CAPP : C++ Aspect-Oriented Parallel Programming with AspectC++ and OpenCL

Robert Clucas
University of the Witwatersrand
1 Jan Smuts Avenue
Johannesburg, South Africa
robert.clucas@students.wits.ac.za

ABSTRACT

Parallel programming provides higher computational performance over sequential implementation if the program is parallelizable. However, external parallel capable devices are required to perform the parallel computation. Current programming API's such as OpenCL and CUDA which provide an interface for parallel programming require substantial amounts of code for performing even the simplest parallel computations. This leads to code which includes cross-cutting components such as setting up the parallel programming context, compiling the parallel kernel, and transferring data between the host and device memory spaces when the kernel is executed. An Aspect-Oriented Parallel Programming model is developed, using AspectC++, which uses an abstract aspect to remove the cross-cutting components from the C++ code. The aspect sets up the OpenCL context, compiles the OpenCL kernel, and manages the data transfer between the memory spaces each time a kernel is executed. The aspect is woven into the C++ code before compilation rather than at runtime, which improves performance. The result is simple C++ code which does not require any OpenCL code and is void of cross-cutting concerns, and allows a parallel kernel to be executed with a single function call. The model was applied to the SAXPy and Black-Scholes option pricing problems. Computational performance was 7-11% slower than the OpenCL implementation while the amount of code was greatly reduced from the OpenCL solution, and model C++ files was comparable in structure and length to native C++ implementation.

CCS Concepts

•**Computing methodologies** → **Parallel programming languages;** •**Computer systems organization** → *Parallel architectures;* •**Software and its engineering** → *Context specific languages;*

SAICSIT '15 September 28–30, 2015, Stellenbosch, ZAR

DOI: 10.1145/1235

Keywords

Aspect; device; host; parallel; cross-cutting

1. INTRODUCTION

In recent years CPU core frequencies have started to plateau [7]. Multiple core CPUs and many core GPUs were the result of this plateau. The cores used for CPUs and GPUs differ in complexity and are hence advantageous for different tasks. GPUs use many (up to thousands) simple cores to increase computational efficiency, while CPUs use fewer, complex cores which generally have higher frequencies than those used in GPUs. Due to the number of cores available when using a GPU, GPUs are suited to data parallel tasks where the same operation can be performed on each element of a high dimensional dataset. Modern CPUs can also provide data parallelism through single instruction multiple data (SIMD) operations and their numerous cores. However, due to having fewer cores they provide less dramatic increases in performance and require large amounts of power to perform the instruction level parallelism [12]. General-Purpose GPU (GPGPU) programming involves combining CPUs and GPUs into a single, hybrid system which provides increased computational performance by using the CPU (*host*) to pass data to the GPU (*device*) which performs the computation on the data in a parallel manner. For GPGPU programming there are two main API's available to the programmer - OpenCL [10] and CUDA [14].

OpenCL is an attempt to provide a standard API for programming parallel capable hardware. It provides support for all main CPU and GPU hardware vendors, namely Nvidia, AMD and Intel. This wide range of support is advantageous as a parallel implementation using the OpenCL API could run simultaneously on the CPU and GPU, maximising the hardware capabilities of the parallel system. CUDA applies a similar methodology and also provides an API for writing programs which can be executed on parallel capable hardware, however, it is specific to Nvidia hardware and hence parallel kernels cannot be executed on CPUs or GPUs from any other hardware vendor.

These parallel systems are more difficult to program than sequential systems. This is mostly due to the addition of the external *devices*, their low level nature and that communication is required between them and the *host*. To perform parallel computation on the *device* using OpenCL the generalised sequence of events, which is similar to the generalised sequences of events for a CUDA C program [8], is (The colors after the description relate to the examples given in Listings 1 and 2):

1. Initialize the data on the *host* (Green)
2. Setup the OpenCL variables, which involves (Red):
   - Setting the OpenCl platform
   - Creating the OpenCL Context
   - Setting the parallel device(s) to use
   - Creating the OpenCL command queue
3. Create the OpenCL kernel (Blue)
4. Create the OpenCL buffers and move the data from *host* to *device* (Magenta)
5. Execute the kernel on the parallel device(s) (Green)
6. Move the results from *device* memory back to the *host* memory (Magenta)

This is a substantial amount of work to perform parallel computation. Comparatively, to perform the same computation on the *host* only steps 1 and 5 defined above are required. All other steps increase the complexity and cross-cut the main intention of the C++ core code (code directly related to the problem) which for parallel programming is the computation of an algorithm on the data. This overheard is illustrated by a comparison of Listings 1 and 2 which provide a simple vector addition implementation using C++ and OpenCL respectively.

The need for GPGPU programming arises from the computational complexity of algorithms which results in non-GPGPU systems not being able to perform the computation in a sufficiently small amount of time. The additional complexities make parallel development inefficient and require programmers to learn the complex OpenCL or CUDA API's, hence programmers are generally reluctant to to learn that which is required for parallel programming. However, the computational performance increase provided by GPGPU programming is still desired and thus the complex, low-level interaction between the *host* and *device* must be overcome to achieve the gain in performance.

To remove the requirement of the programmer having to deal with the parallel programming complexities, Aspect-Oriented Programming (AOP) [11] can provide a solution which allows the cross-cutting components to be modularized into aspects which are not part of the C++ files which define the core code, but separately defined. This results in simple C++ core code consisting of code only directly related to the program's intention. The aspects are *woven* into the C++ files before compilation so that when the code is compiled the cross-cutting code is present.

An AOP implementation for C++ has been available since 2001 in the form of AspectC++ [5] [18]. However, the use of aspects in low-level parallel programming is limited. While there are very few examples of using aspects to modularizes the complexities of parallel programming numerous proposals have been made which aim to reduce the complexities using object-orientated API's or new parallel languages. This paper presents the C++ Aspect Parallel Programming *CAPP* model which makes use of aspects, implemented using AspectC++, to hide the above mentioned complexities present in OpenCL parallel programming from the programmer.

```
#define T float
void VectAdditionC++(int argc, char** argv) {
  // Instantiate vector add class
  VectAddCppClass vectAdd;
  // Declare data vectors
  vector<T> in1, in2, out;
  // Fill data vectors
  for (int i = 0; i < NUM_ELEMENTS; i++) {
    out.push_back(0.f);
    in1.push_back(rand());
    in2.push_back(rand());
  }
  // Execute Kernel
  vectAdd.RunKernel(in1, in2, out);
}
```

Listing 1: Vector addition using C++

The aspects modularise both static components - the initialisation of the OpenCL variables - and dynamic components - creating the relevant buffers and allocating and deallocating memory on the host and device when kernels are executed. The kernel function is not dealt with by the *CAPP* model. This decision was made as the kernel is specific to the implementation of the algorithm and would involve incredible complexity to allow a general aspect to convert a C++ function to an OpenCL kernel without loss of performance. To write an OpenCL kernel the programmer does not need to learn the OpenCL API, however, does require an understanding of how parallel computation it performed in terms of the thread arrangement. It is a assumes that the programmer would have an understanding of how their algorithm is parallelized and hence could implement the kernel. Future improvements which remove this requirement are discussed in Section 7.

The modularize the parallel code the *CAPP* model has two main goals:

- To allow for a parallel implementation which is comparatively simple, in terms of code structure and number of line, with a sequential, C++ only implementation
- To provide performance which is comparable with a non-aspect implementation written using OpenCL or CUDA

The rest of the report is structured as follows: Section 2 reviews work related to the simplification of parallel programming; Section 3 describes the *CAPP* programming model and the use of AspectC++, and its features, to achieve the above mentioned goals; Section 4 presents the results of *CAPP*, CPU and OpenCL implementations to two problem domains: SAXPY and Black Scholes option pricing; Section 5 discusses the results and comments on the limitations of the *CAPP* model; Section 6 concludes and Section 7 provides possible directions of exploration for future work in this area.

2. RELATED WORK

With the increasing popularity and necessity of parallel implementations, attempts to minimise the complexity of writing parallel code have both increased in number and made the task simpler. OpenCL and Nvidia CUDA are both examples of this. They provide extensions to the C language which are interfaces to the low-level parallel hardware. As

```
#define T float
void VectAdditionCl(int argc, char** argv) {
  // Declare OpenCl variables
  vector<cl::Platform> platforms;
  vector<cl::Device>   devices;
  vector<cl::Buffer>   buffers;
  cl::Context          context;
  cl::CommandQueue     queue;
  cl::Program          program;
  cl::Kernel           kernel;
  // Declare data vectors
  vector<vector<T>>    inputs;
  vector<T>  in1, in2, out;
  // Fill data vectors
  for (int i = 0; i < NUM_ELEMENTS; i++) {
    out.push_back(0.f);
    in1.push_back(rand());
    in2.push_back(rand());
  }
  inputs.push_back(in1); inputs.push_back(in2);
  // Get OpenCL Platforms
  clPlatform::get(&platforms);
  // Create context parameters
  cl_context_properties cps[3] = {
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platforms[0])(),
    0
  };
  // Create OpenCL context
  context = cl::Context(CL_DEVICE_TYPE_GPU, cps);
  // Get available devices
  devices = context.getInfo<CL_CONTEXT_DEVICES>();
  // Create command queue
  queue = cl::CommandQueue(context, devices[0]);
  // Get kernel source
  ifstream kSource("vectadd.cl");
  // Convert kernel source to string
  string kstring(
    istreambuf_iterator<char>(kSource),
    (istreambuf_iterator<char>()) );
  // Create OpenCL program source
  cl::Program::Sources source(1,
    make_pair(kstring.c_str(),
              kstring.length() + 1) );
  // Create OpenCL program
  program = cl::Program(context, source);
  // Create OpenCL kernel
  kernel = cl::Kernel(program, "vectadd");
  // Create input buffers
  for (auto& input : inputs) {
    buffers.emplace_back(context,
      CL_MEM_READ_ONLY,
      input.size() * sizeof(T) );
    queue.enqueueWriteBuffer(buffers.back(),
      CL_TRUE, 0, input.size() * sizeof(T),
            &input[0] );
  };
  // Create output buffer
  buffers.emplace_back(context, CL_MEM_WRITE_ONLY,
    out.size() * sizeof(T) );
    // Set kernel arguments
  for (int i = 0; i < buffers.size(); i++) {
    kernel.setArg(i, buffers[i]);
  }
  // Set thread dimensions
  cl::NDRange global(NUM_ELEMENTS);
  cl::NDRange local(1);
  // Execute Kernel
  queue.enqueueNDRangeKernel(kernel,
    cl::NullRange, global, local);
  // Get results from GPU
  T* res[NUM_ELEMENTS];
  queue.enqueueReadBuffer(buffers.back(),
    CL_TRUE, 0, out.size() * sizeof(T), res);
}
```

Listing 2: Vector addition using OpenCL highlighting the different cross-cutting components

mentioned in Section 1 this introduces cross-cutting code which 'tangles' the core code. Additionally the API's are extensive which require the programmer to invest a significant amount to learn before writing a parallel program.

APIs, frameworks and entirely separate languages have been proposed which hide the low-level interactions with the parallel capable from the programmer. These systems generally provide wrappers around the low-level interfaces which often reduces the performance - a key consideration in parallel programming.

CuPP [1] provides a C++ framework designed to increase the ease of implementing parallel applications using CUDA. It provides both low-level and high-level interfaces which interact with the parallel capable hardware, hence providing a C++ interface for programming parallel applications. This is also an attempt to wrap the cross-cutting code into modules such that it is hidden from the programmer, but in an object-oriented manner. The objects-orientated additions are not external to the C++ code resulting in more code than what is required for a traditional CUDA implementation. Since the framework is built around CUDA, support is only provided for Nvidia devices.

CAF [4] [3] uses the actor model to allow computation to be performed in a distributed manner. It provides bindings for OpenCL which allow the programmer to specify only the OpenCL kernel from which CAF creates an actor capable of executing the kernel. The framework allows computation on a vast number of parallel capable devices, which is a notable feature. However, providing support for so many devices introduces overhead and makes the framework very large. Furthermore it requires learning the actor model which introduces additional complexity into the parallel programming process.

C++ AMP [13] provides extensions to the C++ language which allow data parallel capable hardware to be used to accelerate computation. C++ AMP is only available for the Windows environment and is thus limited in scope which is contrary to the generality the proposed aspect implementation aims to provide.

RapidMind [16] and Brook [2], while are similar to standard C++, essentially define new languages for parallel programming.

RapidMind provides a parallel programming environment which, by taking advantage of C++ template metaprogramming, provides the programmer with three data types: *Value*, *Array*, and *Program*. The *Value* and *Array* data types hold data elements and groups of data elements, respectively. The *Program* data type stores operations which can act on *Array* data types in a parallel manner. The *Program* data type is essentially the same concept as the kernel provided by Nvidia and OpenCL, but in the C++ language. RapidMind also makes use of macros which add complexity to the code, rather simplifying the code - which is the intention of aspects.

Brook provides high-level abstractions to hide the low-level parallel programming complexities from the programmer. Similarly to RapidMind there are three abstractions which create the high-level parallel programming environment: *Stream*, *Kernel*, and *Reduction*. The *Stream* deals with the data, while the *Kernel*, similar to both the kernel in the Nvidia and OpenCL and the *Program* data type for RapidMind, allow operations to be defined which act on the *Streams* (data). The *Reduction* is provided to generate a

result from a high dimensional *Stream*.

Both RapidMind and Brook do a lot of work at runtime to move the data to the *device* mememory and to allow parallel computation to be performed on the *device*. This reduces their performance when compared with CUDA or OpenCL implementations [20] which has led to these parallel environments not gaining high levels of acceptance within the parallel development community.

Work done on the use of aspects for parallel programming is limited, especially for low-level programming which interfaces directly with the parallel hardware. Use for aspects in a parallel context is proposed by [17]. However, Java is used and the model is based on multi-core CPUs and hence does not include the numerous complexities which arise from having additional computational devices such as GPUs.

A new parallel aspect language, based on AspectC++, is proposed by [19]. Features closely related to AspectC++, but more specific to parallel programming, are defined to hide most of the cross-cutting complexities from the programmer. They manage to reduce the cross-cutting components considerably and allow the core computations to be defined using C++. The aspect model defines a *kernel* feature which behaves in the same way as *advice* in AOP. Furthermore the memory structure of the device is encapsulated using templates and allows the programmer to specify the type of *device* memory to be used in C++. They provide a compiler to weave the aspects defined using their aspect language into the C++ core code. A performance reduction of only ≈20% of the CUDA implementation was achieved. The aspect language and compiler are not yet fully functional as well as being specific to parallel programming. This specificity is beneficial for parallel only applications, however, does not lend itself to large C++ applications where only specific components may benefit from parallel acceleration. Thus at this stage could not be used to modularize non-parallel cross-cutting concerns into aspects.

The *CAPP* model provides a few advantages over the related work: (1) The aspect components use AspectC++ which is based on standard C++ and allows the core code to be written in standard C++ rather than providing an alternative language for parallel programming. This is beneficial for large systems as cross-cutting concerns arising from non-parallel complexities can be modularised using AspectC++. (2) Low-level parallel programming complexities are hidden from the programmer using aspects. This results in C++ core code which has no components that cross-cut the code's intention, as well as kernel functions which can be executed from the C++ core code with a single function call. (3) There is negligible loss in performance since the aspects are woven into the C++ core code before compilation and hence do not have runtime implications other than the necessary OpenCL related work.

3. C++ ASPECT-ORIENTED PARALLEL PROGRAMMING (CAPP) MODEL

The *CAPP* model presented in this section allows the programmer to write parallel programs in traditional C++ with the addition of specifying the kernel function to perform tasks on some data in a parallel manner. The aspects are written using AspectC++ and provide the functionality of the cross-cutting OpenCL code, hence 'untangling' the C++ core code. The aspects are then woven into the C++ core class before compilation to allow the cross-cutting functionality to exist within the C++ core class at compile time. A high-level representation of the weaving process, and the minimum OpenCL functionality which is woven into the C++ core class, is shown in Figure 1. The rest of this section explains the specific use of AspectC++ and it's features in the *CAPP* model to provide a modular parallel programming environment. The OpenCL vector addition example of Listing 2 is the reference from which the subsequent aspect Listings are based.

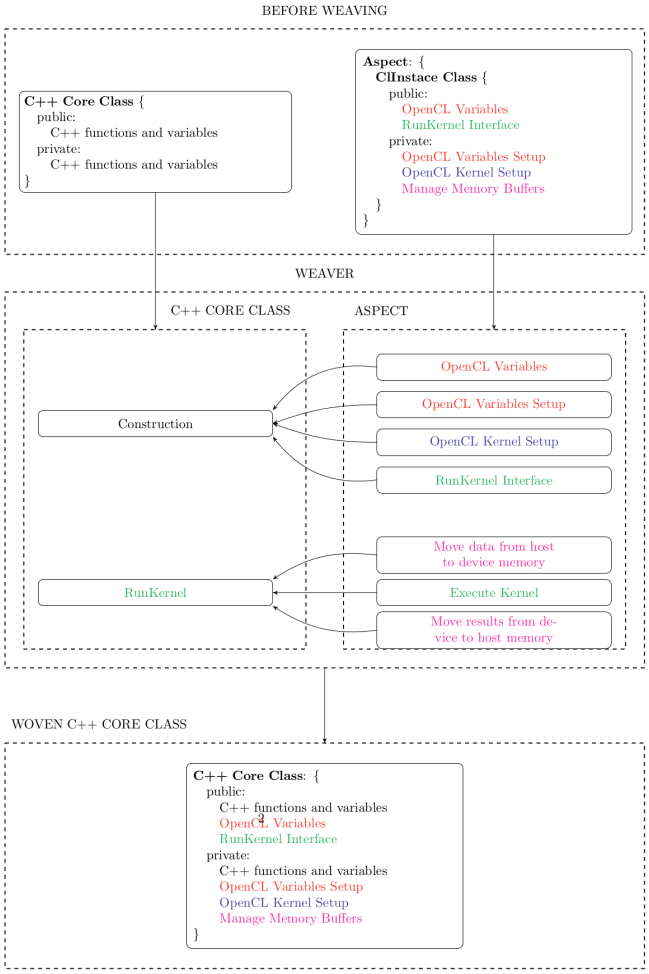

Figure 1: High-level overview of the weaving process for the *CAPP* model. The output of the weaving process produces the code for compilation.

3.1 Abstract ClContext Aspect

AspectC++ is based on C++ and hence provides functionality for both inheritance and polymorphism. The *CAPP* model makes use of these features and defines an abstract aspect, the *ClContext* aspect, which provides the core C++ code with the cross-cutting OpenCL components necessary for executing a parallel kernel. The OpenCL components, both variables and member functions, are defined in the abstract aspect and are inherited by the derived aspect. Any additional OpenCL functionality (which may improve specific parallel implementations) can then be defined in the derived aspect. Polymorphism is provided by AspectC++

through virtual pointcuts, which were used in the abstract *ClContext* aspect to define an interface for the derived aspect to specify which C++ core classes require parallel capability. The *CoreClasses* pointcut in Listing 3 shows this. The abstract aspect thus provides all the necessary OpenCL functionality which must be present for a parallel capable class, without defining which C++ class the aspect functionality should be woven into. Listing 4 shows an example of a derived aspect for a C++ vector addition class (VectAddCppClass), which uses the virtual pointcut *CoreClasses* to define the C++ class into which the functionality must be woven.

3.2 OpenCL Variable Introduction

Observing Listing 2 there numerous OpenCL specific variables (variables defined below the red comments) which are required to setup the OpenCL environment. These are only the necessary variables, there are many more which provide additional features for parallel programming. Many of these variables must be modified or recreated each time a new kernel needs to be executed, requiring a significant amount of overhead when compared to the C++ version of Listing 1.

AspectC++ provides the *slice* feature for defining C++ classes within aspects. Furthermore, it allows for *advice* to be defined for the virtual pointcut. This advice can be implemented to use C++ inheritance which will allow aspect weaver to weave the *slice* class as a base class for any C++ core classes, or to derive from any C++ core classes. Using these features, the *CAPP* model defines the *ClInstance slice* class within the abstract *ClContext* class. The *ClInstance* class has the cross-cutting OpenCL variables as public variables, and the cross-cutting OpenCL operations as private member functions. The weaver then weaves the *ClInstance* class to be a base class of the C++ classes defined by the *CoreClasses* pointcut so that the OpenCL variables are part of those C++ classes. Listing 3 shows the *CLInstance slice* class within the abstract aspect, as well as the cross-cutting variables (below the red comments as per Listing 2) and member functions which provide the cross-cutting OpenCL setup functionality.

3.3 OpenCL Setup Function Introduction

Again observing Listing 2, a large amount of the cross-cutting code comes from the operations on the OpenCL variables. These operations (below the red comments in the Listings) are for the setup of the OpenCL environment, and determine the available devices and platforms, and create the OpenCL context. They cross-cut the C++ core code since they are not directly related to the intention of the program. Furthermore, these operations are generic - they are the same for any OpenCL program - allowing them to be defined as functions of the *ClInstance slice* class. The operations for creating the kernel (below the blue comments in Listings) have the same problems and can therefore also be defined as functions of the *ClInstance slice* class.

The *ClContext* aspect then defines the *ClSetup* pointcut to specify where in the C++ classes these functions should be executed. The *CAPP* model defines the functions to be executed on construction of the C++ class. This is done so that once the C++ class has been instantiated, all OpenCL variables have been initialized to the appropriate values which allow parallel computation from the C++ class.

To provide the programmer with flexibility regarding the

```
// ClContext.ah
aspect ClContext {
  class ClInstance {
  public:
    // OpenCL variables
    vector<cl::Platform> platforms;
    vector<cl::Device>   devices;
    vector<cl::Buffer>   buffers
    cl::clContext        context;
    cl::CommandQueue     queue;
    cl::Kernel           kernel;
    cl::Program          program;
    // Other variables and functions
  private:
    // Setup OpenCL variables
    void SetupOpenCl(string devType, string
        kSource);
    // Setup OpenCL kernel
    void SetupKernel(string kSource, string kName)
        ;
    // Manage buffers on kernel execution
    void ManageBuffers(vector<vector<T>>* inputs,
                       vector<vector<T>>* outputs)
                           ;
  };
  // Interface for defining C++ core classes
  pointcut virtual CoreClasses() = 0;

  // Specify ClInstance as baseclass of classes
  // defined by CoreClasses()
  advice CoreClasses() : baseclass(ClInstance);

  // Other aspect components
};
```

Listing 3: Abstract ClContext aspect which defines the OpenCL variables required for parallel programming

```
#include ``ClContext.ah"
aspect VectAdd : ClContext {
  // Define C++ core class to weave OpenCL
  // functionality into
  pointcut CoreClasses() = "VectAddCppClass";
};
```

Listing 4: Derived aspect defining the C++ core classes which require the OpenCL variables

parallel context, the *ClSetup* advice makes use AspectC++'s *tjp* pointer. The *tjp* pointer provides the aspect with access to the C++ class, into which the aspect will be woven. Using *tjp->that()* in an aspect is equivalent to using *this* within a C++ core . This feature was used to allow the programmer to specify the computation device (CPU or GPU), the kernel source file, and kernel name as arguments of the C++ class constructor. Using *tjp->that()* allows the aspect to use the values provided by the programmer from the C++ class. The aspect can then set up the OpenCL environment to the programmer's preference. Listing 5 shows the definition of *ClSetup* pointcut, and its advice which calls the private member functions of the *ClInstance* class, as defined in Listing 3.

3.4 RunKernel Interface

Executing the kernel is the main component of a parallel application. Before the kernel is run, the data on which the kernel operates must be moved from the memory of the *host* to the memory of the *device*. Once the kernel has finished

```
// ClContext.ah
aspect ClContext {
  class ClInstance {
    // As per Listing 3 ...
  };
  // Other pointcuts and advice

  // Pointcut which defines where the OpenCL
  // setup function should be woven
  pointcut ClSetup = construction(CoreClasses());

  // Advice specifying how the OpenCL
  // setup functions should be woven
  advice ClSetup() : around() {
    // Setup OpenCL variables
    tjp->that()->SetupOpenCl(
      tjp->that()->devType, tjp->that()->kSource);
    // Setup OpenCL kernel
    tjp->that()->SetupKernel(
      tjp->that()->kSource, tjp->that()->kName);
    // Continue with core class constructor
    tjp->proceed();
  }
};
```

Listing 5: Abstract aspect with the pointcut and advice for OpenCL setup

executing, the results reside in the memory of the *device* and are not available to the *host* until transferred back from the *device* memory to the *host* memory. In OpenCL this is done using the cl::Buffer type and specifying if the *device* memory must be read from or written to. These operations are shown in Listing 2 below the magenta comments.

This process of memory allocation, movement, and deallocation on each call of the kernel is not required for a C++ implementation and therefore cross-cut's the program's main intention. Furthermore, memory management is a difficult problem in the C and C++ languages, which is exaggerated by the introduction of the *device* memory. As these operations are required each time the kernel is executed, the amount of cross-cutting code growing linearly with the number of kernels and kernel calls.

These problems are solved by the *CAPP* model through aspects by defining a pure virtual *RunKernel* function in the *ClInstance slice* class. Since the *ClInstance slice* class will be a baseclass of any C++ class requiring parallel functionality, the *RunKernel* function implementation must be defined in the C++ class. By defining the *RunKernel* function as a pure virtual function the aspect knows the structure of the function, which allows the *ClContext* aspect to define advice which can use the arguments passed to the *RunKernel* function by the programmer, to perform the cross-cutting memory management code each time the *RunKernel* function is called from the C++ class.

The *RunKernel* function specifies that the arguments provided to it from the C++ code must be vectors, which hold the input and output data for the kernel. This allows the C++ class defined by the programmer to execute a kernel by simply calling the *RunKernel* function and passing the inputs and outputs as function arguments, as shown by Listing 7. The *tjp* pointer is used again by the aspect, but this time to access the *RunKernel* function arguments, and hence the input and output data for the kernel so that the memory can be appropriately managed between the *host* and *device*.

Listing 6 shows the *ManageKernel* pointcut and advice, which calls the *ManageBuffers* function in the *ClInstance*

```
// ClContext.ah
#define T float
aspect CLContext {
  class ClInstance {
  public:
    // OpenCL variables ...
    // Execute kernel
    virtual void RunKernel(
      vector<vector<T>>& inputs,
      vector<vector<T>>& outputs) = 0;
  private:
    // Setup OpenCL variables ...
    // Setup OpenCL kernel ...
    // Manage memory buffers
    void ManageBuffers(
      vector<vector<T>>* inputs,
      vector<vector<T>>* outptus);
  };
  // Other advice and pointcuts

  // Pointcut defines where the memory
  // buffers should be managed
  pointcut ManageKernel() =
    execution("% ...::...::RunKernel(...)") &&
    within(CoreClasses());

  // Advice specifies how to manage buffers
  advice ManageKernel() : before() {
    // Manage memory buffers
    tjp->that()->ManageBuffers(tjp->arg<0>(),
                               tjp->arg<1>());
  }
};
```

Listing 6: Abstract aspect components which hide kernel cross-cutting concerns

```
#define T float
int VectAddCpp(int argc, char** argv) {
  // Instantiate vect addition class
  ParVectAddCppCore vectAdd;
  // Create input and output buffers
  vector<vector<T>> inputs;
  vector<vector<T>> outputs;
  // Fill vectors with data ...

  // Execute kernel
  vectAdd.RunKernel(inputs, outputs);
}
```

Listing 7: C++ core class executing a pallel kernel using C++

class, to perform the cross-cutting memory management between the *host* and *device* each time the *RunKernel* function is called from the C++ class.

4. RESULTS

This section presents the results of application of the *CAPP* model to two GPGPU programming problems. The first problem is known as SAXPY (Single-precision a*X + Y) [9], and is relatively simple to implement in parallel. The second problem is the evaluation of option prices using the Black-Scholes option pricing model [15], which is more complex than the SAXPY problem, and hence applies the *CAPP* model to more 'tangled' C++ code. Results are given for both the performance and number of lines of code for the *CAPP*, C++ (CPU only), and OpenCL implementations to demonstrate both the performance capabilities and the code simplification a *CAPP* implementation can provide.

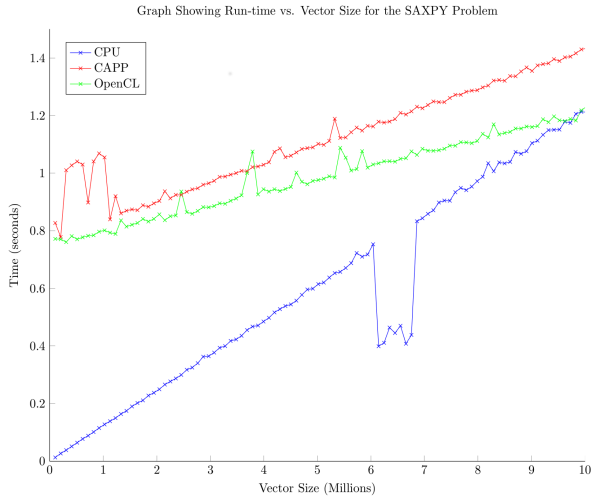

Figure 2: Results for the three implementations of the SAXPY problem. Lower bar height is better.

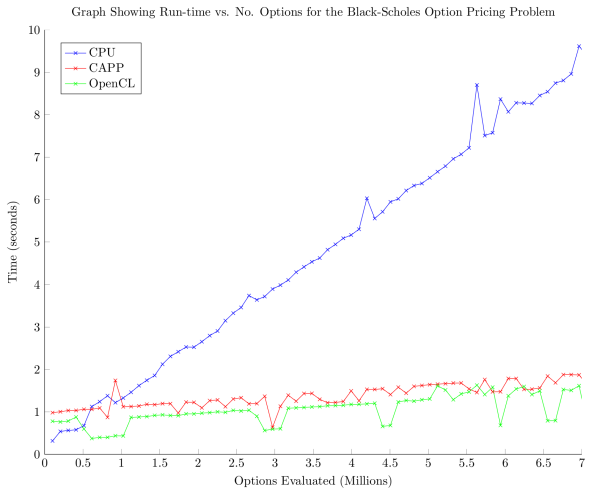

Figure 3: Results for the three implementations of the Black-Scholes option evaluation problem. Lower bar height is better.

4.1 SAXPY Problem

Various vector sizes were used and the runtime of each of the implementations was measured to validate that the solution using the *CAPP* model scales similarly to the OpenCL implementation. Figure 2 shows the results for the runtime of each implementation. Transferring the data between the CPU and the GPU is a large overhead for the SAXPY problem [6]. This is evident by the offset of the OpenCL and CAPP results for small vector sizes. Since the CPU implementation doesn't require data transfer to an external device the offset is not present. The gradient of each of the results provides important information as it shows how each of the implementations scale with an increase in the data size. The gradient of the CAPP (0.61) and OpenCL (0.48) solutions are comparable, with the *CAPP* solution scaling slightly worse. The gradient of the C++ implementation (1.19) is considerably worse and does not scale particularly well. The average performance of the *CAPP* implementation was 7% slower than the OpenCL implementation which the worst case performance being 21% slower.

The second performance metric is the number of lines of code of each implementation. The results are shown in Table 1. The important files are the C++ (.h, .cpp) and kernel (.cl) files as these are what the programmer will need to write. The aspect files (.ah, .cc) are part of the CAPP model and do not need to be re-implemented by the programmer.

4.2 Black-Scholes Option Pricing Problem

This problem requires evaluating option prices using the Black-Scholes model for options pricing and is a good example of an application of the *CAPP* model to a real world problem. The number of options was varied and the runtime was measured for the each implementation, the results are shown in Figure 3. This example illustrates level of performance the *CAPP* model can provide. The *CAPP* and OpenCL runtimes are very similar. The average performance of the *CAPP* implementation was 1.4% slower than the OpenCL implementation, while the C++ implementation was up to 9 times slower than both the *CAPP* and OpenCL implementations.

The comparison of the number of lines of code for each of the implementations is shown in Table 2. The file types of interest are the same as those mentioned in the SAXPY example.

5. EVALUATION AND LIMITATIONS

5.1 Analysis of Results

The results presented in the Section 4 show the similarity in the performance of a *CAPP* implementation and an OpenCL implementation, as well as the performance advantage over a C++, CPU only implementation. The similarity of the *CAPP* and OpenCL performance proves the viability of using the *CAPP* model for parallel programming. The

Table 1: Comparison of the number of lines of code per file type for the SAXPY problem

	CAPP	OpenCL	C++
.h	48	0	38
.cpp	43	152	73
.ah	53	0	0
.cc	78	0	0
.cl	5	5	0
Total	**227**	**157**	**111**

Table 2: Comparison of the number of lines of code per file type for the Black-Scholes option pricing problem

	CAPP	OpenCL	C++
.h	37	0	32
.cpp	55	181	121
.ah	53	0	0
.cc	78	0	0
.cl	73	73	0
Total	**296**	**254**	**153**

number of lines of code comparison provided further justification for using a *CAPP* implementation as the number of lines of C++ code were dramatically reduced. Furthermore, the resultant C++ code is modular and hence easily maintainable, which is typically not the case for OpenCL code. Therefore, the *CAPP* implementation code was more comparable to the C++ code, both structurally and in length. The *CAPP* model thus provides performance comparable to an OpenCL implementation but written in modular C++.

5.2 Advantages

The *CAPP* model provides numerous advantages which should be mentioned.

While the OpenCL functionality is hidden from the programmer by the aspects, this code is still available to the programmer if desired. This is could be beneficial in many scenarios, for example where a team is using the *CAPP* model GPU experts could modify the implementation provided by the *CAPP* model to improve performance but non-GPU experts could still use the C++ interface.

The use of AspectC++ results in the *CAPP* model being capable of providing general Aspect-Oriented Programming. The benefit of this is that large-scale applications could use standard AspectC++ to modularize other cross-cutting concerns from the application, logging for example, and using the *CAPP* functionality to improve the performance of critical areas of the application using standard C++.

The *CAPP* code which performs all the OpenCL components is reliable. A developer would not have to debug the OpenCl setup and memory management code when testing a parallel algorithm. Furthermore, the C++ code is simple. These features results in a much faster parallel programming development cycle and isolate potential sources of error to the kernel function provided by the programmer.

5.3 Limitations

Due to the weaving process occurring before compilation, it was expected that the *CAPP* implementation would provide performance equal to that of the OpenCL implementation. However, the results showed a marginal performance reduction. This was determined to be due to AspectC++ weaver producing code which compiles to a slightly less efficient executable.

Since AspectC++ is still in the development process it does not support all the features of C++, for example generic programming through templates. This places a limitation on the C++ code the programmer may write if they are using the *CAPP* model. The development of the AspectC++ compiler is therefore a limitation of the *CAPP* model as the latest C++ features will not be available. However, future releases of the AspectC++ compiler are scheduled to include support for many more modern C++ features.

6. CONCLUSION

This paper described the *CAPP* model which uses aspects to remove cross-cutting code from the C++ code, thereby providing simpler, more modular parallel capable C++ code. The aspects were implemented using AspectC++, and the parallel functionality was implemented using OpenCL. The cross-cutting components handled by the aspect are the setup of the OpenCL context, the compilation of the kernel, the transfer of memory between the host and device when the kernel is executed, and the kernel execution itself. The aspect defined by the *CAPP* model is abstract and can be used for any parallel programming application. An interface is provided for calling the OpenCL kernel from the C++ code so no cross-cutting OpenCL code is present in the C++ code, and the developer is not required to learn the OpenCL API.

Two example problems were looked at: SAXPY and options pricing. *CAPP*, OpenCL and C++ implementations were compared for each problem. The performance of the *CAPP* model was shown to be similar to that of the OpenCL implementation, with 7 and 1.4% performance reductions, respectively. Structurally the *CAPP* implementation was similar to the C++ implementation in terms of the number of lines and modularity, and better designed that the OpenCL implementation. Both *CAPP* implementations required no OpenCL code in the C++ header or implementation files, therefore the *CAPP* model provides high performance parallel programming using C++.

7. FUTURE WORK

The *CAPP* model proposed in this paper looked at using aspects to hide the complexities and cross-cutting concerns of parallel programming. However, only a minimal amount of the OpenCL API was used to allow OpenCL kernels to be run from the C++ code. The OpenCL API is extensive and provides many additional features which could be included in the abstract aspect. Furthermore, additional, more specific aspects could derive from the abstract aspect but provide other functionality, for example execution timing, performance profiling, or logging.

The slight performance reduction of the *CAPP* implementation over the OpenCL implementation could be reduced by optimising the aspect code.

OpenCL provides functionality which is available for parallel devices from multiple vendors, which may not take advantage of features specific to some vendors - for example dynamic parallelism available to some Nvidia GPUs. Future work would involve extending the *CAPP* aspects to take advantage of these specific hardware features. This would require including the CUDA API into the model. To perform this efficiently templates would need to be used, and hence the next release of the AspectC++ compiler would be required, to minimise runtime evaluation and rather perform the evaluation at compile time.

To provide a parallel programming environment which can be written explicitly in C++, the kernel would need to be written using C++. Future version of the *CAPP* model could provide the option to the programmer to either specify the kernel themselves, or to have the *CAPP* model attempt to parallelize a C++ function into a kernel. It is unlikely that this feature would ever achieve performance levels equivalent to kernels specified by experienced programmers. However, even if only minimal speedup were to be achieved, this would prove useful for applications where any additional performance is needed but where the programmers have no knowledge of parallel programming.

8. REFERENCES

[1] J. Breitbart. CuPP - A framework for easy CUDA integration. *Parallel and Distributed Programming*, IPDPS:1–8, May 2009.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook

for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23:777–786, 2004.

[3] D. Charousset, T. C. Schmidt, and R. Hiesgen. CAF_C++ Actor Framework : An Open Source Implementation of the Actor Model in C++. Available at http://www.actor-framework.org/#about [Accessed 7 May 2015].

[4] D. Charousset, T. C. Schmidt, and R. Hiesgen. CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications. In *Proc. of the 5th ACM SIGPLAN Conf. on Systems Programming and Applications (SPLASH '14) Workshop AGERE!* ACM, October 2014.

[5] A. Gal and W. S.-P. andOlaf Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *Accepted at the Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. OOPSLA, October 2001.

[6] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144, April 2011.

[7] S. V. Group. Clock frequency. Available at http://cpudb.stanford.edu/visualize/clock_frequency [Accessed 13 June 2015].

[8] M. Harris. An Easy Introduction tp CUDA C and C++. NVIDIA CUDA ZONE. Available at http://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/ [Accessed 3 June 2015].

[9] M. Harris. Six Ways to SAXPY. NVIDIA CUDA ZONE. Available at http://devblogs.nvidia.com/parallelforall/six-ways-saxpy/ [Accessed 8 June 2015].

[10] Khronos Group. *OpenCL 1.0 Specification*, Dec. 2008. Rev. 29. https://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, June 1997.

[12] R. Kumar, K. Farkar, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA hetrogeneous multi-core architectures: the potential for processor power reduction. In *Proc. of the 36th International Symposium on Microarchitecture*, pages 81–92. IEEE/ACM, Dec. 2003.

[13] Microsoft. C++ AMP (C++ Accelerated Massive Parallelism). Available at https://msdn.microsoft.com/en-us/library/hh265136.aspx [Accessed 7 May 2015].

[14] Nvidia. *CUDA RUNTIME API*, March. 2015. http://docs.nvidia.com/cuda/index.html#axzz3cGMEdjIx.

[15] M. Pharr and C. Kolb. *Options Pricing on the GPU*. Addison-Wesley Professional, 2005.

[16] RapidMind Inc. *Writing Applications for the GPU Using the RapidMind Development Platform*, 2006. http://www.cs.ucla.edu/~palsberg/course/cs239/papers/rapidmind.pdf.

[17] J. L. Sobral, M. P. Monteiro, and C. A. Cunha. Aspect-oriented support for modular parallel computing. In *High Performance Computing for Computational Science - VECPAR 2006*, volume LNCS 4395, pages 93–106. Springer-Verlag Berlin Heidelberg, 2007.

[18] O. Spinczyk, A. Gal, and W. SchrÃűder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proc. of the 40th International Conference on Technology of Object-Oriented Languages and Systems*. TOOLS, February 2002.

[19] M. Wang and M. Parashar. Object-oriented stream programming using aspects. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11, April 2010.

[20] V. Weinberg, M. Brehm, and I. Christadler. Omi4papps: Optimisation, modelling andÂăimplementation for highly parallel applications. In S. Wagner, M. Steinmetz, A. Bode, and M. M. MÃijller, editors, *High Performance Computing in Science and Engineering, Garching/Munich 2009*, pages 51–62. Springer Berlin Heidelberg, 2010.