Aspect-Oriented Parallel Programming Using AspectC++ and OpenCL

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

ABSTRACT

This paper provides a sample of a LaTeX document which conforms, somewhat loosely, to the formatting guidelines for ACM SIG Proceedings. It is an *alternate* style which produces a *tighter-looking* paper and was designed in response to concerns expressed, by authors, over page-budgets. It complements the document *Author's (Alternate) Guide to Preparing ACM SIG Proceedings Using LaTeX2$_\epsilon$ and BibTeX*. This source file has been written with the intention of being compiled under LaTeX2$_\epsilon$ and BibTeX.

The developers have tried to include every imaginable sort of "bells and whistles", such as a subtitle, footnotes on title, subtitle and authors, as well as in the text, and every optional component (e.g. Acknowledgments, Additional Authors, Appendices), not to mention examples of equations, theorems, tables and figures.

To make best use of this sample document, run it through LaTeX and BibTeX, and compare this source code with the printed output produced by the dvi file. A compiled PDF version is available on the web page to help you with the 'look and feel'.

CCS Concepts

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

Keywords

Aspect; device; host; pointcut; advice

1. INTRODUCTION

In recent years parallel systems become more established as processor core frequencies began to plateau. The result of this was parallel systems which made use multiple core CPUs and many core GPUs. 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. 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, however, due to having fewer cores they provide lower levels of performance increase and require large amounts of power to perform the instruction level parallelism [7]. General-Purpose GPU (GPGPU) programming involves combining CPU's and GPU's 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, namely OpenCL [5] and CUDA [9].

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 on both the CPU and GPU present in the hybrid system, simultaneously. CUDA applies a similar methodology and also provides and 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 when compared to traditional, sequential systems. 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 [**?**], is (The colors after the description relate to the example given in Listing 2)

1. Initialize the *host* data (Green)
2. Setup OpenCL variables, which involves (Red):
   - Setting the OpenCl platform
   - Creating the OpenCL Context
   - Setting the parallel device to use
   - Creating the OpenCL command queue
3. Creating the OpenCL kernel (Blue)

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

DOI: 10.1145/1235

```
#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++

4. Create OpenCL buffers to move data from *host* to *device*(Magenta)
5. Execute kernel on parallel device (Green)
6. Move results back from *device* to *host* (Magenta)

which is a lengthy sequence of events to perform parallel computation. Comparatively, to do the same computation using only the *host*, only two steps would be required, steps 3 and 5 defined above. All other steps increase the complexity of performing parallel programming and cross-cut the intention of the core code (code directly related to the problem) which is the execution of the operations on the data. This is illustrated by a comparison of Listings **??** and 2 which show a vector addition kernel implementation using C++ and OpenCL respectively.

The need for GPGPU programming arises from the computational complexity of the algorithms which results in non-GPGPU systems not being able to compute the algorithm in sufficient time. The additional complexities make parallel development inefficient and require programmers to learn the complex OpenCL and CUDA API's, hence programmers are generally reluctant to take on the learning curve required for parallel programming. However, the computational speed up provided by GPGPU programming is still required and thus the complex, low-level interaction between the *host* and *device* must be overcome to gain access to the increased computational performance.

To remove the requirement of the programmer to deal with this complexity, aspect-oriented programming (AOP) [6] can provide a solution which allows the cross-cutting components to be *woven* into the core computational code at compile time, but being modularised into aspects before compile time, resulting in simple core code which consists only of code directly related to the program intention, and aspects which contain the code which cross-cuts the program's intention.

Aspects are a relatively new concept in computer science and work on aspects on C++ is limited. An AOP implementation for C++ has been available since 2005 in the form of AspectC++ [**?**]. Furthermore the use of aspects in low-level parallel programming is even more limited, however, numerous proposals have been provided which aim to reduce the complexities of low-level parallel programming (summarised in Section 2). This paper presents the Aspect Parallel Programming *APP* model which makes use of aspects, using AspectC++, to hide the above mentioned complexities present in parallel programming using OpenCL, from the programmer.

The aspects perform both the static components - the initialisation of the OpenCL variables - and the 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 *APP* model. This choice was made since the kernel is specific to the implementation of the algorithm and should thus be given to the developer. Converting C++ code to an OpenCL kernel would require building a compiler, which is not the aim of the proposal, or performing runtime conversion, which would result in performance loss. To write a kernel the developer does not need to learn the OpenCL API, but only needs an understanding of how parallel computation it performed in terms of the thread arrangement, which should be known if the programmer would like to parallelize their algorithm.

The *APP* 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 but that also allows direct access to the low-level hardware through aspects - if required by the programmer
- Provides 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 both using aspects and other methods; Section 3 describes the *APP* programming model and the use of AspectC++, and its features, to achieve the above mentioned goals; Section 4 presents the results of implementations of *APP* to two problem domains, SAXPY and Black Scholes option pricing; Section 5 provides a comparison of the *APP* model with both a non-aspect non-parallel C++ solution, and a non-aspect parallel OpenCL solution, in terms of both performance and code complexity and comments on the limitations of the *APP* model; Section 6 concludes and Section **??** discusses 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 and provide low-level access to the parallel hardware. As mentioned in Section 1 this introduces cross-cutting code that 'tangles' the core code. Furthermore, while access is given to the parallel capable hardware, the API's are extensive and have steep learning curves.

APIs, frameworks, and entirely separate languages, which hide the low-level interactions with the parallel capable hardware from the programmer, have been proposed. These systems provide wrappers around the low-level interfaces which often reduces performance, a key consideration in parallel programming, and requires the programmer to learn an additional set of API functions.

```
#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 params
  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),
    (istream_buf_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(inputs[0].size());
  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

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 thus 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 complexity into learning the parallel programming process.

C++ AMP [8] 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 [10] and Brook [2], while are similar to standard C++, essentially define new languages which allow for parallel acceleration.

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 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 same concept as the kernel provided by Nvidia and OpenCL, in the C++ language. RapidMind also makes use of macros, which add complexity to the code, rather simplifying the code - 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 models 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 allow computation to be execution on the device, as well as to move the data to the device memory. This reduces their performance when compared to CUDA or OpenCL implementations, which has resulted in 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 [11], however, Java is used

and the model is based on multi-core CPUs and does not include the numerous complexities which arise from having other computational devices such as GPUs.

A new, parallel aspect language, based on AspectC++, is proposed by [**?**]. Features closely related to AspectC++, but more specific to parallel programming. are defined to hide most of the complexities from the programmer. The result is a large reduction of the cross-cutting components present in low-level parallel programming. The system allows 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++. A compiler is used to weave the defined language into the C++ core code. The performance reduction of only ≈20% was achieved. The aspect model is not yet fully functional and is specific to parallel programming, thus at this stage could not be used to modularize non-parallel cross-cutting concerns into aspects.

The *APP* 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 by the aspects. This results in code code the has no components which cross-cut the code's intention, and thus kernel functions which can be executed from the *host* with a single function call of an instantiated C++ class. (3) There is negligible loss in performance. This is because AspectC++ weaves the aspects into the core code before compilation so that to the compiler the code looks like the tangled code, but to the programmer is 'untangled'.

3. THE ASPECT-ORIENTED PARALLEL PROGRAMMING (APP) MODEL

The *APP* model presented in this section allows the programmer to write parallel programs in traditional C++ with the addition of the kernel function which performs tasks on some data in a parallel manner. The aspects components are written using AspectC++ and provide the functionality of the cross-cutting OpenCL code, hence 'untangling' the core code. To demonstrate the use of the *APP* model, the OpenCL vector addition of Listing 2 will be used as an example.

3.1 Abstract ClContext Aspect

AspectC++ is based on C++ and hence provides functionality for both inheritance and polymorphism. The *APP* 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 if required. Polymorphism is provided by AspectC++ through virtual pointcuts. Virtual pointcuts were used by the abstract *ClContext* aspect to define an interface which allows the derived aspect to specify which C++ core classes require parallel capability. The *CoreClasses* pointcut in Listing 3 shows this. This allows the abstract aspect to provide any general functionality which should be present for a parallel class, without defining which class the aspect should weave the functionality into. Listing 4 shows the derived aspect, which uses the virtual *CoreClasses* to define which of the C++ core classes the OpenCL functionality should be woven into.

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 so that the kernel can be executed on the parallel capable hardware. Many of these variables are required to 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 **??**.

AspectC++ provides the *slice* feature for defining C++ classes within aspects. Furthermore, it allows for *advice* to be defined for the the virtual pointcut and this advice can use C++ inheritance to weave the *slice* class as a base class for C++ core classes, or to derive from C++ core classes. Using these features, the *APP* 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 advice for the virtual pointcut tells the aspect weaver to weave the *slice* class as a base class of the C++ core class requiring parallel capability. Listing 3 shows the *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.

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, create the OpenCL context. They cross-cut the C++ core code since they are not related to the intention of the program. Furthermore, these operations are generic - they are the same for any OpenCL program - they can be defined performed in functions of the *ClInstance slice* class. The operations for creating the kernel (below the blue comments in Listings) have the same problems and hence can also be performed in functions of th *ClInstance slice* class.

The *ClContext* aspect then defines a *ClSetup* pointcut to specify where in the C++ core code these functions should be executed on construction of the C++ core class. To provide the programmer with flexibility when writing the C++ core class teh *ClSetup* advice makes use of the *tjp* pointer AspectC++ provides. The *tjp* pointer provides the aspect with access to the C++ core class into which the aspect will be woven. Using *tjp->that()* in an aspect is equivalent to using *this* within the C++ core class. This feature was used to allow the programmer to specify the computation device (CPU or GPU) and the kernel source file and name as variables of the C++ core class. Using AcpectC++'s *tjp->that()*

```
// 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 aspect which defined the OpenCL variables required for parallel programming

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

Listing 4: Derived aspect defining the core classes which need the OpenCL variables

the aspect can use the values provided by the programmer from the C++ core code to setup the OpenCL environment as required. 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 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. This is done using the cl::Buffer type in OpenCL. These operations are shown in Listing 2 below the magenta comments. This process of memory allocation, movement, and deallocation is not required for a C++ implementation, thus cross-cut's the program's main intention. Furthermore, memory management is a difficult problem in the C and C++ language, which is

```
// 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

exaggerated by the introduction of device memory. These operations are required each time the kernel is executed and result in the amount of cross-cutting code growing linearly with the number of kernels and kernel calls.

These problems are solved through aspects by defining a virtual, *RunKernel*, function in the *ClInstance slice* class. The *runKernel* function provides an interface for the programmer in the C++ core class. Since the *ClInstance slice* class is a baseclass of any C++ core class requiring parallel functionality, the *RunKernl* must be defined in the C++ core class and hence will always be present. By defining the *RunKernel* function as an interface the *ClContext* aspect can define advice to implement the cross-cutting code each time the C++ core class calls the *RunKernel* function.

The *RunKernel* function specifies that the arguments provided from the C++ code code must be vectors which hold the input and output data for the kernel, which are the input and output variables for the computation. This allows the C++ core code to execute a kernel by simply calling the *RunKernel* function and passing the inputs and outputs as function arguments, as shown by Listing **??** The *tjp* pointer is again used, but this time to access the *RunKernel* function arguments and hence the input and output data for the kernel so that the buffers can be managed.

Listing 6 shows the *ManageKernel* pointcut and advice which calls the *ManageBuffers* function in the *CLInstance* class to perform the cross-cutting memory management between the *host* and *device*

4. RESULTS

This section presents the results of application of *APP* model to two GPGPU programming problems. The first problem is essentially the 'hello world' of parallel programming and is known as SAXPY (Single-precision A*X + Y). The second example is that of ... which is more complex and shows the use of the *APP* model to more 'tangled' core code. The result are analysed in terms of both performance and code complexity by terms of the number of lines and

```
// 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 with 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 standard c++

modularity of the resulting code.

4.1 SAXPY Problem

The problem was implemented using the *APP* model, OpenCL, and C++. Various vector sizes were tested to validate that the solution using the *APP* implementation scales similarly to the OpenCL implementation. Table **??** shows the performance results for SAXPY problem.

The number of lines of code for the file types for the different implementations are given in Table 1

5. EVALUATION AND LIMITATIONS

6. CONCLUSION

7. FUTURE WORK

The *proceedings* are the records of a conference. ACM seeks to give these conference by-products a uniform, high-quality appearance. To do this, ACM has some rigid requirements for the format of the proceedings documents: there is a specified format (balanced double columns), a specified set of fonts (Arial or Helvetica and Times Roman) in certain specified sizes (for instance, 9 point for body copy), a specified live area (18 × 23.5 cm [7" × 9.25"]) centered on the page, specified size of margins (1.9 cm [0.75"]) top, (2.54 cm [1"]) bottom and (1.9 cm [.75"]) left and right; specified column width (8.45 cm [3.33"]) and gutter size (.83 cm [.33"]).

The good news is, with only a handful of manual settings[1], the LaTeX document class file handles all of this for you.

The remainder of this document is concerned with showing, in the context of an "actual" document, the LaTeX commands specifically available for denoting the structure of a proceedings paper, rather than with giving rigorous descriptions or explanations of such commands.

8. THE *BODY* OF THE PAPER

Typically, the body of a paper is organized into a hierarchical structure, with numbered or unnumbered headings for sections, subsections, sub-subsections, and even smaller sections. The command `\section` that precedes this paragraph is part of such a hierarchy.[2] LaTeX handles the numbering and placement of these headings for you, when you use the appropriate heading commands around the titles of the headings. If you want a sub-subsection or smaller part to be unnumbered in your output, simply append an asterisk to the command name. Examples of both numbered and unnumbered headings will appear throughout the balance of this sample document.

Because the entire article is contained in the **document** environment, you can indicate the start of a new paragraph with a blank line in your input file; that is why this sentence forms a separate paragraph.

8.1 Type Changes and *Special* Characters

We have already seen several typeface changes in this sample. You can indicate italicized words or phrases in your text with the command `\textit`; emboldening with the command `\textbf` and typewriter-style (for instance, for com-

[1]Two of these, the `\numberofauthors` and `\alignauthor` commands, you have already used; another, `\balancecolumns`, will be used in your very last run of LaTeX to ensure balanced column heights on the last page.

[2]This is the second footnote. It starts a series of three footnotes that add nothing informational, but just give an idea of how footnotes work and look. It is a wordy one, just so you see how a longish one plays out.

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

	APP	OpenCL	C++
.h			
.cpp			
.ah			
.cc			
.cl			
Total			

puter code) with `\texttt`. But remember, you do not have to indicate typestyle changes when such changes are part of the *structural* elements of your article; for instance, the heading of this subsection will be in a sans serif[3] typeface, but that is handled by the document class file. Take care with the use of[4] the curly braces in typeface changes; they mark the beginning and end of the text that is to be in the different typeface.

You can use whatever symbols, accented characters, or non-English characters you need anywhere in your document; you can find a complete list of what is available in the *LaTeX User's Guide*[?].

8.2 Math Equations

You may want to display math equations in three distinct styles: inline, numbered or non-numbered display. Each of the three are discussed in the next sections.

8.2.1 Inline (In-text) Equations

A formula that appears in the running text is called an inline or in-text formula. It is produced by the **math** environment, which can be invoked with the usual `\begin. . .\end` construction or with the short form `$. . .$`. You can use any of the symbols and structures, from α to ω, available in LaTeX[?]; this section will simply show a few examples of in-text equations in context. Notice how this equation: $\lim_{n\to\infty} x = 0$, set here in in-line math style, looks slightly different when set in display style. (See next section).

8.2.2 Display Equations

A numbered display equation – one set off by vertical space from the text and centered horizontally – is produced by the **equation** environment. An unnumbered display equation is produced by the **displaymath** environment.

Again, in either environment, you can use any of the symbols and structures available in LaTeX; this section will just give a couple of examples of display equations in context. First, consider the equation, shown as an inline equation above:

$$\lim_{n\to\infty} x = 0 \tag{1}$$

Notice how it is formatted somewhat differently in the **displaymath** environment. Now, we'll enter an unnumbered equation:

$$\sum_{i=0}^{\infty} x + 1$$

and follow it with another numbered equation:

$$\sum_{i=0}^{\infty} x_i = \int_0^{\pi+2} f \tag{2}$$

just to demonstrate LaTeX's able handling of numbering.

8.3 Citations

Citations to articles [?, ?, ?, ?], conference proceedings [?] or books [?, ?] listed in the Bibliography section of your article will occur throughout the text of your article. You should use BibTeX to automatically produce this bibliography; you simply need to insert one of several citation commands with a key of the item cited in the proper location in the `.tex` file [?]. The key is a short reference you invent to uniquely identify each work; in this sample document, the key is the first author's surname and a word from the title. This identifying key is included with each item in the `.bib` file for your article.

The details of the construction of the `.bib` file are beyond the scope of this sample document, but more information can be found in the *Author's Guide*, and exhaustive details in the *LaTeX User's Guide*[?].

This article shows only the plainest form of the citation command, using `\cite`. This is what is stipulated in the SIGS style specifications. No other citation format is endorsed or supported.

[3]A third footnote, here. Let's make this a rather short one to see how it looks.

[4]A fourth, and last, footnote.

Table 2: Frequency of Special Characters

Non-English or Math	Frequency	Comments
Ø	1 in 1,000	For Swedish names
π	1 in 5	Common in math
\$	4 in 5	Used in business
Ψ_1^2	1 in 40,000	Unexplained usage

8.4 Tables

Because tables cannot be split across pages, the best placement for them is typically the top of the page nearest their initial cite. To ensure this proper "floating" placement of tables, use the environment **table** to enclose the table's contents and the table caption. The contents of the table itself must go in the **tabular** environment, to be aligned properly in rows and columns, with the desired horizontal and vertical rules. Again, detailed instructions on **tabular** material is found in the *LaTeX User's Guide*.

Immediately following this sentence is the point at which Table 1 is included in the input file; compare the placement of the table here with the table in the printed dvi output of this document.

To set a wider table, which takes up the whole width of the page's live area, use the environment **table*** to enclose the table's contents and the table caption. As with a single-column table, this wide table will "float" to a location deemed more desirable. Immediately following this sentence is the point at which Table 2 is included in the input file; again, it is instructive to compare the placement of the table here with the table in the printed dvi output of this document.

8.5 Figures

Like tables, figures cannot be split across pages; the best placement for them is typically the top or the bottom of the page nearest their initial cite. To ensure this proper "floating" placement of figures, use the environment **figure** to enclose the figure and its caption.

This sample document contains examples of **.eps** files to be displayable with LaTeX. If you work with pdfLaTeX, use files in the **.pdf** format. Note that most modern TeX system will convert **.eps** to **.pdf** for you on the fly. More details on each of these is found in the *t*he fully functional application. Figure **??** shows the process of weaving the aspect and core component code to produce the final application.

As was the case with tables, you may want a figure that spans two columns. To do this, and still to ensure proper

Table 3: Some Typical Commands

Command	A Number	Comments
`\alignauthor`	100	Author alignment
`\numberofauthors`	200	Author enumeration
`\table`	300	For tables
`\table*`	400	For wider tables

Figure 1: A sample black and white graphic.

Figure 2: A sample black and white graphic that has been resized with the `includegraphics` command.

"floating" placement of tables, use the environment **figure*** to enclose the figure and its caption. and don't forget to end the environment with figure*, not figure!

8.6 Theorem-like Constructs

Other common constructs that may occur in your article are the forms for logical constructs like theorems, axioms, corollaries and proofs. There are two forms, one produced by the command `\newtheorem` and the other by the command `\newdef`; perhaps the clearest and easiest way to distinguish them is to compare the two in the output of this sample document:

This uses the **theorem** environment, created by the `\newtheorem` command:

THEOREM 1. *Let f be continuous on $[a, b]$. If G is an antiderivative for f on $[a, b]$, then*

$$\int_a^b f(t)dt = G(b) - G(a).$$

The other uses the **definition** environment, created by the `\newdef` command:

Definition 1. If z is irrational, then by e^z we mean the unique number which has logarithm z:

$$\log e^z = z$$

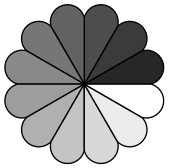

Figure 4: A sample black and white graphic that has been resized with the `includegraphics` command.

Two lists of constructs that use one of these forms is given in the *Author's Guidelines*.

There is one other similar construct environment, which is already set up for you; i.e. you must *not* use a `\newdef` command to create it: the **proof** environment. Here is a example of its use:

PROOF. Suppose on the contrary there exists a real number L such that

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = L.$$

Then

$$l = \lim_{x\to c} f(x) = \lim_{x\to c} \left[gx \cdot \frac{f(x)}{g(x)}\right] = \lim_{x\to c} g(x) \cdot \lim_{x\to c} \frac{f(x)}{g(x)} = 0 \cdot L = 0,$$

which contradicts our assumption that $l \neq 0$. □

Complete rules about using these environments and using the two different creation commands are in the *Author's Guide*; please consult it for more detailed instructions. If you need to use another construct, not listed therein, which you want to have the same formatting as the Theorem or the Definition[**?**] shown above, use the `\newtheorem` or the `\newdef` command, respectively, to create it.

A *Caveat* for the TEX Expert

Because you have just been given permission to use the `\newdef` command to create a new form, you might think you can use TEX's `\def` to create a new command: *Please refrain from doing this!* Remember that your LATEX source code is primarily intended to create camera-ready copy, but may be converted to other forms – e.g. HTML. If you inadvertently omit some or all of the `\def`s recompilation will be, to say the least, problematic.

9. CONCLUSIONS

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow. There is still the Bibliography to deal with; and we will make a disclaimer about that here: with the exception of the reference to the LATEX book, the citations in this paper are to articles which have nothing to do with the present subject and are used as examples only.

10. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

11. REFERENCES

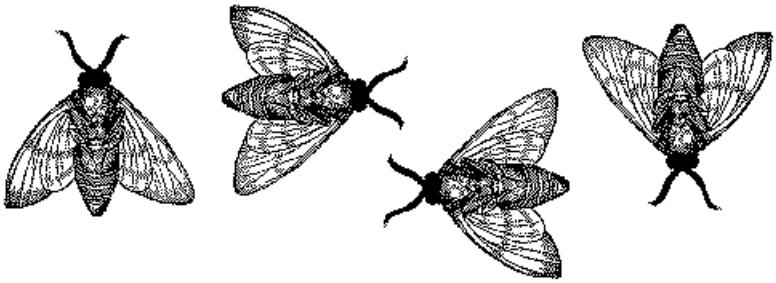

Figure 3: A sample black and white graphic that needs to span two columns of text.

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

[6] 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.

[7] 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.

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

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

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

[11] 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.

APPENDIX

A. HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the **appendix** environment, the command **section** is used to indicate the start of each Appendix, with alphabetic order designation (i.e. the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure *within* an Appendix, start with **subsection** as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction

A.2 The Body of the Paper

A.2.1 Type Changes and Special Characters

A.2.2 Math Equations

Inline (In-text) Equations.

Display Equations.

A.2.3 Citations

A.2.4 Tables

A.2.5 Figures

A.2.6 Theorem-like Constructs

A Caveat for the TEX Expert

A.3 Conclusions

A.4 Acknowledgments

A.5 Additional Authors

This section is inserted by LaTeX; you do not insert it. You just add the names and information in the `\addition-alauthors` command at the start of the document.

A.6 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the

.bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

B. MORE HELP FOR THE HARDY

The sig-alternate.cls file itself is chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of LaTeX, you may find reading it useful but please remember not to change it.