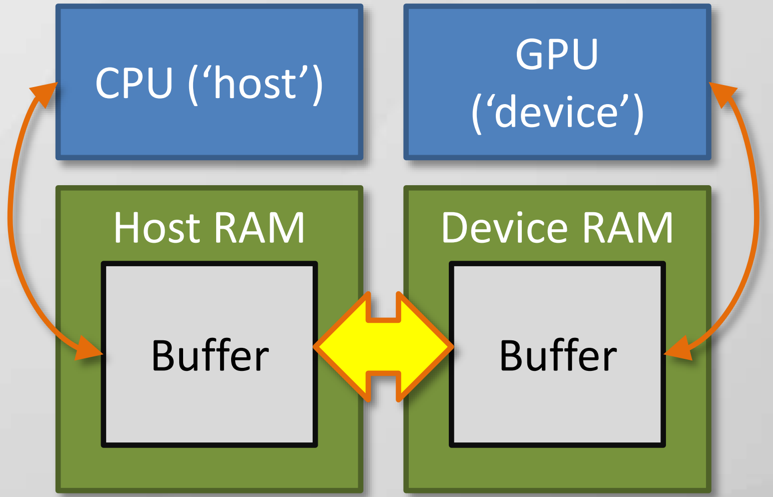


Pointers and buffers

- Ideally, we would share pointers between host and device. That would make life simple, but:
  - It would mean the hardware would have to use the same pointer sizes (tricky on mixed 32-bit/64-bit operating systems)
  - We would have to share the page tables between host and device (tricky on parallel architectures)
  - The device would have to be able to handle seg-faults and interrupt the OS to request pages

Buffer and pointer types

- On the host side, a buffer
  - Has a cl_mem object
  - Has a pointer to the host data
  - Has a size
  - Has some state: mapped/unmapped/in-use/in-transit etc.
- On the Kernel side, a buffer
  - Is a pointer to global or constant memory
  - Has a size



```
cl_mem data;
T *hostPtr;
int elements;
BufferEnum state;
```

```
global T *data;
int elements;
```

```
constant T *data;
int elements;
```

Notice how we have multiple types: 1 type on the host, another 2 on the device. In our higher-level language, we need to encapsulate this

Passing data to a kernel

- The transition from *buffer* to `global` or `constant` pointer is *only* handled by `clSetKernelArg`
  - Cannot do the transition anywhere else. Can't be called in a kernel, can't be called before-hand
  - So, need to create a parallel function call that passes in some buffers, but the function receives `global` and `local` pointers
  - Any data accessed by a kernel must go through this process. This has impact on data-structures

Performance considerations

- Kernels are compiled at run-time, so try to create a list of all of them and compile at the start
- Need to expose buffers in a way that is easy to use, but hard to abuse
- May need to expose map/unmap in a way that allows programmers to minimize their use and keep data on-device where possible
- Probably need to expose `__local` memory as memory local to a small group of threads

Summary

- Handling limitations in features of GPU is mostly easy: some kernels work on a device, others don't
- Handling memory is much harder
  - Especially the subtleties due to the use of buffers and the impact on language design
- Need to be able to expose some performance details