OpenCL C++

Benedict R. Gaster
Advanced Micro Devices, 1 AMD, Sunnyvale, CA, USA
benedict.gaster@amd.com

Lee Howes
Advanced Micro Devices, 1 AMD, Sunnyvale, CA, USA
lee.howes@amd.com

Abstract

With the success of programming models such as Khronos' OpenCL, heterogeneous computing is going mainstream. However, these models are low-level, even when considering them as systems programming models. For example, OpenCL is effectively an extended subset of C99, limited to the type unsafe procedural abstraction that C has provided for more than 30 years. Computer systems programming has for more than two decades been able to do a lot better. One successful case in point is the systems programming language C++, known for its strong(er) type system, templates, and object-oriented abstraction features.

In this paper we introduce OpenCL C++, an object-oriented programming model (based on C++11) for heterogeneous computing and an alternative for developers targeting OpenCL enabled devices. We show that OpenCL C's address space qualifiers, and by implication Embedded C's, can be lifted into C++'s type system. A novel application of C++11's new type inference features (auto/decltype) with respect to address space qualifiers allows natural and generic use of the *this* pointer. We qualitatively show that OpenCL C++ is a simpler and a more expressive development platform than its OpenCL C counter part.

1. Introduction

... justification for Church's calculus, and allows the 'machines' which generate computable functions to be replaced by the more convenient λ-definitions.

Alan Turing

At the beginning of modern computer science, Turing had already noted that notation and "simpler" abstractions would play a major role in defining successful programming models. In this paper we address the syntactic and semantic limitations of current GPGPU programming models, without introducing yet another entirely new compiler and runtime but rather incrementally extending the current capabilities of OpenCL.

Following the single-core and multi-core revolutions there is now a new emerging era of computing: heterogeneous systems combining multi-core CPUs, many-core GPUs, and other accelerator devices. These are no longer just oddities to discuss in architecture classes, they are the future! Many existing applications can be naturally decomposed so that they can be executed in parallel on such systems.

There has been early success in addressing the issue of programming models for heterogeneous computing, c.f. Khronos' OpenCL [16] and NVIDIA's CUDA [15]. However, these systems are low-level, even when considering them as systems programming models. They are effectively extended subsets of C99, limited to the type unsafe procedural abstraction that C provides. Cuda has addressed these limitations somewhat but in an ad-hoc and proprietary setting, allowing them to avoid the portability constraints that OpenCL must cater for.

OpenCL was defined with portability in mind and the ability to support many different host and device platforms. While OpenCL can provide excellent performance, programming in OpenCL is long winded, at best, and down right difficult at worse. For example, to simply enqueue a kernel to a device the programmer must: select a platform; select a device and create a context; allocate memory objects; copy data to device; create and compile programs; create a kernel; create a command queue; enqueue the kernel for execution; and finally copy data back from device.

Figure 1 shows the OpenCL API calls required to implement vector addition, the "Hello World" of data-parallel computing. 30 lines of OpenCL C API code just to execute a single kernel on two input buffers and copy the result back to host memory! Compare this with the 7 lines of OpenCL C++ API code, as shown in Figure 2. It is worth noting that the original OpenCL C code does not contain any error handling code, all this would need to be added, while the OpenCL C++ implementation gets error handling for "free" by using exceptions as well as basic type safety and requires no additional modifications. The OpenCL C++ device code gains from static safety features of C++, while not yet supporting dynamic features such as exceptions due to hardware limitations.

Three novel aspects of OpenCL C++ are:

- A type inference system for infering address space usage, allowing developers to mix explicit and implicit use of address space qualifers.
- The ability to share pointer based data-structures between the host and device. Thus enabling developers to target systems with and without a single virtual memory system between the host and device, from a single source base.
- The exploration of enhancing OpenCL C++'s pointers through the decoupled Access/Execute (Æcute—pronounced "acute") programming model, which allows the programmer to express explicitly both the memory access pattern and the execution schedule of a computation kernel [6]. The OpenCL C++ programmer can annotate pointers, describing their access pattern.

Designing OpenCL C++ we focused on achieving performance and productivity for heterogeneous programming development. In particular we tried to balance the following two goals:

GPGPU-6 March 16, 2013, Houston, TX, USA
Copyright © 2013 ACM 978-1-4503-2017-7/13/03. . . $15.00

```
clGetPlatformIDs(1, &platform, NULL);

cl_context = clCreateContextFromType(platform,
   CL_DEVICE_TYPE_DEFAULT, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices,
   NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
   sizeof(cl_float)*n, NULL, NULL);
// create the program
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
                                           NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                  sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
   CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

Figure 1. OpenCL C Host API Code for Vector Add

```
std::function<Event (const EnqueueArgs&, Buffer, Buffer, Buffer)> vadd =
            make_kernel<Buffer, Buffer, Buffer>(Program(program_source), "vadd");


memobj[0]  = Buffer(CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  sizeof(float) * n, srcA);
memobj[1]  = Buffer(CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  sizeof(float) * n, srcB);
memobj[2]  = Buffer(CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n);

vadd(EnqueueArgs(NDRange(n)), memobj[0], memobj[1], memobj[2]);

enqueueReadBuffer(memobj[2], CL_TRUE, 0, sizeof(float) * n,   dest);
```

Figure 2. OpenCL C++ Host API Code for Vector Add

Safety. To aid productivity OpenCL C++ is intended to be safe. Due to performance requirements and choosing C++ as the base language the set of guarantees are limited compared to some parallel languages, e.g. X10 [3]. In particular, OpenCL C++ does not address illegal pointer references, including NULL pointers, and buffer overflows. However, when possible, types are used to provide static guarantees and initialization errors are ruled out.

Scalability. OpenCL C++ is designed to support development of scalable applications, i.e. the addition of computational resources should lead to an increase in performance. As OpenCL C++ is designed for heterogeneous style architectures, additional computational resources include CPU, GPUs, and other compute devices.

Earlier work on OpenCL C++'s implementation has provided a "real" product, one that is already widely used. Many of the advances described in this paper, e.g. C++ kernel language, defaults, and shared pointers, have recently been released as part this product. Still some other aspects are still in the research and development cycle, in particular our notion of separating data-access from execution is not yet available in production form.

The remaining sections of this paper are; Section 2 introduces the main features of OpenCL C++ programming model; Section 3 expands on the Æcute programming model for pointer types; Section 4 describes the OpenCL C++ Kernel language; Section 5 gives an overview of OpenCL C++'s implementation; Section 6 describes the impact on performance compared to the OpenCL C API and a measure of code complexity; and finally Section 7 concludes.

A note to the reader; this paper describes, among other things, syntactic constructs for extending OpenCL with C++ and address spaces. A companion paper [5], also at this conference, describes a formal foundation for address space usage and a system for address space inference. It is the intention that these two papers be read together.

2. OpenCL C++ Programming Model

OpenCL C++ is defined as three parts, called models, that can be summarized as follows:

- Platform model: Specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL C++ kernels (the devices). OpenCL C++ defines an abstract hardware model, described below, that is used by programmers when writing kernels and functions that execute on devices.
- Execution model: Defines how the OpenCL C++ environment is configured on the host and how kernels are executed on the device. This includes device capabilities and a concurrency model used for kernel execution on devices. The execution model is outlined in Section 2.2.
- Memory model: Defines an abstract memory hierarchy and set of visibility guarantees that kernels may rely on, regardless of the actual underlying memory architecture. OpenCL C++ keeps the current OpenCL memory model, but adapts it to allow implicit use of address spaces with respect to the *this* pointer. In practice, we believe that the OpenCL memory model is incomplete and should be generalized to adopt C++11's memory model [9]. We have made progress on this front but leave a detailed description for future work.

2.1 Platform Model

In the platform model, there is a single host that coordinates execution on one or more devices. Platforms can be thought of as vendor-specific implementations of the OpenCL C++ API. The devices that a platform can target are thus limited to those with which a particular implementation knows how to interact with. For example, if implementation A's device(s) is chosen, it cannot communicate with implementation B's device(s). This differs from the OpenCL model in that no explicit platform, or context for that matter, are required thus simplifying a common OpenCL programming idiom. The more relaxed usage is handled by the introduction of defaults, described below. It is still possible for the programmer to explicitly manage the platforms, contexts, devices, and command queues, but

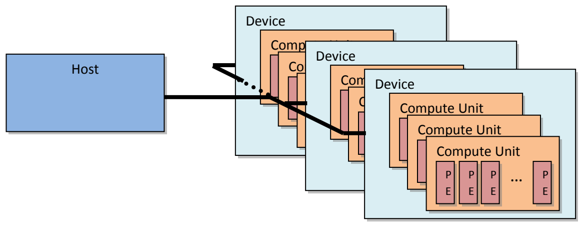

Figure 3. OpenCL C++'s Platform Model

it is not required. It was deemed vital to maintain access to this low level functionality in an API targeted at high performance programming.

The platform model presents an abstract device architecture that programmers target when writing kernels that OpenCL C++ will execute. An implementation maps this abstract architecture to the physical hardware.

The OpenCL C++ platform model defines a device as an array of compute units, with each compute unit functionally independent from the rest. Compute units are further divided into processing elements. Figure 3 illustrates this hierarchical model. It is important to note that for OpenCL C++'s pointers this model is, by default, implicit and managed by the OpenCL C++ runtime. For compatibility with OpenCL, conventional buffers are still managed explicitly.

2.2 Execution Model

As described above the OpenCL C++ platform model defines the roles of the host and devices and provides an abstract hardware model for devices.

2.2.1 Host-Device Interaction

Kernels intended for execution on the device are marked with `kernel`, appearing before the return type, which like OpenCL must be void. Kernels are written in a super set of C++11 and are described in detail in Section 4.

A simple OpenCL C++ kernel, that doubles its input, might be defined as:

```
kernel void addSelf(global Pointer<int> input)
{
  *(input+get_global_id(0)) *= 2;
}
```

OpenCL C++ kernels are compiled as per the OpenCL online compilation model, although it is also possible to support a type-safe offline model too.

2.2.2 Work-items

Like OpenCL, OpenCL C++ device programs assume an explicitly parallel model where a kernel describes the execution of a single lane of execution called a *work-item*. When a kernel starts execution, many work-items may start up, each running the kernel. (A kernel is invoked with a grid-based enqueue operation, described later).

Work-items are executed concurrently, although there is no guarantee that they will execute in parallel and no requirement for preemption.

For each dimension *i*, the set of ID_i are drawn from the set [0, 1, 2, ..., $maxi_i - 1$]. The flattened *ID* of a work-item is defined by:

```
ID = ID0 + ID1 * max0 + ID2 * max0 * max1
```

where:

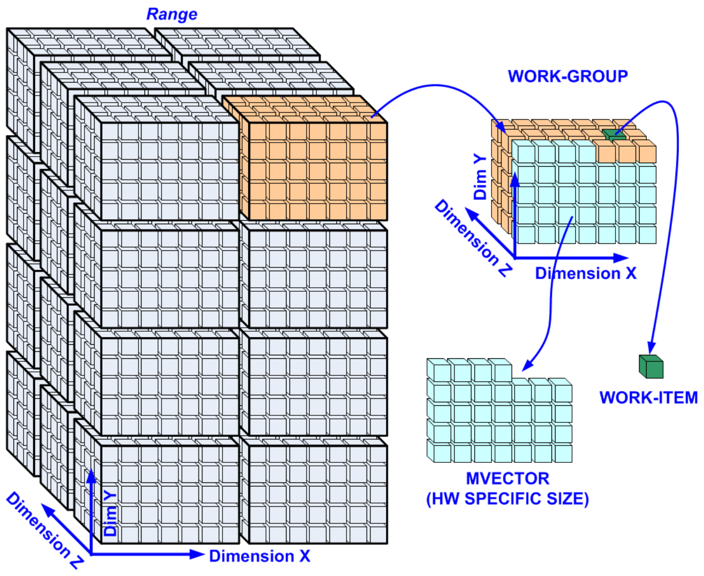

Figure 4. Range, Work-groups, and Work-items in OpenCL C++

```
- ID0 is work-item in dimension 0
- ID1 is work-item in dimension 1
- ID2 is work-item in dimension 2
- max0 is the extent of the execution GRID
  in dimension 0
- max1 is the extent of the execution GRID
  in dimension 1
```

Work-items start execution of work-items in flattened ID order.

2.2.3 Work-Groups

Work-items can be organized into work-groups, of size 1 or greater. Using the notion of a distributed array, discussed later, work-items within a work-group can communicate and share data.

Every work-group has a multi-dimensional index called a work-group index. The work-group index is calculated by dividing each component of the work-item's absolute ID by the corresponding size of the work-group.

Each work-group has a flattened ID (similar to that of a work-item):

```
work-group-id = floor(work-item-id / work-group-size)
```

where:

```
work-group-size = work-group-size-dim0 *
                  work-group-size-dim1 *
                  work-group-size-dim2
```

Each work-item has a unique identifier called the work-item ID within the work-group.

2.2.4 Machine Vectors

Collections of work-items within a work-group are executed in lock-step as part of a vector, called an *mvector*. The specific length of an *mvector* is implementation defined and is exposed in OpenCL C++ only as a symbolic constant (*MVECTOR_SIZE*). Work-items are gang-scheduled in groups (of size *MVECTOR_SIZE*) and it is assumed that these are SIMD executed when convergent, potentially using predication.

The mvector is a construct that exposes a guarantee the compiler and runtime are making to the developer. Work-items that are part of a single mvector are guaranteed to execute synchronously within the bounds of well-defined reconvergence rules (essentially code must reconverge no later than the immediate post-dominator of the divergence). These guarantees ensure that the developer is able to tell when code will be synchronously executing and, more importantly, where sets of work items are guaranteed to make independent forward progress. He is able to develop more flexible algorithms with this information available, and functions such as the width enhanced barrier may then be derived from the base feature.

Work-items are assigned to *mvectors* in work-item flattened absolute ID order. Two work-items within the same work-group will be in the same *mvector* if the floor of (work-item flattened absolute ID / MVECTOR_SIZE) is the same. *work-items* are assigned to *mvectors* in the same order for any work-group. For many implementations this may mean assigning a work-group to a single compute-unit, as is the case in OpenCL, but OpenCL C++ requires only the behavior and an implementation is free to execute a work-group however it chooses as long as it preserves the visible invariants.

The assignment of work-items to *mvectors* is sequential, i.e. they are sequentially numbered. As the maximum number of dimensions of a work-group is 3, X by Y by Z, work-items are packed into *mvectors* in consecutive order in X, then Y, then Z.

As an example, consider the case when the work-group size is $16 \times 4 \times 10$ and the *MVECTOR_SIZE* is 64. Such a work-group has 640 work-items. Dividing this number by 64 (i.e. *MVECTOR_SIZE* in this case) gives 10 *mvectors* per work-group [1]

The mapping follows the pattern:

```
mvector 0: Z = 0, Y = 0, X = 0
mvector 1: Z = 0, Y = 0, X = 1
...
mvector 15:  Z = 0, Y = 0, X = 15

mvector 16: Z = 0, Y = 1, X = 0
...
mvector 31: Z = 0, Y = 1, X = 15
mvector 32: Z = 0, Y = 2, X = 0
...
```

As OpenCL C++ does not specify a single size for an *mvector* it is the developers responsibility to not encode assumptions about a specific implementation size, as the behavior may be undefined on another implementation. This can lead to an application generating incorrect answers and in some cases deadlock.

Standard OpenCL does not expose the machine vector, even though almost all known hardware implementations of the architecture provide some notion of vector, e.g. NVIDIA's WARP or AMD's WAVEFRONT. The advantage of exposing it is the possibility of adding explicit vector operations, such as shuffle.

One interesting operation we have been exploring in this regard is width defined barrier operations. For example, the following OpenCL C++ builtin allows the programmer to capture the set of work-items that are participating in a work-group barrier operation:

```
void barrier(
  const cl_mem_fence flags, const size_t num);
```

This is an extension to the OpenCL C barrier operation with an additional argument that defines how many work-items will take part in the barrier operation. This argument is required to be known statically, to be a power of 2 and describes a consecutive set of work-items. These limitations are required to allow the compiler to narrow down exactly which work items are taking part. With this information the compiler can often determine if communication is within a *mvector* and perform barrier elision. There are numerous examples, c.f. CUDA work on RadixSort [11, 12], where barriers are elided by hand making the code non-portable across different devices, our width defined barriers introduce a portable way to achieve the same functionality and performance. We are currently working on generalizing this idea to width defined functions, that the barrier operation would be one instance of. We leave the details of this to future work.

2.2.5 NDRanges

Work-groups in turn are grouped into a larger structure called a *NDRange*. When a kernel is launched an ndrange is formed that describes the set of work-groups to be executed. As execution proceeds work-groups are distributed onto an OpenCL C++ device. Note, it is not required that separate work-groups be executed on the same device, other than respecting visibility order as defined by the memory model.

The maximum value of the flattened ID (and therefore the maximum size of a range) is $2^{32} - 1$.

The relationship between, work-item, *mvectors*, work-groups, and ranges is shown diagrammatically in Figure 4.

2.3 Memory Model

The OpenCL C++ memory layout is similar to that of a traditional GPGPU segmented memory model, i.e. there are host and device memories, the later of which is divided into separate non-coherent memories.

It is clear that going forward, that many architectures will provide a single unified heap between host and device, c.f. AMD's Fusion APUs [17] or Intel's Sandy Bridge [8], opening up not only the ability to reduce the cost of data-transfers but also opening the way for pointer based data-structures and algorithms shared between devices. In designing OpenCL C++ we believed it was important to introduce the ability to develop data-structures using pointers, on both host and device. This goes beyond any existing GPGPU models, including OpenCL and Cuda.

In the design and introduction of OpenCL C++'s pointer type it was critical to make it work on existing hardware, via custom heap allocators and implicit data-movement between host and device, but also to introduce no additional overhead when mapped to architectures that share a single virtual address space.

With some thought it became clear that this could actually be achieved on today's architectures, while maintaining the prospect of performance benefits from future unified address space architectures. To this end OpenCL C++ introduces both a class that behaves just like a C++ pointer and the notion of heap allocators. The following is an example of how OpenCL C++ pointer types might be used in practice:

```
cl::Pointer<int> x = cl::malloc<int>(N);

for (int i = 0; i < N; i++)  {
  *(x+i) = rand();
}

std::function<
  Event (const cl::EnqueueArgs&,
                 cl::Pointer<int>)> plus =
  make_kernel<cl::Pointer<int>, int>(
  "kernel void plus(global Pointer<int> io)"
   {
     int i = get_global_id(0);
     *(io+i) = *(io+i) * 2;
```

[1] It is possible that some implementations might support work-group sizes that are not multiples of *MVECTOR_SIZE*, at this time OpenCL C++ makes this restriction.

```
  }");

plus(EnqueueArgs(NDRange(N)), x);

for (int i = 0; i < N; i++) {
  cout << *(x+i) << endl;
}
```

The key point to note is that a single shared pointer type, `Pointer<int>`, is used across both host and device. Furthermore, unlike OpenCL and Cuda there are no explicit data-transfers required to move the data between the difference address spaces. In this example pointers are used to store a traditional OpenCL buffer, with implicit data-movement, but in general OpenCL C++ pointers can be read and written to build complex pointer based data-structures. For example, the following code shows how to define a linked list and a function to allocate nodes:

```
struct Node
{
  int value;
  Pointer<Node> next;
};

Pointer<Node> createNode(int x)
{
  Pointer<Node> result = malloc<Node>(1);
  result->value = x;
  result->next = Pointer<Node>();
  return result;
}
```

OpenCL C++'s pointers are based on the notion of Heap Layers [2]. Heap Layers is a mixin based approach to memory allocation. Heap Layers are highly customizable, allowing different allocators for specific types of data as well as specialized allocation sizes. In general, an OpenCL C++ pointer is defined as the template class:

```
template <
  typename T,
  template<class Align> class Heap = SharedHeap,
  typename AlignmentType = size_t >
class Pointer;
```

where

- T is the type of value pointed too;
- Heap is the actual heap allocator, for type T, which itself is a class that must meet a specific interface, described below.
- AlignmentType is used to calculate the alignment of allocations.

The argument of particular interest is `Heap`. This represents the allocator for values of type `T`. There is no requirement that there be a single allocator for a given type, for example, an implementation might provide an alternative implementation for small allocation sizes. The interface for allocators must be implemented by all instances used in `Pointer`:

```
template<typename AlignmentType_>
class Allocator {
public:
  template<typename T>
  size_t malloc(size_t num);

  template<typename T>
  void free(size_t offset);

  Event sync(
    Affinity placement = Anywhere);
};
```

Any allocator providing malloc and free implementations can be used to build custom new/delete implementations in OpenCL C++ enabled applications. Unlike in OpenCL, data movement is, by default, implicit between the host and the devices in the system. However, to allow finer grained control the allocator interface provides a `sync` member function. This takes an affinity (hint) and returns an event, that can be used to control execution synchronously (i.e. wait for memory to be transferred) or asynchronously to perform this action ahead of time.

As specific allocators can be used for all allocations, even of the same type, this allows for very fine grain control of data-movement, while providing a simple implicit move semantics for the default case or when data movement of a particularly region is not on the crucial path.

A default heap is required to be provided by an implementation for all values of `T_`. This can be referenced in the OpenCL C++ namespace as `cl::SharedHeap`, and is assumed to have an alignment type that is set to `size_t`.

2.4 OpenCL C++ Objects

For each base object in the original OpenCL C API, e.g. `cl_device_id`, a corresponding OpenCL C++ class is provided. Unlike the C API the C++ one can provide multiple ways of constructing an OpenCL object, changing just the parameters when required. A requirement of OpenCL C++'s design was to allow close interaction with the underlying C API: it is thus always possible to get to the corresponding C object by use of the `operator()`.

Additionally any C runtime library must either implicitly manage memory or have the user manage it explicitly. OpenCL object lifetime is explicitly controlled by the programmer with routines to increment and decrement the reference count for each entity. It is well known that this can lead to program errors, showing up as space leaks or references to invalid objects. The solution to this problem is implicit reference counting. In C this is difficult due to the lack of destructors, and while the implementation in the OpenCL C++ runtime was not trivial, due to some strange choices in the OpenCL C API, we needed to implement it only once and it now works for all programs.

In general, there is a one-to-one mapping from classes to C API types but in some cases, such as image types, we choose to break this model and use inheritance. In particular, we felt that OpenCL C's choice to represent image and buffer objects as a single `cl_mem` type was a mistake and did not fit with our original goal of type-safety when possible. This comes into its own when combined with the kernel functor objects, described shortly, where calling kernels is largely type safe, in particular when fully-typed pointer objects are used.

2.5 Information routines

For each OpenCL type, e.g. devices and command queues, the host API provides the ability to query information about the specific object. In general, the query commands are of the form:

```
cl_int clGetXXXInfo(
  cl_XXXid object,  cl_XXX_info param_name,
  size_t param_value_size,  void * param_value,
  size_t * param_value_size_ret);
```

where XXX is replaced by the object type being queried. There are many problems with this C-based API but in particular it is not type safe. Type safety appears in many forms, including restricting the ability to pass any integer value for the enumeration argument cl_XXX_info.

As values returned by clGetXXXInfo routines are not reference counted by the implementation using the C API, this means that the application must first call the routine to determine how much memory will be required, then allocate the required storage, then make the info call again, then use the result, and finally free any allocated storage. This is a lot of code to just get a single piece of information! OpenCL C++ addresses all of these issues through an application of type traits [14]. In general any OpenCL C++ object, often a mapping for an original OpenCL C object provides a member function of the form:

```
template <cl_int name> typename
  detail::param_traits<
    detail::cl_XXX_info, name>::param_type
  getInfo(cl_int* err = NULL) const;
```

All of the interesting work is performed in the class detail::param_traits<...> which determines the result type from the template argument by linking it to the corresponding specialized class that was generated from the original enumeration type.

As an example of its application consider the following, possibly invalid, OpenCL C code:

```
size_t size;
clGetDeviceInfo(device,  CL_DEVICE_LOCAL_MEM_SIZE,
 sizeof(cl_ulong), &size, 0);
```

This code is valid when sizeof(size_t) == 8, i.e. for 64-bit applications, but is invalid when sizeof(size_t) == 4. Worse still this could be the kind of bug that is dormant for a long time, and then would only show itself at runtime. This and similar bugs are ruled out by the OpenCL C++ API as the same program would have been expressed as:

```
size_t size =
    device.getInfo<CL_DEVICE_LOCAL_MEM_SIZE>();
```

In the case the sizeof(size_t) == 4, a compile time error would be reported.

2.6 Defaults

OpenCL C++ introduces defaults for common use cases. In particualar, defaults are provided for:

- Platform, simply pick the first one;
- Device, use the CL_DEVICE_TYPE_DEFAULT macro;
- Context, created from the default device; and
- CommandQueue, created on the default device and context.

The API introduces static member functions for each of the correspoding classes of the form:

```
static Type getDefault();
```

where Type is one of the class types taken from the above list.

While defaults provide a simple approach to writing basic applications, excellent for beginners, the model as descibed quickly comes up short as soon as one of the defaults is not as expected. The problem is that changing one default means that others need to change too. To handle this use case OpenCL C++ supports routines for setting a particular default, which has a transitivie effect, i.e. setting the default device causes the default context and command queue to be updated to reflect this change.

With these small changes, writing simple OpenCL C++ programs is much easier and yet still allows the programmer to gain full control when the more expressive API is required. We believe this is a sign of a good API design, as cut-and-paste of boiler plate API code is reduced considerably, while retaining the same level of expressibility.

2.7 Kernel Functors

The OpenCL interface for kernels is very verbose, breaking the kernel creation, argument setting, and dispatch into seperate APIs. Furthermore, there is no gurantree of static type safety, either for argument types themselves or for the number of actual arguments. To address these short comings OpenCL C++ introduces kernel functors, which provide an interface for creating type safe, directly-callable kernels. The cornerstone of this interface is the following set of definitions:

```
struct EnqueueArgs;

template<class... T>
class KernelFunctorGlobal
{
public:
  ...
  Event operator(EnqueueArgs&, T...);
};

template<class... T>
functionImplementation_
{
  typedef std::function<Event (
    const EnqueueArg&, T...) type_;

  functionImplementation_<T...>(
    const KernelFunctorGlobal<T...>&);

  Event operator()(
    const EnqueueArgs&, T...);

  operator std::function<Event (
    const EnqueueArg&, T...) ();
};

template<class... T>
struct make_kernel :
  public functionImplementation_<T...>
{
  make_kernel( const Program&,
    const std::string,  cl_int *);

  make_kernel(const Kernel, cl_int *);

  make_kernel( const std::string, cl_int *);
};
```

At the base is a single functor class, KernelFunctorGlobal that wraps a kernel object and provides a function call method that encodes the seting of arguments and dispatch into a single intrface. EnqueueArgs is used to wrap the command queue, ndranges, and so on and is elided for space reasons. The interesting aspect of the kernel functor implementation is the public interface exposed by make_kernel, which constructs a kernel functor from either a program and name pair, a kernel object, or just a string representing the program (simply selecting the first kernel to appear in the input).

The design is complicated by having a intermediate step, introduced by inheriting from functionImplementation_, which is not exposed to the user in the CL namespace. This complication is down to propagating types correctly in the C++11 type system, in particular, the kernel functors need to be assignable under the use of C++11's `auto` keyword and thus the implementation "helps" the compiler by propagating types. Additionally, C++11 promotes providing stronger types for functor objects, i.e. `std::function`, and this is supported by providing an overload for the cast operator.

While online compilation offers many benefits for library generators thanks to the ability to build kernels on-the-fly, an unfortunate side-effect is that it is difficult for an implementation to guarantee that `make_kernel` is applied to a source file with the corresponding kernel type. The best we can do without extending the OpenCL API is that if the kernel has been compiled with OpenCL 1.2's `-cl-kernel-arg-info` flag, then some information is embedded in the kernel object to describe the arguments. We can use this to perform a (currently slightly limited) runtime check against the functor type. This information could be extended to better represent the full set of C++ types. If we assert as an invariant that the `make_kernel` call matches its target kernel (and assume therefore that we add full runtime checks in the future) it is possible to show that kernel functors are type safe. Furthermore, this lays the ground work for an offline compilation mode that automatically generates these functor objects, or even merging into a single source style model, similar to Microsoft's C++ AMP [13]. We leave these possible extensions to future work.

3. OpenCL C++ Æcute Programming

Pointers have always been a limiting factor for parallel programming. Numerous techniques exist to either offer guarantees to a compiler that pointers are independent (c.f. `restrict` or OpenMP parallelism guarantees) or to remove pointers entirely from languages. Without such techniques it can be difficult for a compiler to infer indepedence of pointer accesses.

Parallelism can be regained from a loop by letting the programmer make a guarantee. It is more difficult for a compiler to deduce when it can safely move data into software-managed caching structures (such as the local memory regions in OpenCL). In particular, if it does move data, inefficiencies can arise when it is unable to deduce reuse, though it has made an assumption that caching is safe.

The decoupled Access/Execute model [6] aims to alleviate this by separating the execution domain of a problem from its memory mapping. By strictly and statically defining the mapping we open opportunities for the compiler to make a wider set of assumptions about data access and hence perform a wider set of optimizations.

Below we show a very simple example of this technique:

```
typedef cl::Pointer<int,
  cl::Access::Mapping<
  cl::Access::Project<100, 100>,
  cl::Access::Region<3, 3, cl::Access::Clamp>>
  AEcutePointerIN;

typedef cl::Pointer<float,
  cl::Access::Mapping<
  cl::Access::Project<100, 100>>
  AEcutePointerOUT;

kernel void plus(
  global const AEcutePointerIN in,
  global AEcutePointerOUT out)
{
  int2 wid =
    (int2)(get_global_id(0), get_global_id(1));
  float sum = 0.f;
  for( int i = 0; i < 3; ++i ) {
    for( int j = 0; j < 3; ++j ) {
      sum += (float)in(j, i);
    }
  }
  out = sum / 9.f;
};
```

In this example we have two pointers: one for input and one for output. The input pointer is marked const as might be standard practice. In addition to this we have additional and flexible information encoded into the pointer type, which we term the mapping.

First we introduce a projection. As OpenCL defines an iteration space for a given kernel launch the *execute* part of the Æcute model is a combination of this launch metadata and the body of the kernel. The projection is the first stage of the *access* aspect in that it defines a mapping from each point in the execution domain, or the kernel's `NDRange`, to the memory object sitting at the far end of the pointer.

For the output pointer this projection is all we need and we have mapped our flat pointer into a 100×100 memory region. Note that we do not need to index this pointer, and indeed indexing it would remove some of the benefit. We have predefined a simple linear mapping to 2-dimensional memory, in much the same way that an array would be mapped, and this is strictly restricted and hence guaranteed to the compiler.

As this simple example is an instance of a blur filter, we have a precisely defined set of input points needed to compute each output location. We make use of this and encapsulate that information in the pointer's type. In this case we have the same projection of the pointer but it defines only the mapping of the *centerpoint* of the input. We also annotate the pointer with a region such that from each projected point in memory, and according to the shape of the projection, we read a 3×3 region from the input at each point. Access into that region can then be random, as we see at the point of use of `in(j, i)`. Accessing `in` directly returns a `Region` object, which is usable as a small 2-dimensional array in the code.

Note that one benefit of this annotation is that the compiler can assess that neighboring points in the iteration space share input data. The system is hence at liberty to construct shared arrays in `local` memory where data is loaded once and used by neighbouring work items.

The final feature shown in the code example is `Clamp`, allowing us to specify the behavior of accesses out of the projection region, and hence out of the buffer. This can be enforced and tracked in debug mode to assist with code correctness.

This is just a simple taste of our thinking on the application of the Æcute model to OpenCL C++ pointer structures. The goal is to bring a wider set of parallel programming correctness and optimization capabilities to the language in a type-safe and syntactically standard fashion.

4. OpenCL C++ Kernel Language

The OpenCL C++ Kernel language is an extended subset of the newly ratified C++11 specification. Like the C-based kernel language for OpenCL, it defines a large number of builtin operations for parallel execution, e.g. work-group barriers. It includes an extensive math library, which unlike C99's math library, carefully defines precision requirements. To support object construction and destruction placement *new* and *delete* operations are provided. We do not yet provide dynamic memory allocation, mainly due to development effort and to date we have not found a large number of parallel applications that require it. Dynamic memory allocation could easily be added using an approach similar to Xmalloc [7] and in fact our implementation of placement *new/delete* uses ideas similar to those addressed in Xmalloc to avoid contention of atomic memory operations.

OpenCL C++ is more than some smart tricks to reduce the amount of host code. In particular, OpenCL C++ addresses abstraction for GPGPU programming both on the host and also for the device itself. Unlike the more traditional automatically managed cache architectures, such as x86 or ARM, the GPGPU programming model exposes the memory hierarchy to the programmer requiring that it be managed explicitly. To support this, OpenCL borrows address spaces from Embedded C, defining global, local, constant and private address spaces. For example, an OpenCL kernel that uses global, local, and private[2] address spaces to sum the values of a sub-vector in local memory and then write the result into global:

```
kernel void sum(
  global int * in, local int * s,
  global int * out)
{
  s[get_local_id(0)] = in[get_global_id(0)];
  barrier(0);
  for (
   int i = 1; i < get_group_size(0); i <<= 2)  {
     int idx = (i * get_local_id(0)) << 2;
     if (idx < get_group_size(0)) {
       s[idx] += s[idx+i];
     }
     barrier(0);
  }
  if (get_local_id(0) == 0) {
    out[get_group_id(0)] = s[0];
  }
}
```

As expected OpenCL C++ lifts address spaces to work for member functions and member data, within objects. However, it is not as straightforward as it might first seem. For example, consider the following simple structure definition and kernel:

```
struct Colour
{
  int r_, g_, b_;
  Colour(int r, int g, int b);
};

kernel foo(global Colour& gcolour)
{
   Colour pcolour = gcolour;
}
```

Initially one might think this is valid but consider that member functions of a C++ class carry an implicit *this* parameter. What address space should be applied to the *this* pointer for Colour, global or private? In the above example it needs to be both: private for the left hand side of the assignment and global for the right.

Allowing the user to specify the particular address space of a member function, including overloading for different address spaces, goes someway to addressing the problem. However, lifting OpenCL address spaces into OpenCL C++ was not as straightforward as it might first seem. In the OpenCL C version of the above example the copy is done via some internal logic as structures are POD only, but in the C++ version there is, in general, a copy constructor which has an implict *this* pointer argument. But what address space does the *this* pointer live in? The answer is both, local and global.

OpenCL C++ solves this problem by extending C++11's type inference to infer implicit address spaces from context. Furthermore, by novel application of C++11's auto/decltype features for type inference OpenCL C++'s type system allows implicit address spaces to be applied to the explicit use of the *this* pointer within member functions. To our knowledge we are the first to show that OpenCL, and by implication Embedded C's [10], notion of address spaces can be fully integrated within C++'s type system.

[2] Within kernels the private address space is the default.

```
aspace ::= global | local | private

aspace-mod ::=
  aspace | 'address-space' variable

aspace-modifer ::=
  aspace-mod | aspace-modifer '+' aspace-modifer
```

Figure 5. Address space modifier syntax

One solution here would be to say that the compiler generated construtors assume the *global* address space and other constructors are overloaded on the address space explicitly. However, this puts the burden on the programmer and the alternative solution that we adopted was to extend C++'s type system to automatically infer the address space from context when not specified. This fixes the above example. More generally we allow a member function to be marked in one or more address spaces, as defined by the grammar in Figure 5.

The rules for overloading of member functions are extended to include aspace-modifer expressions, e.g. the following example defines a version of `Shape` in the *global* and *local* address spaces and a version for the *private* address space:

```
struct Shape
{
  int setColour(Colour) global + local;
  int setColour(Colour) private;
};
```

Figure 5 also defines the modifer *address-space variable* which allows a template type parameter to carry information about an address space. For this to work in the C++ type system we introduce a new template type parameter, *address-space aspace*, where aspace is a variable that can be bound to one of: *global*, *local*, *or private*. Template equivalence, instantiation, and specialization is extened trivailly to account for address spaces. Adapting the above shape example the following uses address space templates to pass the colour argument by reference:

```
template<address-space aspace_>
struct Shape
{
  int setColour(aspace_ Colour&) global + local;
  int setColour(aspace_ Colour&) private;
};
```

Explictly marking member functions with address spaces works because the corresponding *this* pointer context is known and thus can be easily infered. What about when this is not the case? To see the issue consider implementing assigment for our shape example:

```
XXX Shape& operator=(const XXX Shape& rhs)
{
  if (this == &rhs) { return *this; }
  ...
  return *this;
 }
```

what address space should we put for *XXX*? Templated address spaces provide one solution, however, using them can change the semantics of existing code, in particular delaying type-checking and so on until later use. Fortunately, C++11 provides the answer in its new *auto/decltype* feature. We extend *decltype* inference to infer address space qualifer, with respect to context, for the *this* pointer and require that the above example is recast as:

```
operator=(const decltype(this)& rhs)
                          -> decltype(this)&  {
  if (this == &rhs) { return *this; }
  ...
  return *this;
}
```

In the case of *auto* it is easiler extended to infer the address space for the *this* pointer from context and completes the lifting of OpenCL C's address space qualifers into C++11's type system.

There is a further issue that arises from declaring objects in the local address space due to the side effects of object declaration in C++. Construction of such an object has to be run in each work-item. As a result construction in local memory carries an implicit race condition. While this could be dangerous in some circumstances, it was decided that the overhead of running the same operation in each work item would be low on most devices (due to SIMD execution) and deciding on a set of restrictions for use of classes in local declarations would be unwieldy. Care must hence be taken that the parameter passed to the constructor is uniform, and that the constructor code is also uniform, or races may occur. A formal presentation of address space inference is provided by [5].

5. Implementation

OpenCL C++ is implemented in two parts. Firstly the C++ API is implemented as a single, $14,000$ line, C++ header file. While we would not recommend implementing something like OpenCL C++ in a single header file, this was a requirement set by the Khronos Group and not something we had control over. Most of the implementation techniques are standard and do not require additional mention. Our use of traits to implement type safe information functions is novel and we have not found other examples, however it should be noted that it is just an application of template metaprogramming, itself well documented, c.f [4].

As OpenCL C++ provides a thread safe API the implementation of defaults proved difficult for two reasons:

- Implementing state in a C++ header file, including initializers, leads to duplicated symbols. The solution for this was via weak symbols, not part of the C++ standard and thus required using GCC and Visual Studio extensions.
- Introducing shared state means implementing mutual exclusion for the creation of defaults. This was done with a lazy implementation of the double locking pattern, using spin locks[3].

Due to limitations of current C++11 compilers with respect to variadic templates[4], our implementation of kernel functors is based on type lists [1]. The number of valid kernel arguments is restricted to 32 but as host compilers currently implement similarly low limits for std::function and structures can easily be used to wrap arguments this is unlikely to be an issue. Once variadic templates are working in multiple compilers we expect to move the implementation to use them, as it will reduce the code complexity considerably.

The OpenCL C++ API has been validated on all public OpenCL implementations, including Apple's OpenCL for OS X Lion, Intel's OpenCL SDK, and AMD's OpenCL SDK. The latest OpenCL C++ API was released in 2012 and contains all but the Æcute features described in this paper.

[3] Our original implementation discovered a GCC bug with compare-and-swap on OS X Lion.

[4] Only GCC 4.6 supports varadic templates and we found that certain examples failed to generate correct code.

The second part of the OpenCL C++ implementation is the device compiler. In this case we had to pick a specific implementation and as we had access to AMD's compiler technology we choose that. AMD's OpenCL compiler is based on EDG's C++ frontend, which is a well designed and validated C++11 frontend. We modified AMD's existing OpenCL frontend to first work as an OpenCL C++ compiler and then extended it to support the automatic inference of address spaces. Overall this process was straightforward, although we found that some corner cases where only resolved by the development of a ever growing test suite.

The OpenCL C++ device language, as described in this paper is part of AMD's APP SDK 2.6 and can be downloaded from developer.amd.com. Some of the features, particularly the host code, are likely to be adopted more widely in Khronos.

6. Evaluation

OpenCL C++ has not been implemented with the aim of increasing application performance. The primary goals are to:

- Ease development of applications by reducing line code count and increasing the amount of reuse possible in kernels through inheritance.
- Reduce the rate of errors in code, largely through automating some of the necessary memory management.
- Wrap the OpenCL host API as thinly as possible. A thin wrapper then provides minimal performance degredation and maximal flexibility, minimising negative effects arising from its use.

To this end our evaluation consists of two analyses. We take five OpenCL applications, ranging from trivial to fairly complicated. For each application we implement the OpenCL operations both in C and C++. We measure the performance of each as well as the line count differential.

The applications we chose are:

- Vector addition: the "hello world" of data-parallel programming and likely to be one of the first applications a developer is exposed to.
- Pi computation: a computation of the value of pi that acts as a small self-contained unit of work.
- Ocean simulation: a heterogeneous CPU/GPU FFT-based approximation of an ocean surface for rendering.
- Particle simulation: a highly heterogeneous particle simulation performing collision detection and solution on both the CPU and GPU with interactions between the two.
- Radix sort: a high performance radix sort optimised for the GPU and contained within a sophisticated abstraction library for integration with larger applications.

Unfortunately evaluation metrics for the C++ device code are harder to construct. The implementation certainly works and we have high performance complicated applications implemented internally using its features including a template-driven radix sort of similar performance to that used in this paper. Some of these implementations will be made public along with access to the compiler in the near future.

6.1 Performance differential

Performance measurements for all five examples are unexciting, though successful. In all cases performance is identical to within random fluctuations. For example, the OpenCL C++ Ocean implementation drifts within one frame-per-second up or down of the OpenCL C version.

The minimal performance differential observed is to be expected given that the C++ layer is a very thin wrapper around C functions. The cases where it might add overhead where, for example, all arguments are set repeatedly in a functor rather than factoring out some argument setting from a loop can be observed to be rare in practice. The development and code management overhead of maintaining parameter passing spread throughout the code makes it unusual in practice and though the final two examples used here are real applications the case does not come up.

One problem that appeared during testing was that performance of functor interfaces for kernel dispatch was significantly below that of the C APIs. Given the thin nature of the C++ bindings this was surprising.

The solution turned out to be that, for correct behavior, the C++ interfaces copied objects: in particular the launch descripion. This interface contains a queue reference that needs to correctly reference count as the EnqueueArgs struct needs to strictly own a reference to the queue to avoid it being deleted before use. This reference count created a serious performance bottleneck due to a bug in the runtime implementation that led to flushing the queue when releasing a reference to the queue object. So while in the process it appeared that maybe the C++ interfaces caused overhead, in reality they served to identify a runtime bug and performance overhead from reference counting is negligable. On the other hand, safety improvements resulting from the resource aquisition is initialization (RAII) model may be significant.

6.2 Code size

Depending on the use case, code size savings can be substantial, particularly if the code structure allows for use of functors or the application is simple enough for default entities to be valid. Note that default entities are rarely likely to be useful for seasoned programmers who wish to have full control. However, the code size reductions in trivial "hello world" applications substantially reduce the barrier of entry to OpenCL programming.

In the following table we show the line count of relevant files using OpenCL's C API and the OpenCL C++ API. We defined "relevant" as any file with an OpenCL API call in. Code not directly relating to OpenCL API calls is unchanged and the coding standards of the surrounding code are matched as closely as possible.

Application	C lines	C++ lines
Vector addition	268	140
Pi computation	306	166
Ocean simulation	1386	533
Particle simulation	733	601
Radix sort	627	593

Note that the final two applications are part of larger complex frameworks and the OpenCL code was abstracted by the author into a complicated class library. Even in these examples we see gains, but the majority of the code is that generic framework rather than OpenCL API code.

7. Conclusion

In this paper, we have presented OpenCL C++, a production development system based on C++11 that supports OpenCL enabled devices and platforms. We have shown that this can give large productivity results over the existing OpenCL programming model based on C99, and furthermore have done this without modification to the host compiler, unlike solutions such as Cuda.

OpenCL C++'s kernel language is the full C++11 dialect, albiet with some restrictions on the set of supported library routines. We have shown how OpenCL's address spaces, and by implication Embedded C's, can be lifted and extended to work within C++11.

OpenCL C++ is a practical abstraction on top of OpenCL's C interface, providing abstraction for both the host and device components of an application. Many of the features are public and applications are or will soon, be shipping.

OpenCL C++'s Æcute extensions to pointers looks promising and we are actively researching their implementation. We hope to be able to release a public implementation soon, allowing developers to explore its application to real world problems. Combining this work with our work for GPUs applying C++11's memory model, including communication across work-groups and the complete platform, we believe we can bring OpenCL in reach of mainstream parallel programmers.

References

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *In Proceedings of the 2001 ACM SIGPLAN conference on programming language design and implementation (PLDI)*, pages 114–124. ACM, 2001.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[4] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-30977-7.

[5] B. R. Gaster and L. Howes. Formalizing address spaces with application to cuda, opencl, and beyond. In *Proceedings of the 6th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-6, New York, NY, USA, 2013. ACM.

[6] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, Lecture Notes in Computer Science. Springer, 2009.

[7] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1134–1139, Washington, DC, USA, 2010. IEEE Computer Society.

[8] Intel. 2^{nd} Generation Intel Core Processor Family (Codename Sandy Bridge). http://software.intel.com/en-us/articles/sandy-bridge/, 2011.

[9] ISO/IEC. Programming languages C++. 14882:2011(E), 2011.

[10] ISO/IEC. Programming languages Embedded C. DTR 18037, 2011.

[11] D. Merrill and A. S. Grimshaw. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(2):245–272, 2011.

[12] D. G. Merrill and A. S. Grimshaw. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 545–546, New York, NY, USA, 2010. ACM.

[13] D. Moth. Taming GPU compute with C++ AMP. *Channel 9*.

[14] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[15] NVIDIA Corporation. CUDA programming guide, version 4.x, 2011.

[16] OpenCL Working Group. The OpenCL specification, version 1.1, revision 36. Khronos, 2010.

[17] P. Rogers. The programmer's guide to the APU galaxy. In *Proceedings of the 2011 AMD Fusion Developer Summit*, 2011.