

# 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X<sub>2 $\epsilon$</sub>  and BibT<sub>E</sub>X*. This source file has been written with the intention of being compiled under L<sup>A</sup>T<sub>E</sub>X<sub>2 $\epsilon$</sub>  and BibT<sub>E</sub>X.

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 L<sup>A</sup>T<sub>E</sub>X and BibT<sub>E</sub>X, 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 multicore systems such as CPU's with numerous cores and GPU's with many cores, as well as hybrid systems which make use of both numerous core CPU's and

many core GPU's. The cores used for CPU's and GPU's differ in complexity. GPU's use many (up to thousands), simple cores to increase computational efficiency. CPU's use fewer (1 - 20), complex cores that allow for instruction level parallelism, however, as this complexity comes from the vast number of transistors that make up the processor which leads to a large amount of power being required by the processor [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 cores to pass data to the GPU cores which perform the computation in a parallel manner. MAYBE INCLUDE FIGURE.

For GPGPU programming there are two options 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 hardware vendors, namely Nvidia, AMD and Intel. This wide range of support is advantageous in the sense that a parallel implementation using the OpenCL API could run on both the CPU and GPU cores present in the hybrid system, simultaneously. CUDA applies a similar methodology and also provides an API for writing programs that will run on parallel capable hardware, however, it is specific to Nvidia hardware and hence parallel kernels cannot be run on CPU's or GPU's from any other hardware vendor.

Admittedly, the latest version of the CUDA SDK does provide more advanced functionality as it supports C++11 features whereas OpenCL is based on C, but does provide C++ bindings. Due to CUDA only supporting Nvidia hardware, and the growing acceptance of OpenCL as a standard API, OpenCL was chosen for the implementation.

While these hybrid parallel systems follow Moore's law, they are more difficult to program when compared to traditional, sequential systems. There are numerous factors that cause the increase in difficulty. The main cause is the two different hardware types in the hybrid system, the CPU (from here on referred to as the *host* as it is termed in the GPGPU programming model) and the GPU (from here on referred to as the *device* as it is termed in the GPGPU model). This requires, firstly, the setup of the parallel context which enables parallel kernels to be executed on the device but from the host. Secondly, memory transfer between the host and device as the computational data must be moved from the device memory to the host memory before computation and then from device memory back to host memory after the computation. Thirdly, while this is not strictly necessary for performing implementing a par-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2015 ACM. ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

**Figure 1: Process of weaving aspect and core code.**

allel program it is, however, advantageous - knowledge of the memory structure of the device. GPU's often have complex memory heirachy's which provide memory as varying distances from the computational cores to increase computational performance.

The need for GPGPU programming arises from the computational complexity of the algorithm needing to be solved. The above mentioned problems are not related to the core computation and thus introduce overhead. The overhead is distracting and crosses the problem domain, 'tangling' the resultant code. Furthermore, these cross-cutting code is required in multiple places when the parallel implementation of the algorithm requires numerous kernels. Aspect-oriented programming (AOP) [6] can provide a solution to this which allows the cross-cutting components to be *woven* into the core computational code, resulting in code comprising only of components directly related to the core computation.

The AOP paradigm allows code components which do not lend themselves to an object-oriented solution to be modularized using aspects. The code components which have been modularized into aspects are then 'woven' into the core component code through the aspect compiler. This process allows the programmer to separate the core component code, which is written using any normal programming language, and the aspect code, which is written in the aspect language. Before being woven, the core component code consists solely of code related specifically to the problem - the algorithm in the case of parallel programming - and the aspect code contains all the elements that 'tangle' the core component code when present. After being woven, the cross-cutting aspects are combined with the core code to produce the fully functional application. Figure 1 shows the process of weaving the aspect and core component code to produce the final application.

This paper describes the use of aspects, using AspectC++, to hide the above mentioned complexities present in parallel programming from the programmer. The resultant aspect parallel model has two main aims:

- To allow for a parallel implementation which is comparatively simple 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 aspects in *AspectCL* deal with: Firstly, the setup of the OpenCL context that allows the host to execute the parallel kernels on the device, this included the management of the potentially numerous devices in the system. Secondly, The allocation and deallocation of memory on the device when kernels are executed, and thirdly, the execution of kernels when the host code requires the parallel computation to be executed.

The implemented aspects do not deal with the kernel itself and the type of memory that used on the device, i.e the aspects do not determine if the kernel should use global or shared memory. This choice was made as it was identified that the kernel directly related to the problem domain and

the freedom to implement the kernel and define the grid size for the kernel execution should be given to the programmer.

The solution allows the programmer to not have to learn the complex CUDA or OpenCL API's as the aspects hide these complexities from the programmer. However, as the kernels are the responsibility of the programmer, the programmer will require an understanding of how parallel hardware executes a kernel and how the indexes of the kernel grid relate to the computation.

The rest of the report is structured as follows: Section 2 reviews work related to the use of aspects in a parallel context; Section 3 describes the *APP* programming model and the use of aspects with AspectC++, and its features, in *APP*; Section 4 presents the results of implementations of *APP* to two problem domains, SAXPY and (TBD); Section 5 provides a comparison of the *APP* with that of 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 parallel aspect solution; 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 both 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 large and hence slow down the development process.

Attempts have been made to provide APIs, frameworks, and entirely separate languages that hide the low-level interactions with the parallel capable hardware from the programmer, however, these systems provide wrappers around the low-level interfaces which reduces performance which is a key consideration with parallel implementations.

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.

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

C++ AMP [8] provides extensions to the C++ language that allows data parallel capable hardware to be used to accelerate computation. C++ AMP, however, 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 are similar to CUDA and OpenCL.

RapidMind provides an environment which allows for parallel programming by taking advantage of C++ template metaprogramming to provide 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 that can operate on *Array* data types in a parallel manner. The *Program* data type is same concept as the kernel provided by Nvidia and OpenCL. It allows the programmer to define a sequence of operations on the *Array* data types, in the C++ language. It also makes use of macros, which add clutter to the code, opposite to the intention of aspects.

Brook provides high-level abstractions which hide the low-level complexities present when interfacing directly with the parallel capable hardware. Similar to RapidMind there are three abstractions that allow for 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 model 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. This 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 CPU's and does not include the numerous complexities which arise from having other computational devices such as GPU's.

The *APP* model provides a few advantages over the previously mentioned features. Firstly 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. Secondly, low-level complexities are hidden from programmers by the aspects which makes reduces the complexity of the code required for writing a parallel program since all cross-cutting components are removed from the core code. Thus kernel functions can be executed from the *host* with a single function call rather than the numerous lines of code required by CUDA or OpenCL implementations. Thirdly, the devices to be used are allowed to be selected by the developer but are managed by the aspects, again reducing the complexity of the core code. Fourthly, performance is not sacrificed as the aspects are woven into the core code at compile time essentially resulting in a pure OpenCL implementation after compilation.

### 3. THE 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 vector addition example of Listing ?? 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 this to define an abstract aspect, the *ClContext* aspect, which provides OpenCL components which will be present in all programs which require parallel capability. 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 can then be defined in the derived aspect, if required. Polymorphism is provided by AspectC++ through virtual pointcuts, where were used by the abstract to define an interface for the C++ core classes for calling the OpenCL kernel. This is explained in Section ??

#### 3.2 OpenCL Variable Introduction

Observing Listing ?? there numerous OpenCL specific variables that are required to setup the OpenCL environment so that the kernel can be executed on the parallel capable hardware. These variables are required each time a new kernel needs to be executed, which is a lot of overhead when compared to the C++ version of Listing ??.

Using the abstract (*ClContext* aspect), a class can be defined within the aspect which holds the OpenCL variables required for setting up the OpenCL context. The class can be introduced into any C++ core class through the use of AspectC++'s *slice* and *advice* features. The advice sets the class to be a base class of the core class requiring the OpenCL variables. The aspect is made abstract by defining a pure virtual *pointcut* in the aspect which is then defined in a derived aspect to specify exactly which classes in the core code the aspect class should be introduced into as a baseclass.

Listing 1 shows the baseclass, pointcut and advice components of the abstract *ClContext* aspect which allow this. Listing 2 shows the derived *VectAdd* aspect which defines the classes in the core code (in this case the *VectorAddCpp* class which would be present in the host code) which the OpenCL variables should be woven into. When the host code is executed the *VectorAddCpp* class will have all the OpenCL variables defined in the *ClInstance* as public variables.

```

aspect ClContext {
    class ClInstance {
    public:
        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
    };
    pointcut virtual CoreClasses() = 0;

```

```

    advice CoreClasses() : baseclass(CIInstance);
    // Other aspect components
};

```

**Listing 1: Abstract aspect which defined the OpenCL variables required for parallel programming**

```

aspect VectAdd : CIContext {
    pointcut CoreClasses() = "VectorAddCpp";
};

```

**Listing 2: Derived aspect defining the core classes which need the OpenCL variables**

### 3.3 OpenCL Setup Function Introduction

Again observing Listing ??, a large amount of the cross-cutting code comes from the operations on the OpenCL variables. These operations are required to determine the available devices and platforms, create the OpenCL context, and build the kernel from source, and need to be executed numerous times if there are many kernels in the program further tangling the code.

Since these operations are generic - they are the same for any OpenCL program, they can be defined in a functions of the *CIInstance* class and hence will be available to the *CIContext* aspect.

The *CIContext* aspect then defines a *CISetup* pointcut to specify that these functions should be executed on the construction of the C++ core classes (*VectorAddCpp* in the Listings). The aspect weaver will then weave these functions into the C++ core class during compilation. When the program executes, the OpenCL variables in the C++ core class have been set to the necessary values that allow for parallel computation. For example the *kernel* variable has the compiled kernel which can be executed.

Listing 3 shows the definition of the OpenCL setup functions in the baseclass within the aspect, as well as the pointcut and advice for the pointcut which allows the OpenCL setup functions to be run on construction of the C++ core class.

```

aspect CIContext {
    class CIInstance {
        // OpenCl variables
    private:
        void SetupOpenCl(string devType, string
            kSource);
        void SetupKernel(string kSource, string
            kName);
    };
    // Other pointcuts and advice
    pointcut CSetup = construction(CoreClasses());
    advice CSetup() : around() {
        tjp->that()->SetupOpenCl(tjp->that()->devType,
            tjp->that()->kSource);
        tjp->that()->SetupKernel(tjp->that()->
            kSource, tjp->that()->kName);
        tjp->proceed();
    }
};

```

**Listing 3: Abstract aspect with the pointcut and advice for OpenCL setup**

### 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 ?? below blue comments. They are not related to the program intention of performing a parallel computation and hence cross-cut the core code. Furthermore, these operations are required each time the kernel is executed. When using OpenCL the number of lines of code grows quickly with the number of kernels in the program.

This problem is solved by the *clContext* aspect by defining a pure virtual function in the *CIInstance* baseclass which defines the interface for the *RunKernel* function. Any core component classes must implement the *RunKernel* function since they will be derived from the *CIInstance* baseclass. By defining the interface the aspect knows the structure of the *RunKernel* function and can thus access the function arguments through the *tjp* pointer AspectC++ provides.

The virtual *RunKernel* function specifies that the arguments must be vectors, which are the input and output variables for the computation. This allows the C++ core class to pass the input and output data as vectors to the *RunKernel* function which will then be used by the *CIContext* aspect to create the OpenCL buffers, move the data from the *host* memory to the *device* memory, execute the kernel, move the data back to the *host* memory and into the output vector provided to the *RunKernel* function.

Listing 4 shows the structure of the *CIContext* aspect with the pointcut and advice for the managing the OpenCL buffers and running the kernel. Listing ?? shows the C++ core class calling the *RunKernel* function from the host.

```

#define T float
aspect CLContext {
    class CIInstance {
    public:
        virtual void RunKernel(
            vector<vector<T>>& inputs
            ,
            vector<vector<T>>& outputs) =
                0;
    private:
        void ManageClBuffers(
            vector<vector<T>>* inputs,
            vector<vector<T>>* outptus);
    };
    pointcut ManageKernel() =
        execution("%_...::RunKernel(...)") &&
        within(CoreClasses());

    advice ManageKernel() : before() {
        tjp->that()->ManageClBuffers(tjp->arg<0>(),
            tjp->arg<1>());
    }
};

```

**Listing 4: Abstract aspect components with hide kernel cross-cutting concerns**

```

#define T float
int main(int argc, char** argv) {
    // Create input and output vectors
    vector<vector<T>> inputs;
    vector<vector<T>> outputs;
    // Fill vectors with data

    // Aspect parallel class instantiation
    VectorAddCpp parallelVectorAdd;

    // Execute kernel using c++
}

```

```
parallelVectorAdd.RunKernel(inputs, outputs);
}
```

**Listing 5: C++ core class executing a pallel kernel using standard c++**

## 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 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<sup>1</sup>,

<sup>1</sup>Two of these, the `\numberofauthors` and `\alignauthor` commands, you have already used; another, `\balancecolumns`, will be used in your very last run of L<sup>A</sup>T<sub>E</sub>X to ensure balanced column heights on the last page.

**Table 1: Comparison of the number of lines of code per file type for the SAXPY problem (Not including comments)**

	APP	OpenCL	C++
.h			
.cpp			
.ah			
.cc			
.cl			
<b>Total</b>			

the L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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.<sup>2</sup> L<sup>A</sup>T<sub>E</sub>X 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 computer 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<sup>3</sup> typeface, but that is handled by the document class file. Take care with the use of<sup>4</sup> 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 *L<sup>A</sup>T<sub>E</sub>X 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  $\alpha$  to  $\omega$ , available in L<sup>A</sup>T<sub>E</sub>X[?]; this section will simply show a few examples of in-text equations in context. Notice how this equation:

<sup>2</sup>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.

<sup>3</sup>A third footnote, here. Let’s make this a rather short one to see how it looks.

<sup>4</sup>A fourth, and last, footnote.

$\lim_{n \rightarrow \infty} x = 0$ , 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  $\text{\LaTeX}$ ; this section will just give a couple of examples of display equations in context. First, consider the equation, shown as an inline equation above:

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

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  $\text{\LaTeX}$ '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.

### 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.

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 <sub>1</sub> <sup>2</sup>	1 in 40,000	Unexplained usage



Figure 2: A sample black and white graphic.

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  $\text{\LaTeX}$ . If you work with pdf $\text{\LaTeX}$ , use files in the .pdf format. Note that most modern  $\text{\TeX}$  system will convert .eps to .pdf for you on the fly. More details on each of these is found in the the fully functional application. Figure 1 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 "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



Figure 3: A sample black and white graphic that has been resized with the **includegraphics** command.

Table 3: Some Typical Commands

Command	A Number	Comments
<code>\alignauthor</code>	100	Author alignment
<code>\numberofauthors</code>	200	Author enumeration
<code>\table</code>	300	For tables
<code>\table*</code>	400	For wider tables

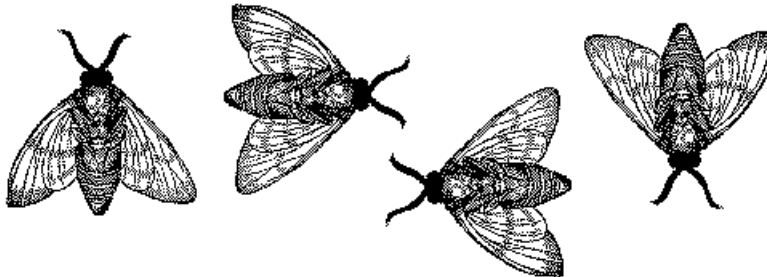


Figure 4: A sample black and white graphic that needs to span two columns of text.

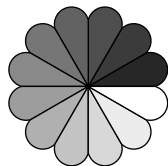


Figure 5: A sample black and white graphic that has been resized with the `includegraphics` command.

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  $z$ :

$$\log e^z = z$$

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 an example of its use:

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

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

Then

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

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

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 T<sub>E</sub>X Expert

Because you have just been given permission to use the `\newdef` command to create a new form, you might think you can use T<sub>E</sub>X's `\def` to create a new command: *Please refrain from doing this!* Remember that your L<sup>A</sup>T<sub>E</sub>X 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 `\defs` 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 L<sup>A</sup>T<sub>E</sub>X 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

- [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 heterogeneous 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#axzz3cGMEdjLx>.
- [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 T<sub>E</sub>X Expert*

### A.3 Conclusions

### A.4 Acknowledgments

### A.5 Additional Authors

This section is inserted by L<sup>A</sup>T<sub>E</sub>X; you do not insert it. You just add the names and information in the `\additionalauthors` 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 L<sup>A</sup>T<sub>E</sub>X, you may find reading it useful but please remember not to change it.