GPU DECLARATIVE FRAMEWORK

by

ROBERT W. SENSER, JR.

B.S., University of Wyoming, 1973

M.A., University of Hawaii, 1975

M.S., University of Hawaii, 1975

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Computer Science and Information Systems

2014

This thesis for the Doctor of Philosophy degree by

Robert W. Senser, Jr.

has been approved for the

Computer Science and Information Systems Program

by

Gita Alaghband, Chair

Tom Altman, Advisor

Michael Mannino

Boris Stilman

Tam N. Vu

November 7, 2014

Senser, Robert W., Jr. (Ph.D., Computer Science and Information Systems)

GPU Declarative Framework

Thesis directed by Professor Tom Altman.

ABSTRACT

This dissertation presents our novel declarative framework, called the *Declarative Framework for GPUs* (DEFG). GPUs are highly sophisticated computing devices, capable of computing at very high speeds. The framework makes the development of OpenCL-based GPU applications less complex, and less time consuming. The framework's approach is two-fold. First, we developed the DEFG domain-specific language in such a way that it uses primarily declarative statements, and design patterns, to define the CPU actions to manage GPU kernels. This approach makes the GPU processing power more accessible. It does this by lowering the amount of specialized GPU knowledge the GPU software developer must possess. It also decreases the volume of code written, meaning these applications can be written more easily and quickly. The second aspect of our approach is the addition of high-value GPU capabilities, most notably the declarative utilization of additional GPU devices. The use of multiple devices, in this DEFG manner, allows for scaling the application run-time performance without rewriting the entire application. This aspect of DEFG makes developing multiple-GPU applications faster and more straightforward.

In this dissertation, we describe DEFG's novel parser, optimizer and code generator, as well as, provide detailed descriptions of DEFG's domain specific language and associated design patterns. Taken together, these components make it possible for the developer to write DEFG source and generate C/C++ programs containing highly optimized OpenCL requests that provide high-performance.

In order to demonstrate the viability of DEFG, we produce applications related

to common areas of computer science: image filtering, graph processing, sorting, and numerical algebra, specifically iterative matrix inversion. We select these applications because each one explores different aspects of GPU use and OpenCL. Taken together, they demonstrate DEFG's ability to function well over a wide range of applications. To show DEFG's capabilities in image processing, we implement the Sobel operator and median image filter applications, with an emphasis on multiple-GPU processing. Our graph-processing, breadth-first search application shows DEFG's ability to process large irregular data structures, with multiple GPUs. In the sorting realm, the novel roughly sorting application shows DEFG's GPU-based sorting capability. Finally, the numerical algebra application, an interesting iterative matrix inversion implementation, exhibits DEFG's ability to implement iterative, GPU-based, numerical processing.

DEFG's domain specific language is able to use mainly declarations to describe the CPU actions needed to manage complex GPU actions. We demonstrate that this approach to GPU-oriented software development succeeds through the production of our OpenCL applications.

The form and content of this abstract is approved. I recommend its publication.

Approved: Tom Altman

This work is dedicated to the memory of my parents, Robert W. Senser and Maria E. Senser, and my younger brother, Dwight W. Senser, Ph.D., whose precious life was lost to a fool with a handgun. *You three Sensers showed me the way.*

ACKNOWLEDGEMENTS

Getting this work to the "finish line" was not an easy task. Two people's extensive support has been beyond measure:

To my intrepid advisor, Tom Altman, my deepest *thank you*. You brought out my best, tolerated my worst, and made it possible for me to run with my passions.

Most of all, to my wonderful wife, Linda Senser, who supported me through the worst moments, who laughed with me through the nuttiest, and who never gave up (even when I wanted to): ***danke sehr***.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

GLOSSARY

BFSDP2GPU Multiple-GPU, breadth-first search DEFG application. 30

CUDA NVIDIA-provided capability to execute C/C++ on NVIDIA GPUs. 2

DEFG Framework to declaratively create GPU applications. 3

GPU Acronym for Graphics Processing Unit. 1

IMIFLX Iterative matrix inversion DEFG application. 110

LVI Acronym for Large Very Irregular, applied to graphs. 65

MEDIAN Median image filter DEFG application. 54

OpenCL Kronos Group specification to execute C/C++ on GPUs. 2

PRAM Acronym for the abstraction Parallel Random Access Machine. 6

RSORT Roughly sorting DEFG application. 33

SIMD Acronym for Single Instruction, Multiple Data. 9

SIMT Acronym for Single Instruction, Multiple Thread. 8

SISD Acronym for Single Instruction, Single Data. 67

SOBEL Sobel image filter DEFG application. 19

CHAPTER I

INTRODUCTION

1.1 GPU Software Development

The graphics processing unit (GPU) is a hardware component having the potential to rapidly execute computer algorithms and code. The raw double-precision floating point throughput of GPUs now exceeds two tera floating-point operations per second (TFLOPS). One GPU, the AMD Radeon HD 7990, can perform 8.2 single-precision TFLOPS and 2.04 double-precision TFLOPS [3]. This computational capability comes at an attractive cost of $1K retail, and has made GPUs very attractive for executing non-graphic applications. The high throughput provided by GPUs has been used in the *high performance computing* (HPC) scientific community to develop many types of applications.

However, while the GPU hardware costs are relatively low, GPU software development costs can be prohibitively high. The software development process associated with GPUs can be complex and more costly than standard software development because of the architecture of GPU hardware and the specialized software development environments needed for GPU programming. Achieving high performance in a GPU environment requires the developer to understand not only the application requirements and parallel software, but also the additional unique characteristics of the GPU hardware. The unique characteristics add complexity, and many pertinent low-level details, to the development environment.

This dissertation presents a novel OpenCL programming framework, which is designed to lower the software development complexity, and thus, the costs of HPC GPU usage. Our approach to simplifying OpenCL programming is two-fold: first, we make use of declarative statements and design patterns to define the CPU-side of a GPU application, thereby lowering the number of lines of code written, and lessening the low-level GPU knowledge needed by the developer.

Second, we add options such as the utilization of multiple GPU devices to scale the application run-time performance without the need to rewrite the application. For certain application types, computing power can be added, in the form of additional GPU devices, without adding significant software development costs.

1.2 Motivation

Developing software for use in high performance computing is often a difficult undertaking. It not only requires a thorough understanding of the application problem being solved and the algorithms used to solve the problem. It also requires an in-depth understanding of the unique characteristics of the hardware platform being utilized. When the platform is parallel in nature, the software becomes even more difficult to write due to the added complexities of parallel execution [47]. The HPC use of GPUs for general processing fits into this latter category of especially difficult software development.

The GPU has been shown to have very high throughput capabilities [71]. Ideally, existing HPC software would be moved to the GPU, and the GPU's high throughput would be immediately available. And at first glance, this appears to be possible because both of the common GPU programming environments, NVIDIA's Compute Unified Device Architecture (CUDA) and Kronos Group's OpenCL Specification, provide the capability to execute C/C++ code as GPU kernels [65, 70]. Unfortunately, GPU software produced this way tends to have very poor performance characteristics.

High performance GPU programming requires the use of specialized, parallel algorithms and GPU-specific, low-level application programming interfaces (APIs). This use, in turn, requires that the developer possess a thorough understanding of the overall GPU hardware architecture. For example, the developer must avoid the major issues of memory latency and instruction path divergence, if he or she wants to obtain high levels of GPU performance. The result is that GPU software development tends to be both complex and time consuming [34].

In this dissertation, we present a novel declarative framework, called the *Declarative Framework for GPUs* (DEFG), which makes development of OpenCL-based GPU applications less complex, and less time consuming [82, 80, 81]. It mitigates the need for a deep understanding of the full CPU-side API used with technologies, such as OpenCL. DEFG allows the developer to focus on the algorithms being used and the most efficient usage of the overall GPU architecture.

In addition, to clearly show DEFG's viability, we demonstrate its use and performance with four diverse, general GPU applications. Each application puts different demands on the framework, thereby showing the framework's applicability, flexibility, and general robustness. Here *robustness* refers to the framework's ability to elegantly handle differing applications' demands and requirements. For certain applications, the DEFG approach makes it possible to scale the application to multiple GPU cards without application changes. This application scaling is made possible by declaring the nature of the application and developing the application GPU kernels, then having the framework generate the code to interconnect the CPU and GPU(s) in a high-performance manner.

1.3 Contributions

In addition to the construction of the novel DEFG framework, our research contributes two groups of OpenCL applications. The first group consists of three existing OpenCL

applications that were converted to DEFG. These DEFG conversions showed the run-time performance of DEFG matching, or exceeding, that of the native OpenCL applications. And, this level of performance was achieved with the software developer writing fewer lines of code, relative to the corresponding, original OpenCL application.

The second group contains four new OpenCL applications, which are used as DEFG use-cases. These latter applications demonstrate the applicability of DEFG in diverse domains, ranging from graph processing to sorting. Each of these applications is measured and analyzed. In some cases, the analysis includes comparisons, in terms of run-time performance and other metrics, between DEFG and non-DEFG application versions.

In summary, this dissertation makes the following contributions to computer science:

- The design, implementation, testing, and analysis of our novel *Declarative Framework for GPUs* (DEFG).

- Application: *Sobel and Median image filtering using multiple GPUs*
  Provides DEFG implementation, measurement, and analysis [86, 87].

- Application: *Breadth-first search application using multiple GPUs*
  Provides GPU algorithmic design, DEFG implementation, measurement, and analysis [38, 59].

- Application: *Sorting roughly (partially) sorted data*
  Provides GPU algorithmic design, DEFG implementation, measurement, and analysis [9, 10].

- Application: *Altman's iterative method of matrix inversion*
  Provides GPU algorithmic design, DEFG implementation, measurement and analysis [7].

1.4 Delimitations

The two most common GPU programming environments are the Kronos Group's OpenCL Specification and NVIDIA's CUDA. CUDA is limited to only NVIDIA products, whereas OpenCL is supported by many hardware vendors and can be run on CPUs and other devices [65, 70]. Due to OpenCL's wider applicability and higher level of API flexibility, our work focuses on OpenCL.

This dissertation covers a wide range of interesting application areas, algorithms, and GPU-related topics such as GPGPU, OpenCL, and the DEFG domain-specific language. However, there are also some related topics that are specifically omitted. In particular, this work does not focus on highly specialized, GPU-generation-specific or product-unique, algorithms and techniques. The GPU technology is constantly changing and overly-focused techniques, helpful for a specific family or generation of GPUs, may soon be obsolete.

This "constantly changing" characteristic can be observed with the latest generations of GPU cards, such as the NVIDIA Fermi and Keplar GPUs. They contain L1 and L2 memory caches, lacking in many previous GPU designs [64, 66]. These hardware memory caches tend to make software-provided data caches held in GPU thread-local storage obsolete.

CHAPTER II

RELATED WORK: Graphics Processing Units and OpenCL

2.1 Introduction

Technologies, such as OpenCL and CUDA, make it possible to run nearly standard C code on graphics processing units (GPUs). GPUs are auxiliary processors that can be packaged on separate cards, included on the central processing unit (CPU) mother board, or manufactured within the CPU's integrated-circuit die.[1] When OpenCL and CUDA are used to solve general problems with GPUs, the acronym "GPGPU," which stands for *General-Purpose computation of Graphics Processing Units*, is often used [69]. With these technologies, one can run parallel algorithms on GPUs with the expectation of achieving high performance. At a first glance, it may be tempting to use basic parallel random access machine (PRAM) algorithms on GPUs, since GPUs appear to supply the majority of the functionality required by the PRAM model. However, we will see that software based only on basic PRAM algorithms tends to perform poorly on GPUs. The solution to this performance issue lies in avoiding the common GPU performance pitfalls such as *instruction path divergence* and *excessive memory latency*.

[1] An integrated-circuit die is a small section of semiconductor material.

2.2 Basic Overview of CPUs, GPUs, and PRAMs

CPUs tend to have a limited number of cores (often less than 16) with significant amounts of cache memory and a limited number of software-managed threads. These CPUs have architectures with specialized logic for predictive branching, out of order instruction execution, and other advanced techniques – all of this aimed at keeping the CPU busy processing instructions. GPUs tend to have hundreds of cores that can simultaneously handle thousands of *hardware-managed* threads. CPUs perform thread switches under software control, whereas GPUs have hardware-managed threads. GPUs can switch between threads with no significant delays, because there is no software-managed context switching involved. Memory caching may be present on newer GPU designs, but when a given thread stalls, GPUs tend to rely on their fast thread switching to keep the processors executing instructions. With the large number of GPU threads, the expectation is that there is going to be a dispatchable thread available. CPUs, on the other hand, tend to rely on memory caching to minimize memory-access-induced stalls [30].

The GPU architecture has a resemblance to the conceptual PRAM. This acronym refers to an abstract model of a machine with memory easily shared between the parallel processors and the presence of as many parallel processors as required [14, 42].[2] From a high-level view, GPUs, with their shared memory and large number of processors and threads, are similar to PRAMs. They appear to be loosely equivalent as each has uniform shared-access to global memory and a large number of concurrent processors. However, to get high performance at a low cost, the GPU architecture has a number of features which make this "equivalence" view incorrect. As will be seen in the discussion on GPU performance, the techniques used to get high levels of GPU performance are contrary to the unrestricted nature of the PRAM model.

[2]For our purposes, we can ignore the four different types of PRAMs outlined by Berman since, in practice, PRAMs and GPUs are not very similar.

2.3 Modern GPUs

The modern GPU is a specialized electronic circuit that is common in almost all computers. Most personal computers now include some type of GPU. The commodity nature of GPUs has helped keep their unit costs low, though they have achieved the potential to exhibit very high throughput [78]. The GPU was originally designed to provide high-speed graphical rendering computer-generated graphics. With the advent of NVIDIA's Computer Unified Device Architecture, the programming of NVIDIA GPUs for non-graphics use became a more straight-forward process [95]. The OpenCL Specification appeared after CUDA and provides for similar programming capabilities over a much wider range of GPU cards and devices [34, 70].

Before CUDA and OpenCL, the problem to be solved in parallel on a GPU had to be re-factored as a rendering display problem, and when the rendering was completed, the display raster image had to be captured and reformatted to generate output results. Now, OpenCL and CUDA make it possible to program GPUs in nearly standard C, using common programming constructs. A key point is that OpenCL and CUDA make it possible to code algorithms, possibly designed for the PRAM model, on the GPU in a manner that does not require presenting the problem as a graphical rendering problem. GPUs can now solve non-rendering problems without resorting to exotic technologies and approaches. Current generation GPUs can process double precision floating point numbers, and high-end GPUs support error correcting memory [66, 65].

As alluded to above, the hypothetical PRAM model and the GPU may appear to be similar. However, the modern GPU is a very specialized piece of hardware and has unique characteristics that make it problematic to code PRAM algorithms directly onto a GPU [42]. Two of these characteristics are listed in Table 2.1. Instruction Path Divergence relates to the nature of the GPU's instruction processing.

The GPU can be described as a *Single Instruction, Multiple Thread* (SIMT) type of parallel processor. SIMT is very similar to the well-known *Single Instruction,*

Multiple Data (SIMD) model of the parallel processor. However, it is different in that work items (threads) can follow different paths through the same code, but at a significant performance penalty [42]. With many current GPU designs, each work-item, in a work-group, executes exactly the same instruction. But, the instructions not in an active work-item, meaning not on the current execution code path, have their results voided. The impact of this GPU design technique is that the work-items not on the current instruction path are effectively inactive. The work-items are not doing any worthwhile work, resulting in a performance loss.

The GPU global memory access delays relate to the high clock speed of the GPU relative to the lower speed of associated global memory. Historically, many GPU designs lacked any type of hardware caching of global memory. More recent GPU designs, such as the NVIDIA Fermi and Keplar designs, do provide some global memory caching [66]. The lack of a cache, or sufficient cache size, is normally compensated for by the GPU rapidly shifting between its hardware-managed threads. At the time a given thread stalls for a memory access, the notion is that another of the hardware-managed threads is ready to dispatch. When solving application problems with high locality of memory reference, this approach works well. However, when solving certain classes of problems with classic algorithms, such as graph-theoretic problems or sparse matrix problems, there may not be a work-item ready to dispatch due to the lack of locality of memory reference. This dispatching irregularity usually results in poor performance.

In summary, when the program code to be executed by a GPU is not designed for GPU use, it may perform very poorly due to instruction path divergence and global-access induced memory stalls. In order to get beyond these issues, software developers can write GPUs programs that access other classes of memory to achieve and use special coding techniques, which are able to avoid excessive instruction path divergence.

Table 2.1: GPU Performance Constraints

	Constraint	Description
1	Instruction Path Divergence	Occurs when threads take different paths through the code.
2	Global Memory Access Characteristics	Each access to global memory needs the time to execute 200-500 instructions.

2.4 GPGPU

The acronym GPGPU, which stands for *General-Purpose computation of Graphics Processing Units*, refers to the use of GPUs to solve general problems beyond the rendering of graphical images. Scientists and software developers noticed that early GPUs provided very fast parallel processing for operations such as scaling and shading. Noticing the high levels of performance achieved, and the low costs, these scientists and developers began to formulate *non-graphical* problems in graphical terms and then perform the processing on inexpensive GPUs. This was a breakthrough, as it showed that GPUs could effectively be used for high performance non-graphical computing, with low hardware costs. As time passed, products like OpenCL and CUDA made it much easier to perform general-purpose computing on GPUs [69]. The term GPGPU now refers both to the early efforts of doing general-purpose computation on graphics-only GPUs [48] and to the wider field of doing general-purpose computation on any type of GPU, be it fully programmable or not [69].

2.5 OpenCL and GPU Basics

The OpenCL specification is managed by the non-profit consortium Khronos Group, and OpenCL-enabled products are supplied by many software and hardware vendors [34]. This specification enables the development of applications over a range of devices, not all of them GPUs. These OpenCL-enabled devices are supplied by

vendors such as NVIDIA, AMD/ATI, and Intel. Altera has recently announced the availability of OpenCL for its high-end FPGA cards [6].

OpenCL devices are programmed in C and the CPU-side of the OpenCL application can be programmed in C or via a C++ wrapper. There are third-party OpenCL bindings for a number of other languages including Java, Python, and Microsoft's .NET. It is worth noting that the GPU programming models supplied by OpenCL and NVIDIA's CUDA are similar conceptually, but not at all the same at the source code level [48]. The use of CUDA is limited to only NVIDIA hardware.

OpenCL is part of the very dynamic graphics hardware and software arena; here, continuous product changes and enhancements are the norm. The features and limits present today may be significantly different in a year – this means that OpenCL is subject to frequent updates. As stated earlier in the *Delimitations* section, this work does not focus on highly specialized, GPU-generation-specific coding techniques, but instead focuses on algorithms, techniques, and approaches for solving GPGPU problems that are applicable to OpenCL over a range of applications and products.

2.5.1 GPU Developer's View and Execution Model

The developer's view of the GPU code is that of one or more kernels. As mentioned above, OpenCL kernels are functions written in the C programming language. OpenCL provides extensions to C that facilitate the execution of the kernel on the GPU. These extensions provide special GPU variable types and access to OpenCL-specific GPU internal variables. OpenCL also provides CPU-side C extension; these extensions provide the complex CPU-side OpenCL API. This API provides a large number of varied CPU functions, and function options. Included are functions to copy buffers of memory to and from the GPU, invoke GPU kernels, and manage error conditions. From a high-level perspective, the CPU copies the required memory buffers to the GPU and then requests that kernels be executed. When the kernels

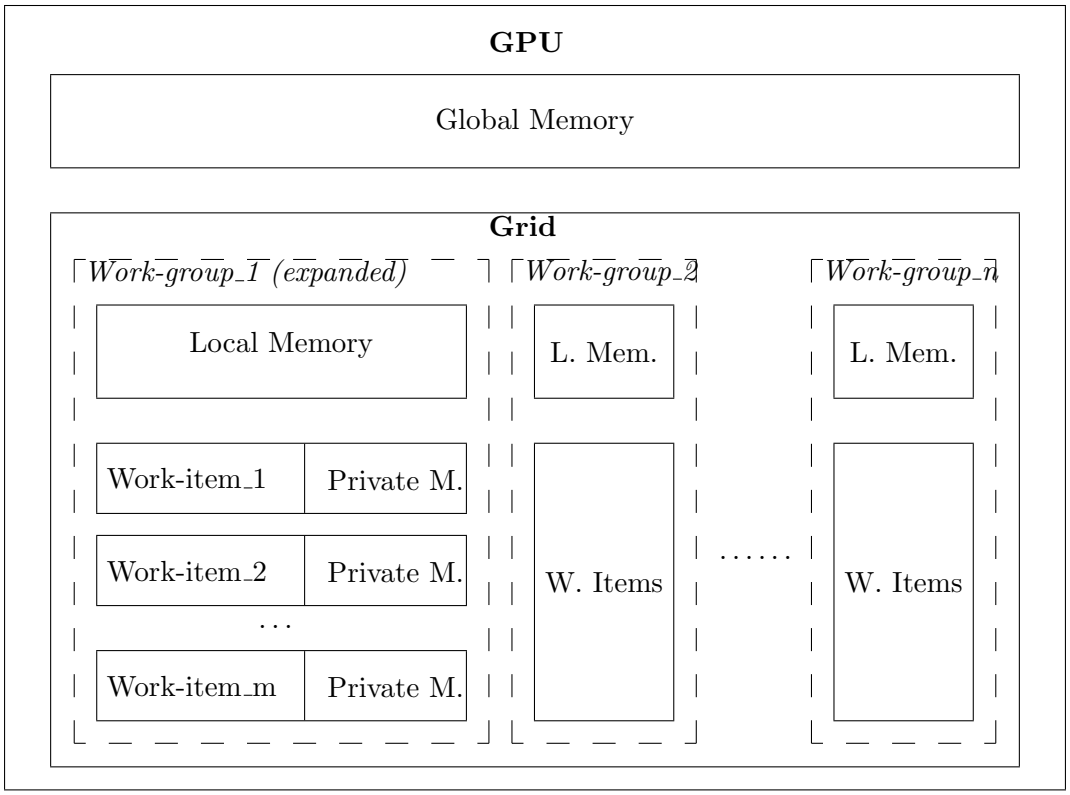

Figure 2.1: OpenCL Developer's View

are finished, the CPU can request buffers be copied back from the GPU.

A kernel is executed by a *work-item*; work-items are grouped into *work-groups*; work-groups are grouped into a *grid*. The developer sees the program code and its execution in terms of kernels, work-items, work-groups, and grids. Figure 2.1 contains a diagram expressing these relationships. At a given time, each work-item is executing one kernel. Each work-item has access to the shared global memory, a limited amount of work-group shared memory, and it's own private memory. How these different types of memory are used by the kernel has a major impact on the GPU performance, because access to the plentiful global memory is relatively slow. It is often necessary to design GPU algorithms that permit reused data to be kept in local memory or private memory [65]. Both local memory and private memory tend to be high-speed RAM, packaged within the GPU itself.

Developers program the GPU kernels and set the characteristic of the work-groups

and grid. However, OpenCL uses *wavefronts* to execute the work-groups, and hence the work-items. The work-groups are executed in arbitrary order, and may or may not execute in parallel. This arbitrary-ordering characteristic of work-group execution impacts the developer, because the order of work-group execution can not be predicted. The algorithms and code used cannot be dependent on work-group ordering. The work-items in a work-group all utilize the same program counter and they can share the local work-group memory [61].

2.5.2 GPU Performance

As the program counter is shared, and since the SIMT model is being used, it is possible for the behavior of one work-item to impact the other work-items in the work-group. In particular, when a decision statement or a loop causes *instruction path divergence*, the work-items not on the instruction path still execute the instructions at the program counter, but the instruction's actions are voided. This means the work-items not on the current path are effectively paused. Depending on the length of the diverged path, the non-active work-items can stay paused for a relatively long period of time. Instruction path divergence is a major issue in getting high performance from GPUs [65, 34].

GPUs are very fast processors, but the time it takes to access global memory can stall the GPU for approximately 200-800 instruction cycles, depending on the actual GPU. This delay is caused by *memory latency*. CPUs have similar memory latency issues, and are designed with sophisticated, multi-level memory caches to help alleviate this performance issue. As mentioned previously, GPUs may also have caches but, in general, GPUs use a different solution to the memory latency issue: GPUs rapidly dispatch another thread that is not stalled. It becomes the developer's task to utilize an algorithm that facilitates the availability of a dispatchable thread. This is one of the areas where the PRAM view of GPUs breaks down; PRAMS do not have

this type of requirement. Techniques such as *data pre-fetch*, use of local registers, and use of local shared memory are often employed to mitigate memory latency issues in GPUs. The GPU's hardware registers and local memory are accessible with minimal delays and the pre-fetching of data involves using techniques to pre-load the required data into local storage or a software cache [48].

An additional major GPU performance consideration occurs when transferring data between the CPU and the GPU. As a high-performance GPU is often provided on a separate card, located on a "slow" PCI Express bus, the movement of data between the CPU and GPU is slow relative to the performance of the GPU [30]. This performance difference, between the PCI Express bus data transfer rate and the GPU throughput rate, is a major concern in achieving high performance [65].

When using complex algorithms, achieving high performance with OpenCL can be a complex and difficult undertaking. In order to achieve good GPU performance, Farber suggests the following three basic rules for GPGPU programming [30]: (1) get the data on the GPU and leave it there; (2) give the GPU ample work to do;[3] and, (3) focus on the reuse of data within the GPU to avoid memory bandwidth limitations. These three basic rules form the basic tenants for DEFG's generation of OpenCL code.

2.6 Parallelization and Domain-Specific Languages

Numerous attempts have been made to construct languages, compilers, and tools to make the production of high performance parallel solutions easier. In 2003, Shen et al. talked about the holy grail of parallelization, which is the automated parallelization of serial programs, being out of reach [83]. However, progress is being made. One approach towards the efficient production of GPU-based parallel solutions is the use

[3]Of these three rules, perhaps this one is the most complex. Finding ways to always have the GPU working is not easy in the face of possibly hidden instruction path divergence and memory latency issues.

of a domain-specific language (DSL). DEFG is a DSL, a language and related tools that facilitate the production of OpenCL applications. Martin Fowler defines a DSL as a computer programming language of limited expressiveness focused on a particular domain, and suggests that DSLs can be broken into two categories: internal DSLs and external DSLs [32]. DSLs of both varieties have been produced for GPU-based high performance computing.

Internal DSLs for GPU-based HPC include extensions to Python, such as PyGPU, PyCUDA, and PyOpenCL [50, 49, 53]. These DSLs tend to consist of Python wrappers placed around a particular GPU's API. There are also C/C++ extensions, such as Bacon [92]. Aside from DEFG, other GPU external DSLs include the SPL digital signal processing language and the MATLAB Parallel Computing Toolbox. The MATLAB toolbox supports CUDA and it permits passing some MATLAB functions to the GPU. It also permits direct GPU kernel execution [57, 100]. Both MATLAB and DEFG require that the GPU kernel be provided by the developer.

DSLs have the ability to provide high-level *abstractions* for complex computing tasks; they can be used to hide complexity [32]. In this dissertation, we show a DSL, namely DEFG, which provides abstractions for the complex CPU code that must be written for OpenCL GPU applications. These abstractions are produced in such a way that the developer is shielded from a great deal of complexity encountered when using the various OpenCL API functions and options [45].

CHAPTER III

OVERVIEW OF DEFG AND ITS PERFORMANCE

3.1 Introduction

This chapter provides an overview of DEFG and summarizes its capabilities and performance [80, 81]. Later chapters will describe the internal workings of DEFG and its associated design patterns, and show the use of DEFG with additional applications. In addition, Section A of the Appendix provides a full description of the DEFG language. Here, we present three sample DEFG application solutions and discuss the way DEFG relates to the application's OpenCL host code and kernel code. We focus our attention on the basics of the DEFG environment and actual DEFG performance results, in terms of both developer productivity and run times.

We approach this discussion of the DEFG implementation as follows: using three existing OpenCL applications and their existing OpenCL kernels without any changes, the existing host CPU is replaced with DEFG-generated code. The DEFG source modules need, on average, about 90% fewer lines of code than the corresponding hand-written host OpenCL modules. We compare the computational performance of the three applications over two different OpenCL platforms, which we call CPU and GPU-Tesla. Performance variations between the DEFG and reference results are identified and analyzed. The next few pages summarize the DEFG implementation and DEFG language, as well as the three existing OpenCL applications we use as reference applications and their conversion to DEFG. We then present a preliminary

look at our experimental results, in terms of lines of code and run times.

3.2 DEFG Framework and DEFG Language

The DEFG implementation consists of a parser written in Java, utilizing ANTLR3 [11], a Java-based optimizer specific to DEFG, and our code generator, which is written in C++. The parser handles syntax checking and results in an abstract syntax tree, expressed as an XML document. The XML syntax tree is then optimized for run-time performance and decorated with cross-reference information needed for code generation. Finally, this tree is processed by our code generator, which uses the TinyXML2 library to accept the XML-formatted tree [91]. For example, the twelve lines of DEFG code shown in Figure 3.1 result in approximately 460 lines of C/C++ code, a snippet of which is shown in Figure 3.2. The OpenCL kernel executed by this code is shown in Figure 3.3. Note that this generated OpenCL code is intended to execute on any supported OpenCL device, including the CPU. With OpenCL, the CPU can function as both the host and the device to execute the kernel.

The DEFG declarative language consists of a number of declare, execute and call statements, and optional statements, such as sequence/times and loop/while. An example DEFG source file is shown in Figure 3.1. The declare statement is used to name the DEFG application, to define and name the GPU kernels to be executed, to define any required scalar variables such as a graph's node count, and to define the

```
01. declare application sobel
02.  declare integer Xdim (0)
03.          integer Ydim (0)
04.          integer BUF_SIZE (0)
05.  declare gpu gpuone ( * )
06.  declare kernel sobel_filter SobelFilter_Kernels ( [[ 2D,Xdim,Ydim ]] )
07.  declare integer buffer image1 ( Xdim Ydim )
08.          integer buffer image2 ( Xdim Ydim )
09.  call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
10.  execute run1 sobel_filter ( image1(in) image2(out) )
11.  call disp_output (image2(in) $Xdim (in) $Ydim (in) )
12. end
```

Figure 3.1: Sample DEFG Code

```
// *** buffers in
cl_mem buffer_image1 = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
                        (BUF_SIZE * sizeof(int)),(void *) image1, &status);
if (status != CL_SUCCESS) { handle error }
status = clSetKernelArg(sobel_filter, 0, sizeof(cl_mem), (void *)&buffer_image1);
if (status != CL_SUCCESS) { handle error }
cl_mem buffer_image2 = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                        (BUF_SIZE * sizeof(int)),(void *) NULL, &status);
if (status != CL_SUCCESS) { handle error }
status = clSetKernelArg(sobel_filter, 1, sizeof(cl_mem), (void *)&buffer_image2);
if (status != CL_SUCCESS) { handle error }
// *** execution
size_t global_work_size[2]; global_work_size[0] = Xdim ; global_work_size[1] = Ydim ;
status = clEnqueueNDRangeKernel(commandQueue, sobel_filter, 2, NULL, global_work_size,
           NULL, 0, NULL, NULL);
if (status != CL_SUCCESS) { handle error }
// *** result buffers
status = clEnqueueReadBuffer(commandQueue, buffer_image2, CL_TRUE, 0,
           BUF_SIZE * sizeof(int), image2, 0, NULL, NULL);
if (status != CL_SUCCESS) { handle error }
```

Figure 3.2: Snippet of Generated OpenCL Code

```
__kernel void sobel_filter(__global uchar4* inputImage, __global uchar4* outputImage) {
        uint x = get_global_id(0);  uint y = get_global_id(1);
        uint width = get_global_size(0);  uint height = get_global_size(1);
        float4 Gx = (float4)(0);  float4 Gy = Gx;
        int c = x + y * width;
        /* Read each texel component and calculate ..*/
        if( x >= 1 && x < (width-1) && y >= 1 && y < height - 1)
        {
                float4 i00 = convert_float4(inputImage[c - 1 - width]);
                // similar lines omitted
                float4 i22 = convert_float4(inputImage[c + 1 + width]);
                Gx =   i00 + (float4)(2) * i10 + i20 - i02  - (float4)(2) * i12 - i22;
                Gy =   i00 - i20  + (float4)(2)*i01 - (float4)(2)*i21 + i02  -  i22;
                /* taking root of sums of squares of Gx and Gy */
                outputImage[c] = convert_uchar4(hypot(Gx, Gy)/(float4)(2));
        }
}
```

Figure 3.3: Snippet of Sobel OpenCL Kernel Code

buffers to be transmitted to and from the GPU device. Lines 1 to 8, in the DEFG sample, express declare statements. The line 6 statement syntax that is enclosed in "[[" and "]]" brackets is our method of setting the global grid size. The call statement is used to invoke C/C++ functions, e.g., to obtain the input data; this sample has call statements on lines 9 and 11. The execute statement on line 10 is used to execute the kernel. The flow of control is a design pattern built into DEFG and, in this case, it executes the statements in order, after the declare statements.

The optional statements can be used to provide support for more complex design

patterns where the GPU kernels may have to be executed a variable number of times. Figure 3.4 contains a DEFG example which executes the kernel once for each graph node. Figure 3.4, line 9, shows the use of the sequence statement. DEFG also contains additional looping statements to process scalar values returned by the GPU kernels. This capability is used in the DEFG breadth-first search solution to conditionally stop the parallel device processing. DEFG generates OpenCL 1.1 code in keeping within the limits of NVIDIA's current OpenCL support [67].

```
01. declare application floydwarshall
02. declare integer NODE_CNT (0)
03.         integer BUF_SIZE (0)
04. declare gpu gpuone ( any )
05. declare kernel floydWarshallPass FloydWarshall_Kernels ( [[ 2D,NODE_CNT ]] )
06. declare integer buffer buffer1 ( $BUF_SIZE )
07.         integer buffer buffer2 ( $BUF_SIZE )
08. call init_input (buffer1(in) buffer2(in) $NODE_CNT(out) $BUF_SIZE(out))
09. sequence NODE_CNT times
10.     execute run1 floydWarshallPass ( buffer1(inout) buffer2(out) NODE_CNT(in) DEFG_CNT(in) )
11. call disp_output (buffer1(in) buffer2(in) NODE_CNT(in))
12. end
```

Figure 3.4: Sample DEFG Code Showing a Sequence

3.3 Viability of DEFG

In order to test the viability of DEFG, we selected three existing OpenCL applications based on well-known algorithms: Sobel image filtering and Floyd-Warshall all pairs shortest path (APSP), both from the AMD APP SDK, and breadth-first search from the OpenDwarfs benchmark [1, 31]. We will refer to these applications as SOBEL, FW, and BFS, respectively. SOBEL was chosen (1) because it represents the class of less-complex GPU problems, where a single kernel is called once and (2) because it has significant RAM locality of reference. DEFG supports concurrent execution on multiple GPU devices, in a declarative manner, and SOBEL provides a good test case for this added support. This multiple-GPU support is discussed later in Section 5.1.

FW and BFS were selected because they represent two different classes of graph-

Table 3.1: Test Configurations

Name	Configuration Data
CPU	Windows 7, Intel I3 Processor, 1.33 GHz, 4 GB RAM, using AMD OpenCLSDK 2.8 (no GPU)
GPU-Tesla T20	Penguin Computing Cluster, Linux Cent OS 5.3, AMD Opteron 2427 Processor, 2.2 GHz, 24 GB RAM, using NVIDIA OpenCL SDK 4.0, NVIDIA Tesla T20 with 14 Compute Units, 1147 MHz and 2687M RAM

Table 3.2: Lines of Code

	DEFG		
	Declarative	Generated	Reference
BFS	42	620	364
FW	12	481	478
SOBEL	12	467	442

oriented GPU problems, with BFS being the more complex to implement in DEFG. The FW algorithm requires that a common operation be repeated for each graph node. In this implementation, FW's GPU kernel is called once for each node. This call-for-each-node behavior must be managed from the CPU host, and hence from DEFG. The OpenDwarfs BFS implementation is based on the work by Harish and Narayanan, and uses a version of Dijkstra's algorithm [22, 38]. The actual OpenDwarfs BFS code is an OpenCL port of the CUDA code from the Rodinia benchmark [85]. This BFS implementation requires that a pair of kernels be repeated until success is indicated by the second kernel. This repetition is managed by the CPU host code.

All three of these applications were converted to DEFG, keeping the unmodified OpenCL kernels. The conversions to DEFG produce exactly the same results as the corresponding reference version. Before discussing the performance results, we summarize the hardware and software used. The tests were run on two different configurations, which we call CPU and GPU-Tesla T20. These configurations are described in Table 3.1. In GPU performance terms, the CPU configuration is significantly less powerful than GPU-Tesla T20 because the CPU is not using a GPU resource; it executes the kernel on the CPU.

Table 3.3: Run-time Performance, in Milliseconds

	CPU		GPU-Tesla T20	
	DEFG	Ref.	DEFG	Ref.
BF-4096	1.5	2.6	4.3	5.8
BF-65536	12.3	14.2	8.0	11.3
FW	111.8	152.0	6.0	51.2
SOBEL	23.0	24.8	3.7	4.1

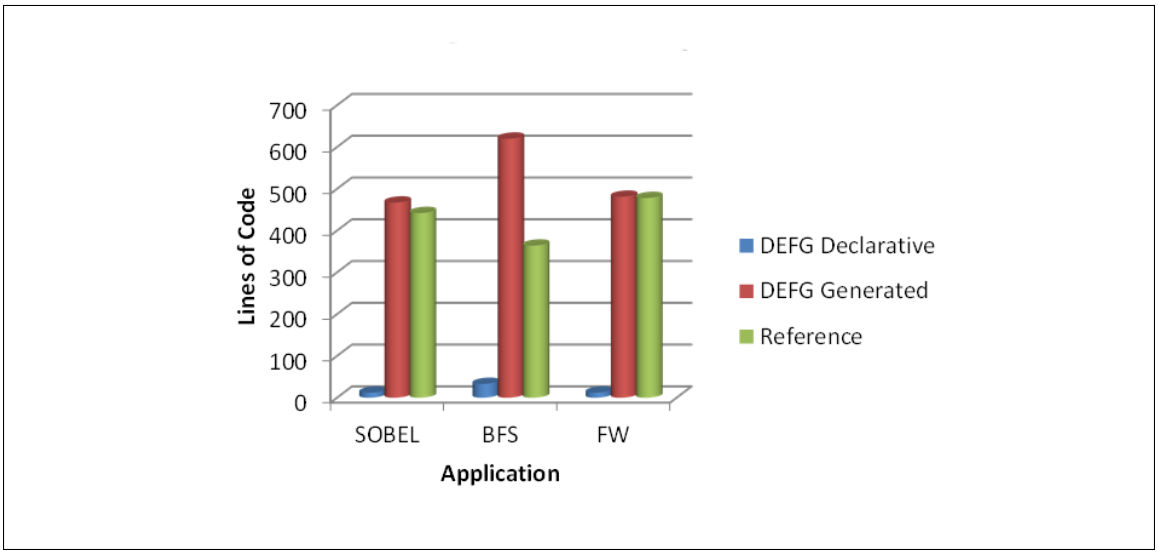

Figure 3.5: Application Lines-of-Code Comparison

3.4 Discussion of Results

In terms of developer-written module line-count results, the three DEFG modules were much smaller than their reference counterparts. Table 3.2 lists the line counts for SOBEL, BFS, and FW; shown are the number of lines of DEFG declarative code, the number of lines of DEFG-generated OpenCL code, and the estimated number of non-comment lines in the OpenCL reference version. This data is shown graphically in Figure 3.5. On average, the DEFG code is 3.9 percent of the generated code, and 5.6 percent of the reference code. It should be noted that the reference code tended to include additional functionality and that the DEFG generated-code counts included an additional 150 lines of template code used to identify and select the requested GPU devices.

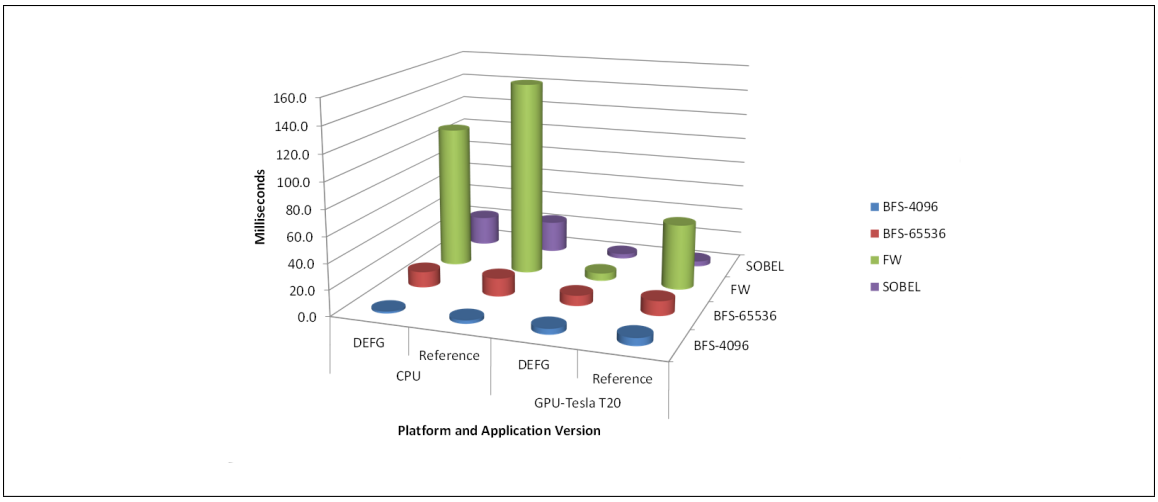

Figure 3.6: Application Run-Time Performance Comparison

The run-time performance comparison turned out to be very interesting. The raw run times, in milliseconds, are presented in Table 3.3. Figure 3.6 presents this data in 3D form. The results shown are the application averages, over ten individual runs for each application. When unexpected results were encountered, we performed reruns with temporary manual code modifications in an effort to isolate the root causes of the unexpected behaviors. At different times, these code changes were made into both the DEFG and reference OpenCL code. However, the numbers shown here are only the original times, i.e., those prior to any manual code modifications.

SOBEL is the simplest application and the run-time performance results between DEFG and the reference cases are comparable. The SOBEL results are shown on the graph in purple. The DEFG performance was slightly faster on all of the configurations. This similarity of results is not surprising as the OpenCL CPU host operations to execute SOBEL are not complex.

The run-time results of the FW tests, which are shown in green, were a surprise to us. We saw no obvious explanation for why DEFG should be consistently faster. For example, with GPU-Tesla T20 the reference FW needed 51.2 ms while DEFG FW consumed only 6 ms. We reviewed the OpenCL code for both DEFG and the AMD SDK-supplied reference case, and did not find any significant differences in buffer

usage or the OpenCL API functions used. We did notice that the reference case was using asynchronous events, when not in fact required, and we temporarily disabled them and reran the reference case. The FW reference case run times on the Tesla T20 dropped three-fold from an average 51.2 ms to 17 ms. We feel the DEFG Tesla time of 6 ms and the reference case time of 17 ms are reasonably close and this test tends to show that, for this implementation of the Floyd-Warshall algorithm, both the DEFG and reference run times were reasonably comparable.

The BFS run-time comparisons used two different graphs. The first graph has 4,096 nodes, with the results shown in blue on the graph, and the second has 65,536 nodes, shown in red. As a historical note, our earlier prototype version of DEFG BFS was substantially slower than the reference version of BFS; the prototype DEFG needed 59.4 ms to perform what the reference BFS did in 11.3 ms. The full DEFG version contains buffer management optimization. When the full DEFG is used, the 59.4 run time drops to 8.0 ms. This dramatic improvement in performance was due to the DEFG optimizer's removal of unneeded buffer transfer operations.

We cannot leave the BFS performance topic without noting that the OpenCL CPU configuration's performance was better than GPU performance for the 4,096 node case. We postulate that this is explained by the BFS implementation being used. This graph algorithm implementation is based on the work by Harish [38], which does not compensate for the lack of memory caching on many GPUs. The CPU version most likely fared so well due to the multiple levels of memory caching provided by the Intel I3; it is also likely that the 4,096 node case fit entirely in the Intel I3's cache.

In summary, these experiments have shown that, at least with these three applications, the declarative approach used in DEFG can be used to produce OpenCL applications with fewer lines of code and comparable run-time performance levels, relative to hand-written OpenCL application host code.

CHAPTER IV

DEFG THEORY OF OPERATIONS

4.1 Introduction

Our Declarative Framework for GPUs (DEFG) generates the CPU-side code for OpenCL GPU applications. One of the principle DEFG goals is to make the development of GPU software less onerous and difficult. The OpenCL GPU APIs tend to be numerous, very complex, and verbose. DEFG hides much of this complexity and removes unneeded verbosity. Our approach enables the developer, where possible, to declare what is needed and have the DEFG software generate the solution to produce *what is needed*.

Another principal goal is to have DEFG generate code that is humanly readable, allowing DEFG to also function as a learning tool. The developer can declare what is needed and then review the generated DEFG code for insights into how to utilize OpenCL in separate, non-DEFG applications. We use static optimization techniques as much as possible, so as to have the DEFG-generated code be readable. This approach tends to avoid the use of complex optimization and dynamic-decision code at run time. We avoid them because their use tends to make the generated code large, heavy with called modules, and rather hard to understand.

As with the CPU, the GPU has limits; it has finite amounts of processing power and memory. In order to get beyond the limits present in a single GPU, a DEFG aim is to facilitate the use of multiple GPUs within a single application. GPU kernel

designs often break their work into small units that have minimal interactions with each other. With these types of designs, spreading the work over additional GPU hardware is relatively easy to do, if the work can be partitioned in a reasonable way. DEFG provides for this work partitioning in a number of common application use cases. These cases will be described in the section on DEFG Design Patterns.

This Theory of Operations Chapter is divided into two main sections, plus a smaller third section. Section 4.2 talks about the DEFG Design Patterns and stays at a rather high level. It is written from the point of view of a developer wanting to use DEFG. Section 4.3 dives into the concepts and notions behind the workings of DEFG and provides numerous specific implementation details. This second subsection is written from the point of view of the intrepid soul wishing to understand the inner workings of the DEFG environment. Section 6.2.1 explores why a DSL is useful for GPU software development.

4.2 DEFG Design Patterns

In software engineering, the term *design pattern* is often ambiguous; different researchers, designers, and developers may view the notion of a design pattern with varied experiences, expectations and requirements.[1] So as to avoid confusion, we will define what we mean by a DEFG design pattern after we provide a brief overview of the DEFG domain. We will then describe the existing common DEFG design patterns. Before proceeding, we note that a given domain may use only certain commonly described software engineering design patterns and may introduce unique design patterns of its own.

The DEFG language is a domain specific computer language, commonly called a "DSL". As such, it is intended to be used to solve problems within its limited domain. The DEFG domain mainly consists of providing the CPU-side code for OpenCL

[1] We know this to be true from our own extensive software engineering experiences.

applications. A key concept here is that DEFG generates a limited range of solutions and this limited range is reflected in its design patterns. Many of the common software engineering design patterns such as *fork-join*, from process control, *pipeline*, from data management [58], and *observer* [33], from object-oriented programming, do not apply to DEFG.

For purposes of this dissertation, we will use the Gamma, et al. definition of a *design pattern* [33]; a design pattern "is a solution to a problem in a context." Some of our design patterns are very simple and others are quite complex. When describing our more complex patterns, we will follow the Gamma approach of describing the DEFG patterns with four essential elements: the *pattern name*, the *problem* addressed, the *solution* provided, and the *consequences* of use.

4.2.1 DEFG Invocation Patterns

4.2.1.1 Sequential-Flow Pattern

The Sequential-Flow pattern is the default behavior of DEFG. In DEFG programs, all of the statements after the declare statements are executable statements and the default behavior is to execute these in order, from top to bottom. DEFG programs that use this pattern, and not the Single-Kernel Repeat Sequence and Multiple-Kernel Loop patterns, discussed next, tend to have the following structure:[2]

```
declare application  <name>
  declare integer ...
  declare gpu ...
  declare kernel ...
  declare integer buffer ...
  call <get data CPU function> ....
  execute <GPU kernel> ...
  call <consume data CPU function> ...
end
```

For some DEFG applications, this simple behavior is sufficient to provide the

[2]Three dots, an ellipsis, represent omitted text and text between < and > represents the name of applications, kernels, functions, etc.

basis for a solution. Each referenced GPU kernel and GPU function is executed once and the execution occurs in the order listed. The call and execute statements can be mixed in any order. This design pattern is clearly shown in our SOBEL and MEDIAN digital filter applications[3].

4.2.1.2 Single-Kernel Repeat Sequence Pattern

When a given kernel needs to be repeatedly executed a fixed number of times, and this number is known before the kernel is executed, the Single-Kernel Repeat Sequence design pattern is used. This pattern provides the preferred DEFG approach to repeatedly executing a kernel for a preset number of iterations. Whenever possible, this pattern is preferred over the Multiple-Kernel Loop pattern because the DEFG optimizer can more easily deal with the OpenCL data transfer operations. The Invoke-While pattern (see below) has more interaction concerns and may not optimize as well by DEFG. This Single-Kernel Repeat Sequence pattern may be used within the Multiple-Kernel Loop pattern to help perform complex operations. DEFG programs using this pattern tend to have this structure:

```
declare application  <name>
  declare integer ...
  declare gpu ...
  declare kernel ...
  declare integer buffer ...
  ....
  sequence <number> times
    execute <GPU kernel> ...
  ...
end
```

The referenced GPU kernel is executed <number> times. This design pattern is very handy when a self-contained iteration operation must be executed, and is demonstrated in our Floyd-Warshall (FW) application.

[3]The DEFG application names are capitalized.

4.2.1.3 Multiple-Kernel Loop Pattern

Sometimes the number of times a GPU kernel, or other group of DEFG statements, needs to be repeatedly executed is not known in advance. For this case, DEFG provides this Multiple-Kernel Loop pattern. The range and flexibility of this design pattern, and the loop/while DEFG statements that implement it, are purposely limited by this domain specific language so as to enable our very useful DEFG optimizations. The main limit imposed is that this design pattern cannot be embedded. The Multiple-Kernel Loop design pattern tends to have this style of DEFG coding:

```
declare application  <name>
  declare integer ...
  declare gpu ...
  declare kernel ...
  declare integer buffer ...
  ...
  loop
    ...
    execute ...
    ...
    execute ...
  ...
  while <variable> <condition> <constant>
  ...
end
```

We note that this pattern can contain the previous two patterns. The consequence of this pattern's non-embedded limit is that some algorithms/applications may have to be re-factored; however, as re-factoring for GPU use is common, our experience is that this will not likely be an issue for the developer. We have found that by having the GPU kernels perform the work and having the CPU manage the work, this pattern's limits are not frequently encountered. Of course, there may be applications where this type of re-factoring is not desirable or possible. In this case, using DEFG might not be the correct software development tool. Our BFS applications show this Multiple-Kernel Loop pattern in use; it is also used in our iterative matrix inversion (IMI) application.

4.2.2 DEFG Concurrent-GPU Patterns

Providing support for applications to utilize more than a single GPU is a primary aim of DEFG. In order to provide this capability with minimal code changes, we supply several design patterns. The Multiple-Execution pattern engages additional GPUs.

4.2.2.1 Multiple-Execution Pattern

This pattern is both *simple* and deceptively *complex*. It is simple in the sense that with several simple code changes, any non-BLAS[4] DEFG program can be changed to utilize more than a single GPU. It is complex in that it only makes sense to do this if the algorithms and kernels used by the application can support multiple-GPU operations. Additional comments about this pattern are given later in Section 5.1 and in Section A of the Appendix.

The problem this design pattern addresses is the need for more GPU processing power and memory. The solution it provides is to engage additional GPUs in the application execution. The benefits of adding more GPUs, when the algorithms and kernels permit, may seem rather obtuse from a distance. However, if the application data needing to be processed on the single GPU is a single byte larger than the RAM available on the single-GPU, the application will fail. The ability to get a solution by utilizing a second, already-available GPU without having to rewrite the entire application is a significant advantage.

[4]The BLAS design pattern is discussed below.

This pattern is engaged by using a declare gpu statement that selects multiple GPU devices, and using multi_exec statements (instead of execute statements). The consequences of using this pattern are enormous. If the application is not designed correctly, it will fail or worse, produce wrong results. If the application is designed for multiple GPU use, there is the potential for getting results more quickly, handling larger application data sets, or both. DEFG programs that use this pattern tend to have this structure:

```
declare application  <name>
  declare integer ...
    declare gpu <name> ( all )
  declare kernel ...
  declare integer buffer ...
  ...
  multi_exec ...
  ...
end
```

A number of our applications, including: SOBELM, MEDIANM, RSORTM and BFSDP2GPU, use this pattern. The BFSDP2GPU application utilizes this pattern in a much more complex manner compared to the other listed applications. This breadth-first search application has active communications between its two GPUs and it also manages the multiple-thread updating of its shared buffer with the Prefix-Allocation pattern, which is discussed below. The BFSDP2GPU application is specifically discussed in Section 5.2.

4.2.2.2 Divide-Process-Merge Pattern

This pattern is used in association with the multi-GPU pattern. It is as much of a set of design guidelines as it is a pattern that activates additional DEFG features. When the Multiple-Execution pattern is active, DEFG's default behavior is to automatically divide the data buffers into segments of equal size and give each segment to a unique GPU. DEFG also correspondingly lowers the size of the work-group for each GPU.

We note that this behavior can be changed by using the buffer options, as discussed in Section A of the Appendix.

When DEFG is left in its default multi-GPU buffer behavior, the developer must be certain that this mode of operation fits the application's requirements. It may be the case that the CPU function used to display, or further process, the resulting data may have to take special steps. In particular, our RSORTM application has special merge processing in the CPU function that writes the resulting sorted data to disk. This design pattern uses the default multi-GPU buffer behavior, which splits the buffers into equal segments. Our RSORTM application makes use of this pattern and is discussed in Section 5.3.

4.2.2.3 Overlapped-Split-Process-Concatenate Pattern

This pattern is also used in association with the multi-GPU pattern. It is mutually exclusive to the preceding Divide-Process-Merge pattern. This pattern is engaged by adding the halo option to the buffers holding the GPU data, as shown in this DEFG abbreviated code:

```
declare application  <name>
  declare integer ...
    declare gpu <name> ( all )
  declare kernel ...
  declare integer buffer ... halo (<n>)
  ...
  multi_exec ...
  ...
end
```

Some data contains boundaries or edges that bring about special processing requirements when work splitting is engaged. The obvious case for this is the two-dimensional image filtering. But, it also shows up in applications such as moving-average calculation and digital signal processing. When the dimensions of these edges are fixed in size and known in advance, this pattern can be used to have DEFG automatically duplicate the edge data between GPUs. DEFG manages internally the

insertion of the duplicated information at the splits of the data; it also removes these overlaps when the data is returned. The halo buffer option causes DEFG to use this special processing and the associated n value provides the size of the overlapped area. For a 1D structure, the n value represents individual elements. For a 2D structure, n provides the number of overlapped (image) lines. This design pattern is very effectively used in our SOBELM and MEDIANM image processing applications.

4.2.3 DEFG Prefix-Allocation Pattern

As part of DEFG, a number of OpenCL kernels are supplied. These kernels include: *bermanPrefixSumP1*, *bermanPrefixSumP2b*, and *getCellValue*. The first two kernels are based on the prefix sum algorithms given in Berman [14]. This very abbreviated code shows these kernels being used:

```
...
multi_exec run2 PrefixSumP1( offset (out) ...
...
sequence KCNT times
  multi_exec run2 PrefixSumP2b (offset2 (inout) offset (inout) ...
...
multi_exec run3 getCellValue( offset2 (in) ...
...
```

These three kernels form a general prefix sum capability that can be used from any DEFG application. We considered using other prefix sum (prefix scan) algorithms; however, they had the power-of-2 buffer size requirement, which is not acceptable here, since DEFG is intended to be used in general-purpose manner. The provision of a prefix sum capability in DEFG makes it possible to allocate the space in a shared buffer without having to use performance-impacting, low-level synchronization constructs. The ideas behind this buffer allocation approach are discussed in Section 5.2. The *getCellValue* kernel returns the last value in the results buffer, which is equal to the number of items used when allocating buffer space.

This code does not show some of the "housekeeping" steps needed to manage the buffer passed to the *getCallValue*. These steps and the full parameter lists for the kernels can be observed in our multi-GPU BFSDP2GPU application's source code. The source code for these kernels is given in the Source Code Appendix, Section B.4.

4.2.4 DEFG Dynamic-Swap Pattern

This design pattern represents the DEFG capability we added to logically swap GPU buffers. The use of this pattern increases DEFG performance and makes the DEFG source code easier to read. It is common to see a given GPU kernel repeated a number of times where the output from the previous iteration is the input to the current iteration. DEFG assigns a fixed name to each GPU buffer. Arrays of buffers are not supported. This pattern enables the content of two fixed-name buffers to be swapped without actually moving the data between the buffers – the buffers are "interchanged" by just swapping their respective CPU references. This pattern is used via the interchange statement, as shown in this abbreviated DEFG code:

```
...
declare integer buffer <bufferA> ...
declare integer buffer <bufferB> ...
...
interchange(<bufferA> <bufferB>)
...
```

This very handy design pattern is used in our RSORT application.

4.2.5 DEFG Code-Morsel Pattern

The Code-Morsel design pattern came about after our surrender to the elegant power of small C/C++ code snippets. After a great deal of thought, we included in DEFG the capability to insert arbitrary snippets of C/C++ code. As we utilized this facility, we came to view these snippets as very useful and we, correspondingly, gave them a

positive name; we called them "morsels." DEFG morsels can be used in a number of ways. We break down their use into two categories: *cosmetic* and *functional*.

Cosmetic morsels are generally benign and are used to do things like add descriptive program output or assist with application debugging. They do not participate in the basic processing of the application. Functional morsels do participate in the active processing of the application and they can add a lot of power to the application. Unfortunately, they also have the potential to create hideous, hard-to-find bugs.

The code in the morsels is not parsed by DEFG. This means the DEFG optimizer has no indication what the morsel is doing as far as consuming data or updating it. Morsels are used via the include and code statements. Appendix Section A, the DEFG User's Guide, provides additional morsel-usage information. Shown here is an abbreviated morsel sample:

```
...
code [[ printf("version %s size: %d, logSize: %d\n", ... ]]
...
loop
  ...
  // something in this loop sets values in buffer againPart ...
  ...
  code [[again = againPart[0] + againPart[2]; ]]
while again ne 0
...
```

The first code statement used above is cosmetic and the second is functional. Cosmetic morsels are used in many of our applications and functional morsels are used heavily in the BFSDP2GPU application.

4.2.6 DEFG Anytime Pattern

Anytime processing is our facility to stop an algorithm or kernel before its normal ending point, so that results can be presented earlier. This is not program termination associated with an error. Here is an abbreviated DEFG sample:

```
...
loop
  ...
  loop_escape at < n > ms
  ...
while again ne 0
...
```

A common use of the anytime approach is to release the current application results when an event, such as the passage of a certain amount of time, occurs. Of course, this type of processing is only of value if the application's algorithms and design facilitate the presence of incremental results. This pattern is used in our iterative matrix inversion application and described in Section 5.4.

4.2.7 DEFG BLAS-Usage Pattern

The BLAS-Usage pattern enables the use of BLAS-based double-precision matrix multiplication from the AMD clMath library [2].[5] This pattern is used with the DEFG blas statement. This abbreviated DEFG code shows its use:

```
...
declare double < ds > ...
        double < ds > ...
declare double buffer <bufferA> ...
        double buffer <bufferB> ...
        double buffer <bufferC> ...
...
blas ( < d1 > * < bufferA > * < bufferB > + < d2 >  * <bufferC>
      -> <bufferC> )
...
```

Here $d1$ and $d2$ are scalar variables; $bufferA$, $bufferB$, and $bufferC$ are buffers

[5]DEFG has the potential to include other capabilities from this, and other OpenCL-oriented, application libraries.

holding the matrices. The results are stored in $bufferC$. This design pattern is used in our iterative matrix inversion (IMI) application.

4.2.8 When to consider not using DEFG

After describing DEFG and it's design patterns, it is important to note that using DEFG to create certain types of OpenCL applications might not be a wise choice. Here are the application characteristics we have identified that indicate that DEFG use may be questionable:

1. *The application has "tight" integration with other components concerning resource sharing, threading models, specialized GUI processing, etc.*

   This tight integration is likely to lead to problems since DEFG manages all its resources (connection handles, buffers, GPUs, etc.) as though it is the sole user. DEFG is designed to be used by a single operating system thread; it is not a multi-threaded CPU application.

2. *Some of the application's OpenCL operations are conditionally executed.*

   It is possible to put conditional DEFG morsels around the DEFG execute and multi_exec statements. However, this approach is likely to create issues as the DEFG optimizer cannot be depended upon to have the application's variables and buffers updated with the correct content.

3. *Complex CPU processing is integrated with the GPU processing.*

   This item is similar to the previous one. If the complex CPU processing can be put into functions invoked with the DEFG call statement, this approach will likely work. If the complex code is inserted with DEFG morsels, the problems described in the previous point are likely to occur.

4. *Complex error handling, such as ignoring an error or restarting after an error, is in use.*

When DEFG encounters an error condition, it expects to terminate its processing. The DEFG User's Guide, presented in Appendix Section A, describes how the DEFG generated-error text and the call to the *exit()* function can be redirected. However, DEFG does not generate code to continue execution after an error.

4.3 DEFG Internal Operations

4.3.1 The DEFG Translator

The general design of our DEFG Translator is diagrammed in Figure 4.1. The translator is really a compiler; it inputs the DEFG source code and outputs C/C++ code, which is then used by a standard compiler. We use the term *translator* with our DEFG tool, instead of the more common *compiler* term, to simplify describing the interactions of the DEFG translator and the standard compiler.

Before describing the specifics of how DEFG programs become C/C++ programs, let us consider what type of application is produced by DEFG. If we ignore the GPU aspects of DEFG, we can say that DEFG is a domain specific language (DSL) that has two main facets. First, DEFG actions rely upon the flow of control options just described; in other words, DEFG has a fixed set of inter-mixable execution-flow models that it follows. Second, DEFG marshals data for movement to remote devices and it manages these devices with optimized remote procedure calls (RPCs). In addition, there are provisions to insert additional CPU actions into the DEFG programs via the use of the DEFG `call` and `code` statements. However, while DEFG facilitates the execution of these additional actions, it does not "understand" or "manage" them; they exist outside the context of the DEFG Translator. We now describe our translator in detail.

The translation is done in three steps, as depicted in Figure 4.1. The DEFG source code, stored in a text file, is processed by the DEFG Parser and the results of

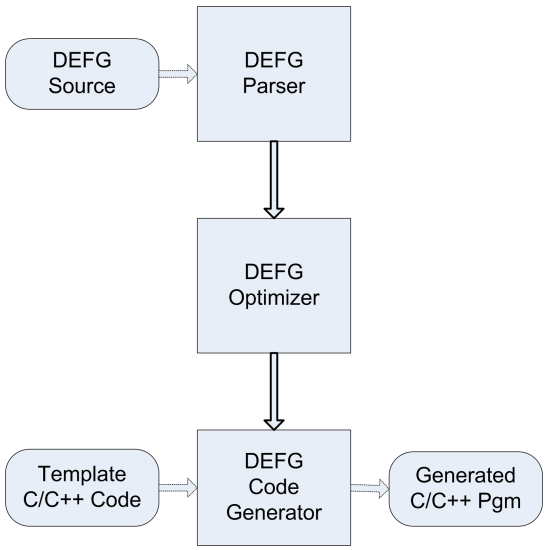

Figure 4.1: DEFG Translation-Steps Diagram

the compilation are stored in an XML document. The resulting XML document, also having been stored in a text file, is substantially updated by the DEFG Optimizer. The optimized XML document is then used by the DEFG Code Generator, along with a C/C++ code template. The XML is parsed by the code generator, processed, and the final C/C++ program is written out. This program, stored in a normal text file, is then further processed as a standard C/C++ program. When an error is encountered by our translator, it stops processing the input, writes error text, and terminates. The additional step showing compilation by a standard compiler is not shown in Figure 4.1. Programs generated by DEFG have the potential to run on mobile processors, as discussed in the Section A.6 of the Appendix.

4.3.1.1 The DEFG Parser

The DEFG parser is written in the Java version of ANTLR 3 [11]. The full DEFG grammar for our translator is shown in the Section B.6 of the Appendix. Our DEFG grammar definition, used by ANTLR, contains embedded Java statements. These Java statements output the XML snippets that, in total, form the resulting XML document. The XML document contains a full description of the parsed DEFG pro-

gram, stored in an easy-to-process XML form. Our parser detects simple DEFG syntax errors, but does not do any verification of references between DEFG statements. These more complex checks are done by our optimizer.

We now focus on the parser inputs and outputs. The input used for the sample output example below is shown in Chapter III, Figure 3.1. Figure 4.2 shows the generated XML output. Some XML nodes and attributes have been omitted and replaced with ellipsis ("...") to keep the size of the figure manageable. In the figure, line 1 is a comment that notes the translation date and the name of the input file. The next line provides the application name, *sobel*; the *main* XML attribute marks this as a program to create an application and not a C/C++ function.

XML was chosen for the intermediate abstract syntax tree format because, in Java, XML documents can be easy to create and relatively easy to parse. In addition, this generated XML document is stored in a simple text file and it is easily viewed from any number of tools. The readability of XML is sometimes disputed, but it is humanly readable and having the abstract parse tree visible as an XML document made the parser and optimizer debugging somewhat easier.

Lines 3 through 5 show that three integer variables are defined. Lines 6 through 8 define the GPU devices to be used. In this case, the GPU selected will be the first device that the OpenCL run time presents, because of the "$*$" setting. The OpenCL kernel to be used, *sobel_filter*, is defined in lines 9 through 11. Lines 12 and 13 define the buffers to be used, *image1* and *image2*, and their respective sizes. The call to the *init_input()* function is defined next, in lines 14 through 19. Note the included references to the *image1*, *Xdim*, *Ydim*, and *BUF_SIZE* buffers and variables. Each buffer or variable has an associated *mode* attribute. These mode settings are critical for the correct operation of the DEFG optimization. The code generated from lines 20 through 23 will cause the *sobel_filter* kernel to be executed on the selected GPU. These lines will be further decorated by the optimizer; as shown below. Finally,

```
01. <!-- built with V09: 2014/08/29 07:24:05 from sobel.txt -->
02. <application name="sobel" main="y">
03. <variable type="integer" alloc="y" name="Xdim" value="0"/>
04. <variable type="integer" alloc="y" name="Ydim" value="0"/>
05. <variable type="integer" alloc="y" name="BUF_SIZE" value="0"/>
06. <gpu name="gpuone">
07. <device name="*"/>
08. </gpu>
09. <kernel name="sobel_filter" module="SobelFilter_Kernels">
10. <parm>[[ 2D,Xdim,Ydim ]]</parm>
11. </kernel>
12. <buffer type="integer" name="image1" size="Xdim" dim2="Ydim" ... />
13. <buffer type="integer" name="image2" size="Xdim" dim2="Ydim" ... />
14. <call name="init_input">
15. <use_buffer name="image1" mode="in"/>
16. <use_buffer name="Xdim" mode="out"/>
17. <use_buffer name="Ydim" mode="out"/>
18. <use_buffer name="BUF_SIZE" mode="out"/>
19. </call>
20. <execution mode="single" kernel="sobel_filter" ...>
21. <use_buffer name="image1" mode="in"/>
22. <use_buffer name="image2" mode="out"/>
23. </execution>
24. <call name="disp_output">
25. <use_buffer name="image2" mode="in"/>
26. <use_buffer name="Xdim" mode="in"/>
27. <use_buffer name="Ydim" mode="in"/>
28. </call>
29. </application>
```

Figure 4.2: Sample XML Output From DEFG Parser

lines 24 through 28 define that the *disp_output()* function be run on the CPU. The application end is marked in line 29.

4.3.1.2 The DEFG Optimizer

For clarity, we will now start exclusively using the term "tree" to denote the XML document holding the DEFG program's abstract syntax tree. The DEFG optimizer is a Java program that has three basic purposes: (1) it looks for and reports DEFG coding errors not detectable by the parser, (2) it decorates the tree with cross reference and optimization information, and (3) it reforms the tree branches by relocating selected requests to move GPU buffers to optimal tree locations. The decorating of the tree with cross reference information is critical for the correct functioning of the DEFG code generator. The code generator handles one tree branch (DEFG statement) at a time and requires that the tree has all of the information needed for a

```
...
20. <execution kernel="sobel_filter" mode="single" ...>
21. <use_buffer name="image1" mode="in "arg="0" move="toDev" type="integer" .../>
22. <use_buffer name="image2" mode="out" arg="1" move="none" type="integer" .../>
23. </execution>
...
```

Figure 4.3: Sample XML Output Snippet From DEFG Optimizer

single statement's code generation included on each tree branch. This requirement for complete information on each branch has made the code generator, discussed below, more straight-forward.

DEFG optimization occurs in two forms. The first form applies to all DEFG call, execute, and multi_exec statements. The in, out, and inout options, on the associated variables and buffers, are used to determine when the contents of a given variable or buffer need to be transferred between the CPU and GPU. These transfers are only performed if the given option setting indicates the need to update the data from the CPU or GPU. Figure 4.3 shows a snippet of the decorated tree for the execution of the *sobel_filter* kernel. Lines 20 through 23 correspond with the same lines in Figure 4.2. We can see that lines 21 and 22 now contain additional information. In particular, Figure 4.3 shows that more information, such as the argument count, data type, and required-movement setting, is present. The *move* attribute setting of *toDev*, present on line 21, is optimization information that will inform the code generation when to actually transfer the given buffer.

The second form of DEFG optimization is the relocation of certain buffer movement operations to locations outside of loops. When a given call, execute, or multi_exec statement accesses variables or buffers that are not modified inside a given loop, the request for the data is moved to a tree location that precedes the loop. This movement of certain requests prevents unneeded and repetitive data movement operations from being executed at run time.

The DEFG loop/while statements are limited and cannot be embedded inside other

loops. There are several reasons for this DEFG limit, but the main one relates to the optimizations performed. The optimizer has to "understand" the DEFG looping and we found that the optimization could be performed reasonably, if we limited the DEFG loop/while statement power by forbidding embedding. To be clear, the DEFG sequence statement can be embedded in loop/while statements; a loop/while just cannot be embedded in another loop/while. More will be said about these limits in Chapter VI, on future research. We believe there is a better approach, based on using a less *static* optimization technique.

The implementation of these two optimizations in DEFG has given the DEFG-generated code good performance. The performance is similar to that achieved with hand-written C/C++ OpenCL applications, at least in the applications we tested. We believe that these optimizations are one of the anchor facilities of DEFG; they help make DEFG a viable application generation approach by providing good run-time performance.

4.3.1.3 The DEFG Code Generator

The generation of the DEFG application code is a complicated, non-trivial endeavor. It is so for a number of reasons. The code generator has to create C/C++ code that produces the desired result, on multiple operating system platforms.[6] We support OpenCL GPUs from different vendors. In addition, the code generator has to produce code for two modes of operation: single GPU execution and multiple GPU execution. It is also tasked with generating code that a person will be comfortable reading. Although OpenCL is a defined standard, it has multiple implementer-defined features and options [70]. Where possible, we have used the simplest, common-denominator approaches; when these caused errors or produced poor performance, we used more advanced, and sometimes environment-specific, approaches. The developer is shielded

[6]We have limited our formal testing to Linux and Windows.

from all these issues by DEFG; it handles the many details of developing C/C++ code for GPU use.

A high-level diagram of the DEFG Code Generator is shown in Figure 4.4. The DEFG Code Generator is written in C/C++, executes on Windows, and uses the TinyXML2 [91] library to parse its input XML file. An early step in its processing is accessing the text file which holds the DEFG template C/C++ code. The DEFG template is copied directly to the file containing the generated code. This template provides the "boiler plate" code that initializes the run-time resources and selects the one or more devices for run-time processing; it also contains special markers. These markers are placeholders that are replaced by the unique code generated for each DEFG translation.

Speaking abstractly, the first major step in the code generation is producing the GPU device selection and device management code. Here, the OpenCL environment and command queues are created, the GPU needed kernels are loaded, and the required CPU memory buffers are allocated. The CPU buffer memory is obtained with C-style *malloc()* calls and is of a fixed size. This size is changeable, at run time, by the DEFG_MAX_BUF environment variable. Once allocated, the buffers are segregated and managed by the generated DEFG code.

From a high level, after the environment is established, the steps outlined in the middle box of Figure 4.4 are performed. These operations generate the code for the DEFG statements defined in the input tree. After these operations are completed, the code to release all of the resources and terminate the application is generated. After the code generation is completed, the output file is available for compilation by a standard compiler.

In the remainder of this section, we will focus on the basic operations outlined in the middle box of Figure 4.4. These operations consist of: (1) GPU-Oriented Operations, (2) CPU-Oriented Operations, and (3) Loop/While Statement Operations.

The optimized syntax tree is processed one branch at a time, beginning with the root. For each non-declare statement in the tree, a single operation type from one of these three groups is performed. We will summarize each of these operations below. We note, in advance, that the multiple-GPU support buffer options: halo, multi, and nonpartable are handled in the *buffer movement* code generation and in the *execute NDRange* code generation. Through this layering and abstraction, these complex buffer options are handled without greatly impacting the more basic DEFG code generation for marshaling buffers, making OpenCL API calls, and handling errors.

The GPU-oriented code generation operation is made up of two phases. The first phase is only done when required by the *move* attributes in the input tree. It consists of generating the OpenCL code to create buffers and to transfer variables and buffers. The OpenCL GPU device buffers are only created when actually used. They are transferred (actually copied) when their contents are valid on the CPU and not the GPU. This can happen if the generated CPU-side code updates a variable or buffer on the CPU. It is the job of the DEFG optimizer to manage this coordination of transfers.

The second phase of this operation consists of generating the code to invoke the requested kernel or call the BLAS library. With the GPU kernel invocations, each selected GPU has the kernel arguments set and the kernel started via the OpenCL *clEnqueueNDRangeKernel()* API call. Code is generated to describe any detected error indications.

The CPU-oriented code generation has four options, depending on the DEFG statement being handled. For a code statement the text between the "[[" and "]]" delimiters is inserted directly into the generated code. The use of code statement, called a "morsel," is discussed in the User's Guide, Section A in the Appendix. The set statement's code generation consists of simply copying the associated value into the referenced scalar variable. This statement may seem very limited in power. However,

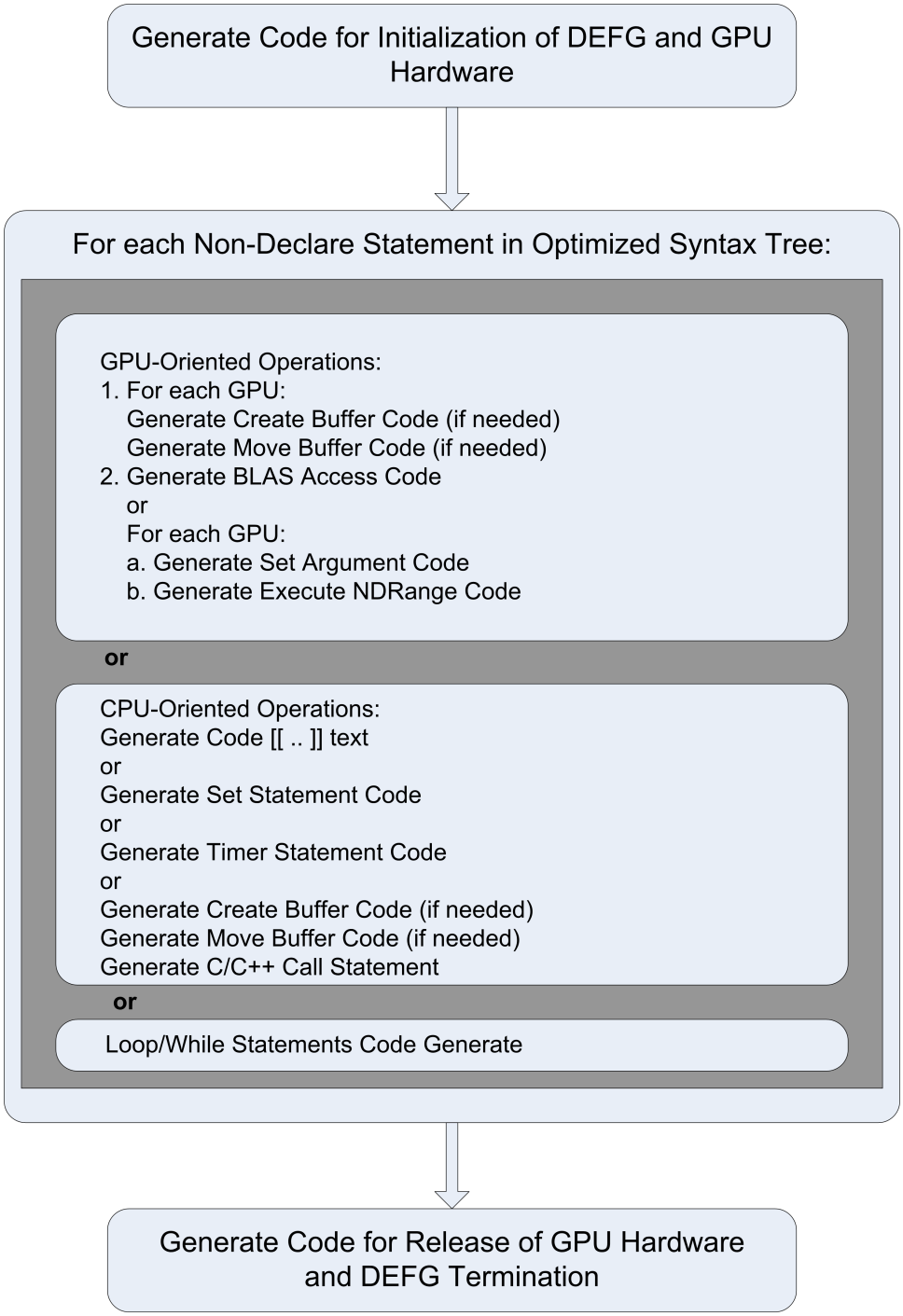

Figure 4.4: DEFG Code Generation Diagram

the DEFG optimizer is aware of its actions. This causes any needed transfer operations to be generated, if this field is then referenced from a GPU. The DEFG timer statements cause the generation of code to start, stop, and read the DEFG-provided CPU timer.

The most interesting CPU-oriented code generation involves the DEFG call statement. The transfer code is generated only when required by the *move* attributes in the input tree; it transfers (copies) variables and buffers from the GPU to the CPU. After these optional actions are completed, the call to the defined C/C++ function is made. The DEFG call is part of the DEFG optimizer processing; CPU variables and buffers are only updated with GPU content when the GPU content has been updated.

The loop and while operation generates the C/C++ code to implement the DEFG loop and while statements. Due to some unwanted behavior in the way C/C++ local variables are scoped, we use the C/C++ if and goto statements to implement this DEFG looping capability.

In Figure 4.5, we show a snippet of the C/C++ code generated for the DEFG execute statement shown in Figure 3.1, line 10. This snippet causes execution of the *sobel_filter* kernel with *image1* as input and *image*2 holding the resulting filtered image.

This snippet has had many of the less important C/C++ lines and parameters removed; these removals are marked with ellipsis ("..."). The generated code includes comments to facilitate the code being "human readable." The first eleven lines show the conditional execution of a buffer creation on the GPU device. The next six lines show the transfer of the *image1* to the GPU device and the setting of the first two arguments for the kernel execution on the device. The last two lines show the staging of the kernel's execution in the OpenCL command queue.

This chapter has described the DEFG translator. Our design splits the trans-

```
// *** KERNEL: <<<< sobel_filter >>>>
// *** WRITE FIELD: image1(in-toDev)
...
if (defg_create_1 == 0) {
  defg_create_1 = 1;
  ...
  defg_buffer_image1[0] = clCreateBuffer(defg_context, ..., &defg_status);
   if (defg_status != CL_SUCCESS) {
     // error handling, etc
     ...
   }
}
...
defg_status = clEnqueueWriteBuffer(defg_Queue[0], defg_buffer_image1[0], ...);
...
defg_status = clSetKernelArg(defg_sobel_filter[0], 0, ...);
...
defg_status = clSetKernelArg(defg_sobel_filter[0], 1, ...);
...
// *** EXECUTION: sobel_filter
defg_status = clEnqueueNDRangeKernel(defg_Queue[0], defg_sobel_filter[0], ...);
if (defg_status != CL_SUCCESS) {
  // error handling, etc
  ...
}
...
```

Figure 4.5: C/C++ Snippet for *sobel_filter* Kernel Execution

lation processing into three steps, which works well, as it allows us to separate the complexities inherent in optimization and code generation. The middle step, the optimization step, really keeps the error detection and high-level optimization away from the down-and-dirty OpenCL-oriented code generation. For DEFG to support the CUDA product, which has similarities to OpenCL, but also has differences, a new code generator would need to be provided; the parser and optimizer would remain largely unchanged.

CHAPTER V

NEW AND DIVERSE DEFG APPLICATIONS

In order to demonstrate the power and viability of DEFG, we designed and implemented a set of new applications using DEFG and OpenCL. Some of these new applications are based on our existing SOBEL and BFS applications, discussed in Chapter III. The others are entirely new implementations.

Our first new applications consist of two image filters. The first filter is an enhanced Sobel operator filter for multiple-GPU operation and the second is a median filter, functioning in single-GPU and multiple-GPU modes. We refer to our filtering applications as SOBEL and MEDIAN and may sometimes add a suffix to indicate additional functionality. For example, SOBELM is our multiple-GPU version of SOBEL.

In the next new application, multiple-GPU support is added to the existing DEFG-based BFS application. Here, parallel prefix scan is applied in a novel and interesting way to dynamically manage the buffers passed between the different GPU devices. Prefix scan's allocation of the buffer space to individual GPU threads makes it possible to manage the shared buffers without the costs of slow, atomic locking on buffer structures. This application is referred to as BFSDP2GPU.

The new RSORT application is our proof-of-concept implementation of roughly sorting. Roughly sorting is used when the data to be sorted is already partially in sequence. This sorting approach scans the data, computes a measure of disorder and then breaks the data into segments, which are individually sorted [9, 10].

Our final new DEFG application implements the Altman iterative matrix inversion algorithm [7, 8] and is referred to as IMI. In this application, DEFG forms an interface layer between the application's logic and the Basic Linear Algebra Subprograms (BLAS) numerical library [2].

Each of our new applications is described, and analyzed, in this chapter. The analysis for each application varies based on the goals for each. For example, our image filtering applications show the ease with which DEFG can provide the CPU-side code for high-performance, multiple-GPU applications. We carefully measure the respective run-time performances. On the other hand, the iterative matrix inversion application shows the ability of DEFG to be expanded to use existing GPU libraries. It is not related to multiple-GPU operations and, hence, we are not as interested in run-time performance. We are more concerned with the sizes and types of matrices that can be processed.

In *Designing Scientific Applications on GPUs* [23], Raphael Couturier talks about the implementation of GPU applications. He groups them into categories by image processing, optimization, numerical applications, and adds software development.

Our mix of applications directly map to three of Couturier's four categories. The image filtering applications implement the Sobel operator and the median filter. Touching on the Couturier topic of numerical applications, in particular solving sparse linear systems, is the DEFG iterative matrix inversion application. This DEFG environment, with its multiple GPU support, touches the recurring Couturier topic of enabling the support of applications over multiple GPUs. These applications provide a good mixture of GPU solutions that demonstrate the power of DEFG.

The run-time results were obtained using the Hydra server at the University of Colorado Denver's Computer Science and Engineering Department, with a few exceptions. In these exceptions, results were obtained using other hardware and we clearly note these exceptions.

5.1 Application: Image Filters

5.1.1 Problem Definition and Significance

Image filters can be used to enhance the quality of images, as well as, to locate the edges contained in images. In this section, two DEFG image applications will be discussed: *Median* filtering and the *Sobel* operator, both within the context of single-GPU and multiple-GPU operations. Single-GPU operation refers to executing the DEFG application on a single GPU; likewise, multiple-GPU operation refers to executing the DEFG application on multiple GPUs. Having the option to execute a given DEFG application on multiple GPUs provides the potential to obtain results more quickly and to solve larger problems.

5.1.2 Related Work

The Sobel operator can be used for image edge detection and is designed to approximate the gradient value at the specific pixel being processed [84]. A computed gradient value that is relatively high represents a hill, slope or wall, that is, an edge. The Sobel operator for edge detection was first described in an unpublished 1968 article from the Stanford AI Lab by Irwin Sobel and Jerome Fredman: *A* 3×3 *Isotropic Gradient Operator for Image Processing* [21, 86]. This operator computes an approximation of two gradients using a pair of 3×3 convolution masks, one mask for the horizontal estimation of the gradient and one for the vertical [93]. There are other similar operators including the *Robinson* operator and the *Kirsh* operator, differing in the weights used in the convolution mask [87].

The median filter was outlined by Tukey, in 1971, for use in signal smoothing and it is a special case of a rank filter [43, 87]. It is commonly used to enhance the quality of an image by forcing points with highly varied intensities to be similar to their neighbors [84]. The median value of a given pixel's neighbors is computed and

is used to replace the pixel's value. The processed pixel is normally centered in its neighborhood. In our median filter work, neighborhoods of 3×3 and 5×5 pixels are used. This type of filtering has many uses, including that of noise removal, which is an instance of "smoothing." Variations in the median filter include increasing the size of the neighborhood and changing the shape of the neighborhood.

Over time, numerous improvements to the Sobel operator have been proposed. Since the Sobel operator only uses a 3×3 mask, it is very sensitive to noise in the image. Ma, et al. [56] proposed an improved Sobel algorithm using a median computation, based on a larger 5×5 mask. Using color images, Wesolkowski, et al. [96] demonstrated that the Sobel operator shows good edge detection results, as compared with other similar operators. Wang [94] showed how a filter based on the Sobel operator can be used as an integral component of a vehicle identification system where the Sobel operator is used to highlight the contour of the vehicle against the environment around the vehicle. In our work, we use the Sobel operator in its original form, with a 3×3 mask.

5.1.3 Approach to Research

We begin our DEFG Application Chapter with the Sobel operator and median filter applications because they represent the class of straight-forward GPU problems, where a single kernel is called once and there is significant locality of memory reference. This locality-of-reference characteristic tends to provide for good GPU performance without having to create a complex kernel or set of kernels. The median filter presents an application that is similar to the Sobel operator, but is more computationally intense, especially when using neighborhoods of size 5×5. This increased computational intensity is useful when exploring the performance characteristics of the multiple-GPU DEFG versions.

Our single-GPU Sobel and median filter implementations show that DEFG han-

dles single-invocation OpenCL kernels with minimal developer effort, and with good performance. When performing multiple-GPU DEFG processing, the Sobel and median filters provide more complex usage cases since the images sent to each GPU must overlap slightly and *DEFG needs to manage this data overlap.* A goal of DEFG is to enable multiple GPU operations with very limited changes to DEFG code. Our expectation is that when this DEFG capability is used appropriately, the OpenCL kernels will not require modifications.

A few words of caution concerning the DEFG multiple-GPU capabilities: of course, the OpenCL kernels must have been implemented using GPU coding techniques that permit multiple GPU operations. DEFG makes the utilization of multiple-GPU-capable kernels less work for the developer, but it does not provide any "Holy Grail" (see Shen [83]) for automatic parallelization of existing algorithms and code.

DEFG provides a number of design patterns that support multiple-GPU processing. Multiple-GPU image applications can use DEFG's *Overlapped-Split-Process-Concatenate* design pattern to handle those neighborhood masks that are at the image edges. Since the Sobel operator and median filter require at least a 3×3 neighborhood mask, some image content must be repeated so that the edge pixels at the image-split locations are present where they are needed. Each GPU must have in memory the pixels it requires. When these separately processed images are later concatenated to reform the final image, this overlapped image content must be taken into consideration. The Overlapped-Split-Process-Concatenate design pattern handles this overlapping in a generalized and easy-to-use manner. The DEFG buffer halo option facilitates this design pattern; it provides DEFG with the needed information to correctly split and reform images. The halo option is discussed in Chapter IV and in the User's Guide, Appendix Section A.

5.1.4 Additional Background

5.1.4.1 Single-GPU DEFG Sobel Operator Application

In Section 3.4, we discussed our DEFG-based version of the Sobel operator, utilizing the Sobel kernel from the AMD Application SDK 2.8 [1]. The CPU-side OpenCL code was replaced by DEFG-generated code. This DEFG Sobel implementation produced exactly the same results as the SDK version and showed very similar run-time performance. Figure 5.1 shows the results of running the Sobel operator with the DEFG. The image on the left is the input image and the image on the right shows the filtered results. The edges of the highway lane markers, tree tops, and cloud banks are delineated and highlighted. The filter has the effect of highlighting changes in color.

5.1.4.2 Single-GPU DEFG Median Filter Application

Our filter application research efforts involve using the DEFG multiple-GPU support to produce enhanced application performance characteristics. However, it became clear that the Sobel operator was not computationally intense, in terms of run time, relative to the time it takes to move the image to and from the GPU; a large portion of the GPU-based Sobel Filter execution time was consumed moving the image to and from the GPU.

Figure 5.1: Sobel Operator Performed with DEFG: Before and After Images

Figure 5.2: Median 5×5 Filter Performed with DEFG: Original, Noised-Added, and After-Processing Images

Therefore, a second filter application was added: the MEDIAN filter. The median filtering was supported in both 3×3 and 5×5 neighborhood versions. This DEFG CPU-side program to support the median filter was very similar to the Sobel version. The significant differences between these two filter applications were in the corresponding OpenCL kernel code. Figure 5.2 shows the results of having run the median 5×5 filter with DEFG. The left-most image is the original image. The middle image consists of the original image with approximately 2 percent of the pixels replaced by black "noise" pixels. The right-most image shows the results of running the median application to clean up the noise. A careful examination of the resultant image shows that the noise was removed, but the quality of the image suffered as a result. One can see that, in the right-most image, the blades of grass are somewhat blurred and the clouds are less clear; some of the quite sharp cloud edges have been dulled.

Figure 5.3 shows the Median filter kernel code, with a 3×3 neighborhood. Lines 12 through 20 moved the neighborhood pixels into the *sort_buf* array. Lines 21 through 29 ordered the elements in the array and line 30 copied the middle pixel value, the median, to the pixel being processed.[1] The larger 5×5 version of this kernel was very similar to this 3×3 version; the major difference was that 25 pixels were sorted and

[1]The sorting algorithm used here was not highly optimized, helping ensure that this median filter kernel is more computationally intense than our Sobel operator kernel.

the middle element of the resulting 25-item array was chosen as the new pixel value.

We note that these two kernels from our single-GPU work were used in our multiple-GPU work without modification. This was in line with our aim of, where possible, obtaining DEFG multiple-GPU support without having to recode the OpenCL kernel or make substantial changes to the DEFG code. In the best case, we expect that the changes to the DEFG code are simple changes to the declarative statements and the simple switching of execute statements to exec_multi. The algorithms and kernels that are suitable for this relatively easy switch to multiple-GPU processing tend to be self-contained and have simple data access patterns. Some computer scientists describe these as embarrassingly parallel, perfectly parallel, or pleasingly parallel algorithms [98]. Embarrassingly parallel or not, applications that use multiple GPUs still need the infrastructure code present to manage the many GPUs and the many buffers. DEFG provides this infrastructure code.

5.1.4.3 Multiple-GPU DEFG Filter Applications

The DEFG Overlapped-Split-Process-Concatenate design pattern and the associated halo option are used here to permit the filtering to be performed on two separate GPUs, with each GPU processing half of the image. As noted previously, the complicating factor is the need for any pixel processed to have its neighbors present. This means that a small section of the image, the parts of any halo, may need to be present on both GPUs.

The code used to execute the Sobel operator with a single GPU was previously shown in Figure 3.1. For comparison, the DEFG code used to execute the Median 5×5 filter on two GPUs is shown in Figure 5.5. A review of the two DEFG programs shows three significant changes beyond the kernel name change. First, the declare gpu, on line 5, has been changed to use the all option. Next, the halo (2) option has been added to the buffer declarations on lines 7 and 8. Finally, line 10 has been

changed from an execute statement to a multi_exec statement. The all option allows DEFG to use all the GPUs attached to the CPU node; in the case of the UCD Hydra server, this was two GPUs. The addition of the halo (2) option notifies DEFG that when running with multiple GPUs, DEFG must manage the overlapped halo area. The (2) denotes that the overlap is 2 units; as this is a 2-dimensional data structure, i.e., an image, the 2 denotes two lines. The use of multi_exec causes DEFG to execute kernel code on all the selected GPUs. These changes are sufficient to allow for the generation of C/C++ code utilizing more than one GPU, even though the partial image sent to each GPU must be overlapped. The overlap management is handled entirely by DEFG.

Figure 5.4 shows a schematic representation of a 6 pixel $\times$ 6 pixel image that might be processed by DEFG with two GPUs. In the schematic, the pixels prefixed

```
01. __kernel void median_filter(__global uint* inputImage,
02.                              __global uint* outputImage) {
03.   uint sort_buf[9];
04.   uint h;
05.   int i, j;
06.   uint x = get_global_id(0);
07.   uint y = get_global_id(1);
08.   uint width = get_global_size(0);
09.   uint height = get_global_size(1);
10.   int c = x + y * width;
11.   if( x >= 1 && x < (width-1) && y >= 1 && y < height - 1) {
12.       sort_buf[0] = inputImage[c - 1 - width];
13.       sort_buf[1] = inputImage[c - width];
14.       sort_buf[2] = inputImage[c + 1 - width];
15.       sort_buf[3] = inputImage[c - 1];
16.       sort_buf[4] = inputImage[c];
17.       sort_buf[5] = inputImage[c + 1];
18.       sort_buf[6] = inputImage[c - 1 + width];
19.       sort_buf[7] = inputImage[c + width];
20.       sort_buf[8] = inputImage[c + 1 + width];
21.       for (i=0; i < 9; i++) {
22.           for (j=i; j < 9; j++) {
23.               if (sort_buf[i] > sort_buf[j]) {
24.                   h = sort_buf[i];
25.                   sort_buf[i] = sort_buf[j];
26.                   sort_buf[j] = h;
27.               }
28.           }
29.       }
30.       outputImage[c] = sort_buf[4];
31.   }
32. }
```

Figure 5.3: Kernel Code for Median Filter with 3×3 Neighborhood

Schematic Image:					
B61	B62	B63	B64	B65	B66
B51	B52	B53	B54	B55	B56
B41a	B42a	B43a	B44a	B45a	B46a
A31b	A32b	A33b	A34b	A35b	A36b
A21	A22	A23	A24	A25	A26
A11	A12	A13	A14	A15	A16

Figure 5.4: Image Schematic Showing Overlap with 2 GPUs

Table 5.1: Execution Times on Hydra Server, in Milliseconds

Filter Name	Write to GPU ms	Execution GPU ms	Read from GPU ms	Total GPU Run ms
SOBEL	1	1	1	3
SOBELM	2	1	1	6

with "A" are in the sub-image for GPU_1 processing and the pixels prefixed with "B" are in the sub-image for GPU_2 processing. In this hypothetical example, a Sobel operator is to be run for the non-edge pixels in the full image. So, the halo (1) statement option would be used since the Sobel operator uses a 3×3 mask. The use of this halo option would have the effect of copying pixels A11 ... A36b *plus* B41a ... B46a to GPU_1 and A31b ... A36b *plus* B41a ... B66 to GPU_2. By copying the additional pixels, each GPU would have all of the values needed to correctly execute the Sobel operator. When the image buffers are later transferred back to the CPU, the image overlap area is managed by DEFG, so that the correct pixels appear in the final image.

```
01. declare application median5
02.  declare integer Xdim (0)
03.          integer Ydim (0)
04.          integer BUF_SIZE (0)
05.  declare gpu gpugrp ( all )
06.  declare kernel  median5_filter Median5Filter_Kernels   ( [[ 2D,Xdim,Ydim ]] )
07.  declare integer buffer image1 ( Xdim Ydim ) halo (2)
08.          integer buffer image2 ( Xdim Ydim ) halo (2)
09.  call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
10.  multi_exec run1 median5_filter ( image1(in) image2(out) )
11.  call disp_output (image2(in) Xdim (in) Ydim (in) )
12. end
```

Figure 5.5: DEFG Code to Execute the 5×5 Median Filter

5.1.5 Experimental Results

5.1.5.1 Multiple-GPU DEFG Filter Performance

Table 5.1 shows the SOBEL and SOBELM applications' run-time performances, obtained from executing these applications on Hydra.[2] SOBEL is the name given to DEFG single-GPU version of the Sobel operator and SOBELM is the name for the DEFG multiple-GPU version. Clearly the single-GPU version, which used 3 ms, was faster than the 2-GPU version, which used 6 ms. Equally clear is that the time needed to move the image to and from the GPU, 2 ms in the case of SOBEL, was greater than the time needed for the execution of the kernel, in this case 1 ms. The SOBEL application, directly based on the Sobel application from the AMD SDK, uses more time moving the image than processing it.

In an effort to understand the unimpressive performance with our SOBELM 2-GPU application, we manually inserted additional timers into the DEFG-generated code and performed additional DEFG logging. These steps gave us basic run-time profiles for SOBEL and SOBELM. After reviewing these run-time profiles, we postulated several potential explanations for the observed performance: (1) the OpenCL multi-GPU operations have generally poor performance; (2) the image being used for filter testing was too small; (3) the Sobel operator was not computationally intense enough for multi-GPU use; (4) the DEFG approach to multi-GPU kernel execution

[2]The SOBELM run time is not equal to the sum of the partial totals due to timer limits.

Table 5.2: Images Used with Filter Application Testing

Image Name	Description	Image Dimensions	Image Size in bytes	Source
BUFLO	Downtown Buffalo	1714×1162	5,977,382	Mike Boncaldo
FIELD	Rice field	512×512	786,486	freedigitalphotos
IMG1000	Colored box	1000×1000	3,000,054	generated
IMG5000	Colored box	5000×5000	75,000,054	generated
IMG7000	Colored box	7000×7000	147,000,054	generated
IMGHUGE	Colored box	22000×22000	1,452,000,054	generated
ROAD	Road to the clouds	170×153	78,390	freedigitalphotos

on the GPU performed poorly; and, (5) the DEFG approach to multi-GPU buffer movement to and from the GPU performed poorly.

Our previous experiences with other OpenCL applications, in particular the mining of digital coins [54], showed that the OpenCL support for multiple-GPU operations was "solid" and showed notable high performance. So we discounted the first explanation. In order to test the view that the filter test image was too small, we obtained and experimented with a set of larger images. Our full set of images is listed in Table 5.2. The BUFLO [3], FIELD[4], and ROAD[5] images were obtained from Google Images and the World Wide Web. This table assigns a unique name for the images, along with presenting each image's characteristics.

As part of testing the explanation that the SOBEL operator is not computationally intensive enough, we added the median filter to our set of DEFG applications. We produced median filter applications in two forms. The first form used a 3×3 neighborhood with 9 pixels and the second form used a 5×5 neighborhood with 25 pixels. The second form, with its 25 pixel neighborhood, was more computationally intense due to the median filter's sorting step. Using these additional images and filters, we performed a series of new experiments on Hydra.

The results of these experiments are shown in Table 5.3. The first column lists

[3]http://www.mikeboncaldo.com/photos - Used with permission.
[4]http://www.freedigitalphotos.com - Used as per agreement.
[5]http://www.freedigitalphotos.com - Used as per agreement.

Table 5.3: Run Times for Various Images

Image Name	Filter Name	Average Execution Seconds
FIELD	SOBEL	0.003
	SOBELM	0.006
	MEDIAN	0.003
	MEDIANM	0.006
	MEDIAN5	0.009
	MEDIAN5M	0.008
BUFLO	SOBEL	0.018
	SOBELM	0.023
	MEDIAN5	0.058
	MEDIAN5M	0.048
IMG1000	SOBEL	0.010
	SOBELM	0.013
	MEDIAN5	0.023
	MEDIAN5M	0.019

Image Name	Filter Name	Average Execution Seconds
IMG5000	SOBEL	0.215
	SOBELM	0.240
	MEDIAN5	0.542
	MEDIAN5M	0.412
IMG7000	SOBEL	0.420
	SOBELM	0.479
	MEDIAN5	1.062
	MEDIAN5M	0.794
IMGHUGE	SOBEL	failed -4
	SOBELM	4.650
	MEDIAN5	failed -4
	MEDIAN5M	7.775

the image used, the second provides the name of the filter used, and the last column shows the average run times for three Hydra executions of the named image with the given filter. The characteristics of each image are given in Table 5.2.[6]

We note that the execution times for SOBEL and SOBELM compared to MEDIAN and MEDIANM, for the FIELD image, were very similar. Our experience has been that the Sobel operator and the 3×3 Median filter showed similar run times, no matter which image was used. Therefore, we omitted the MEDIAN and MEDIANM execution times in the remainder of Table 5.3. Instead, we focused on the SOBEL/SOBELM and MEDIAN5/MEDIAN5M results.

The FIELD image execution times showed that MEDIAN5 took about 3 times longer to execute as compared to SOBEL and that MEDIAN5M took less than 2 times the time of SOBELM. Clearly, the impact of using two GPUs with the MEDIAN5M median filter application was different from that with the SOBELM Sobel operator application. Comparing the MEDIAN5 and MEDIAN5M results, using the FIELD

[6]The FIELD image from Table 5.2 is shown in Figure 5.2, on the left, and the ROAD image from Table 5.2 is shown in Figure 5.1, also on the left.

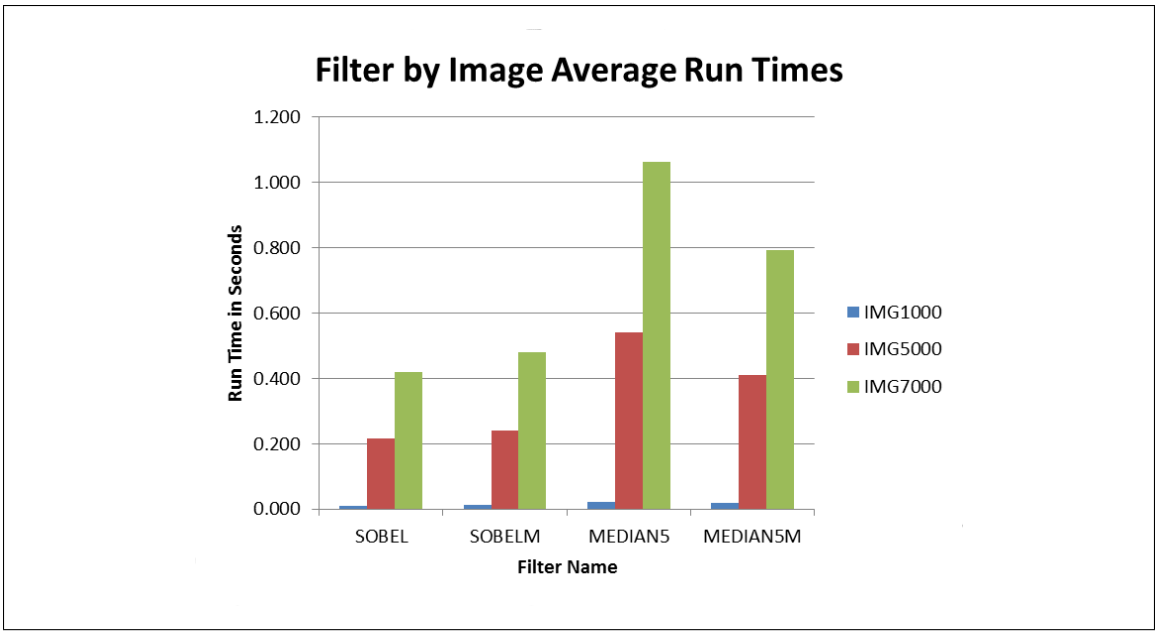

Figure 5.6: Plot of Filter by Image, Average Run Times

image, showed that the MEDIAN5M application was slightly faster, having used 0.008 seconds versus 0.009 seconds for single-GPU MEDIAN5.

The average run times for the IMAGE1000, IMAGE5000, and IMAGE7000 images, plotted with applications SOBEL, SOBELM, MEDIAN5, and MEDIAN5M, are displayed in Figure 5.6. We note that the IMAGEHUGE is not included in this plot, as it was too large to be processed by the single-GPU applications. This plot shows that multiple-GPU MEDIAN5M was faster than single-GPU MEDIANM for IMAGE5000, of size 5000×5000, and for IMAGE7000, of size 7000×7000. For the higher computational intensity median 5×5 filter, the multiple-GPU processing was faster when using the larger images.

5.1.5.2 Profiled Multiple-GPU Performance

In an effort to further understand the computational intensity issue, we ran additional experiments with the DEFG OpenCL logging facility engaged. These logs provided OpenCL API call-profiling information. In particular, the times needed to write the

Table 5.4: Detailed Run Times for BUFLO Image

Filter Name	Write to GPU ms	Execution GPU ms	Read from GPU ms	Total GPU Run ms
SOBEL	6	5	3	19
SOBELM	7	5	7	23
MEDIAN5	6	45	3	59
MEDIAN5M	8	25	8	44

images to the GPU(s), execute the kernel(s), and read the images back from the GPU(s) were logged. Table 5.4 shows the results of these experiments, using the BUFLO image. This data indicates that the median 5×5 filter benefited from 2-GPU execution with an execution time of 25 milliseconds versus a 1-GPU time of 45 milliseconds. It is also clear that the clock time needed to write the images to the GPUs, and read the images back from the GPUs, had increased with 2-GPU operation. For example, the MEDIAN5 write-to-GPU time was 6 ms and the MEDIAN5M write-to-GPU time was 8 ms. We concluded that with this implementation of DEFG, the Sobel operator is not computationally intense enough to benefit from 2-GPU usage. The Sobel operator showed increased write and read times while the execution time did not decrease.

The increase in write and read times was unexpected; our expectation was that these times would drop with 2-GPU operation. A review of the OpenCL documentation, combined with our previous experiences with other OpenCL applications, led us to postulate that the DEFG use of a single operating system thread for all operations might be contributing to the increased run times for the write and read operations. We note that the DEFG-generated code used the OpenCL asynchronous calls and maintained multiple command queues, when executed with more than a single GPU. In order to better understand this 2-GPU unimpressive performance, we wrote a non-DEFG OpenCL test program that used Linux pthreads[7] to provide multiple operating system threads. We then compared the run-time speeds of doing the 2-GPU OpenCL

[7] *pthread* in the name given to a commonly used Linux multiple-threading library.

Table 5.5: Run Times for pthread Experiment

Run Mode	Run Time in ms	Comments
1 Thread	610	DEFG-Style
2 Threads	387	pthread-based

write and read operations using a single-threaded approach against doing the same processing with a pthread-based 2-thread approach.

Both test programs moved 128MB to each of 2 GPUs and then moved the same 128MB back. The test programs were executed three times and the average execution times for the 1-thread and 2-thread write/read operations are shown in Table 5.5. As we had postulated, the single-operating-system-thread run times were longer. The average for 1-thread operations was 0.610 seconds in duration, while the 2-thread operations needed only 0.387 seconds. This simple experiment tends to support our belief that the single-threaded approach to OpenCL buffer movement operations was a performance issue.

Before we leave the topic of 1-GPU and 2-GPU image processing, we note that the IMGHUGE image, which is extremely large at approximately 1.45GB, was processed by the 2-GPU versions of the filter applications and not by the 1-GPU versions. The 1-GPU versions failed with OpenCL error -4, defined by the OpenCL header as:

```
CL_MEM_OBJECT_ALLOCATION_FAILURE
```

Plainly stated, the memory allocation on the single GPU failed due to lack of global memory.

5.1.5.3 Summary of Results

Let us now return to the five potential causes that we previously noted for the unexpected multiple-GPU performance. The causes are re-listed here along with our thoughts and conclusions about the validity of each cause.

1. *The OpenCL multiple-GPU operations have generally poor performance.*

   This is not true; OpenCL multiple-GPU applications have been shown to have good performance. We listed digital coin mining as a valid example of good OpenCL performance with multiple-GPUs [54].

2. *The image being used for filter testing was too small.*

   This was basically true for the smallish FIELD image. The larger images did show the performance benefit of using multiple-GPUs with a computationally-intense filter, such as the median 5×5 filter.

3. *The Sobel operator was not computationally intense enough for multi-GPU use.*

   With our DEFG implementation of Sobel operator for multiple GPUs, this was true.

4. *The DEFG approach to multi-GPU kernel execution on the GPU performs poorly.*

   Our experiments showed that the DEFG approach to multiple-GPU kernel execution did not perform poorly. Comparing the MEDIAN5 and MEDIAN5M kernel execution times showed a drop from 45 ms to 25 ms. This was a speedup of 1.8.

5. *The DEFG approach to multi-GPU buffer movement to and from the GPU performs poorly.*

   Unfortunately, our experiments did show that the DEFG approach to multiple-GPU buffer management did not perform as well as it theoretically could have. We saw increases in run-times for multiple-GPU buffer movements and we attributed this to the DEFG run-time use of a single operating system thread. The separate experiment we ran with pthreads tended to confirm that a multi-threaded approach could be faster.

Our experiments have shown that the multiple-GPU support can be useful when used with workloads that are computationally intense enough to offset the added overhead of the increased multiple-GPU buffer transfer times. It is conceivable that the next major version of DEFG will support pthread-style processing, when multiple GPUs are used.

5.2 Application: Breadth-First Search

5.2.1 Problem Definition and Significance

Breadth-first search is a well-studied graph-theoretic problem, with practical application in shortest path analysis of social networks, in Internet packet routing, in the World Wide Web, and in many other areas [16]. Numerous breadth-first search (BFS) algorithms have been implemented on GPUs, with Harish and Narayanan providing one of the first published GPU implementations [38]. The breadth-first search application discussed in Chapter III is based on this Harish work and it will be used in this section as a comparison basis for our new BFS application.

More recently, Merrill et al. have described an interesting method for managing the intermediate data structures needed in BFS by using *prefix sum* to avoid locking and serialization of data structure items [59]. Doing GPU-based graph processing with large very irregular (LVI) graphs can be challenging, because of the high variation in vertex degree and the sheer volume of the vertex and edge data structures. The storage consumed by a LVI graph's data structures may exceed the memory capacity of a single GPU. The specific problem being addressed here is the implementation of BFS with multiple-GPU support, specifically designed to process LVI graphs, using OpenCL and DEFG. The intent is that DEFG and its design patterns will provide the necessary mechanisms needed to manage the flows of data between the GPUs; by using this support, the GPU application developer can focus on the application's algorithms and processing.

5.2.2 Related Work

There are numerous implementations of BFS on GPUs, utilizing a variety of approaches and algorithms. Since 2007, the performance of these BFS algorithms has improved, sometimes by a speedup of up to 15 [42] . Harish and Narayanan provide one of the first published GPU implementations using NVIDIA's CUDA [38]. This work provides basic GPU-usable algorithms for implementing breadth-first search, single source shortest path, and all pairs shortest path; the performance results obtained are often used as a baseline for judging the performance of improved GPU algorithms. The Harish breadth-first search algorithm processes search nodes in parallel, and is level-synchronous. The algorithm consists of two major segments and, in their implementation, each of these segments becomes a CUDA kernel. Using the two separate kernels provides an automatic trialization barrier, thereby enforcing the required level synchronization. Harish references the previous work by Bader and Madduri as part of the basis for their work.

The Bader and Madduri work covers BFS on the Cray MTA-2 [12]. The Cray MTA-2 architecture, with its ample global shared memory, is thread-oriented and has very fast context switch times between threads, making it somewhat similar to the GPU architectural model. These characteristics allow the MTA-2 to not depend upon memory caching, but instead, much like GPUs, to switch to a different thread when memory stalls occur. This architectural similarity makes the algorithm provided by Bader a reasonable starting point for GPU-based BFS processing. The level-synchronized BFS algorithm published by Bader uses the zero-overhead synchronization provided by the MTA-2.

Whereas the Bader work and the Harish work are aimed at getting results within particular environments, Dehne and Yogaratnam provide an overview of the advanced programming techniques, such as packing multiple variables values into one 32-bit integer, that can be used with both CUDA and OpenCL to achieve better performance

[26]. They start with the basic PRAM models of parallel computation and present guidelines on how to adapt these models for use on GPUs. Special attention is given to the difficulties encountered in the GPU processing of highly irregular data access patterns. These difficulties include coalescing global memory accesses, dealing with concurrent write memory accesses, SIMT thread execution (instruction path divergence) and thread synchronization.

In opposition to the Harish approach, Luo, et al. [55] suggest *hierarchical queue management* and *hierarchical kernel arrangement* approaches, which may provide improved performance. The suggested hierarchical queue management approach facilitates having many threads adding to the BFS queue by breaking the queue up into independent segments. Hierarchical kernel management avoids some synchronization at the top kernel level by using the GPU barriers and fine-grained synchronization operators at certain levels. This work deals mainly with optimized locking and synchronization issues and does not address the GPU workload imbalances, covered by Hong, et al. [42].

Hong, et al. cover accelerating the performance of BFS given the specific constraints of the GPU environment. This work notes the poor performance of the earlier PRAM-like GPU implementations was largely due to GPU thread load imbalances. Load imbalance occurs when the work to be done is allocated to the threads in a warp or work-group[8] improperly, and some threads complete their work and remain idle while other threads continue to work in the same warp or work-group. The solution proposed by Hong involves dividing the GPU code in SISD-like and SIMD-like portions and assigning *virtual warps* to the SISD portions. Virtual warps are artificial groupings of threads that allow the software to better manage the warps for certain processing steps. Hong, et al. make it clear that improved GPU performance can be obtained by introducing non-standard programming technique, like virtual warps, to

[8] *warp* is a CUDA term, *work-group* is a OpenCL term, they both refer to the group of threads being executed together under a single instruction counter, SIMD/SIMT style.

the GPU programming environment.

Dinneen, et al. [27] provide a run-time performance comparison between OpenCL and CUDA, and talk about using OpenCL. In addition, they compare the performance of synchronization at the kernel level with fine-grained, atomic synchronization at the level of individual data items. Their results show that OpenCL and CUDA, over their tests, have similar run-time performance, to within 2% of each other. The results of their synchronization comparisons show that different input graphs, depending on size and density, benefit from each approach. No general preference for OpenCL or CUDA is identified.

Merrill, et al. demonstrate a breadth-first search parallelization, which uses *prefix sum* for buffer management, and achieves an asymptotically optimal $O(|N| + |E|)$ work complexity [59]. None of these previously described BFS works achieved this optimum level of work complexity. The Merrill approach performs synchronous, level-by-level traversal of the graph from the starting vertex. This BFS method uses the expansion of the vertex frontier by traversing the associated edges and then pruning the frontier to contain only unmarked vertices. As the marked vertices are pruned, potential duplicate vertices are also removed. Merrill refers to these two phases as "neighbor expansion" and "status-lookup and filtering", respectively. Prefix sum is used to manage shared, updated buffers in a way that avoids the use of serialization and locking; Merrill refers to this technique as *cooperative allocation*. Here, we quote, "Given a list of allocation requirements for each thread, prefix sum computes the offsets where each thread should start writing its output elements."[59] Of particular interest to our work is the use of prefix sum in the management of shared buffers. These buffers are shared between GPU threads and between different GPUs. In this approach, the *virtual pointers*, denoting the edges in the graph, are passed between GPUs as the graph is traversed. In the next section, we take a deeper look at buffer allocation using prefix sum.

5.2.2.1 Buffer Allocation Using Prefix Sum

The parallelized allocation of shared buffer space using prefix sum provides a buffer management technique that does not require the use of serialization and atomic locking. Figure 5.7 provides an example of allocating buffer space based on prefix sum. In this example, threads *t1, t2, t3*, and *t4* need to put values into the *output results* buffer. The threads need to allocate 3, 2, 0, and 1 items, respectively. The prefix sum computes the sum of the preceding items for each thread, which is the offset to the beginning of each thread's area. The color coding matches the requirement for each thread with its associated area in the buffer. For example, thread $t2$, marked in green, requires space for two in the buffer, and the two items are allocated at offsets 3 and 4. This approach gets around the bottleneck that often occurs on GPUs with allocating space in a shared buffer or queue – the bottleneck is avoided since no serialization or atomic operations are needed. In our work, this approach will be used to create an OpenCL version of BFS, which supports multiple GPU devices. In particular, it will be used to manage the edge virtual pointers that need to be passed between GPUs as the graph is traversed.

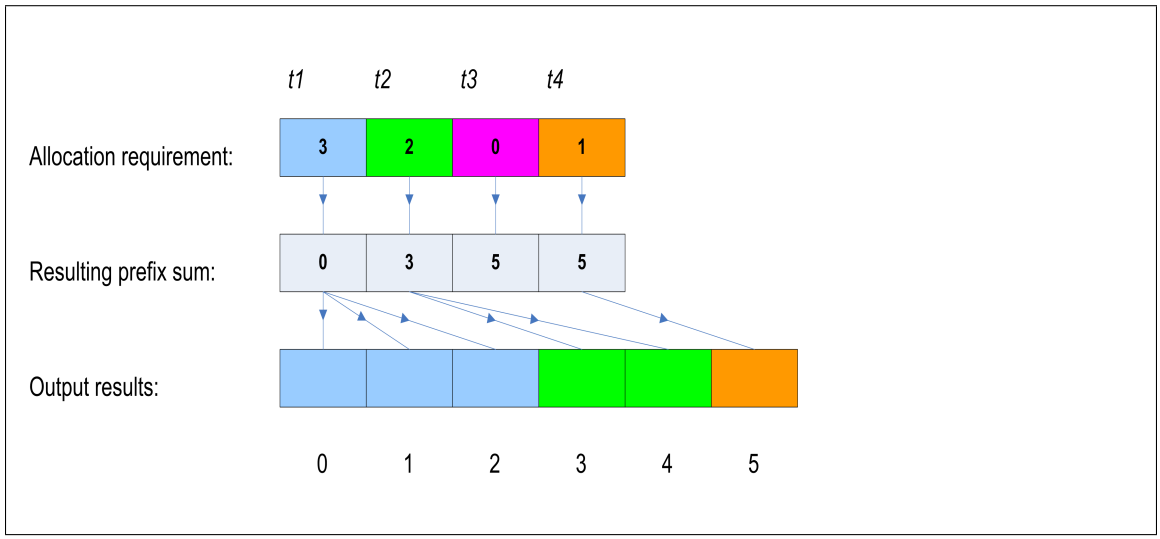

Figure 5.7: Prefix Sum based Buffer Allocation

5.2.2.2 Identification of Graph Repositories

The testing of graph algorithms with large very irregular graphs requires test data. One source of such test data is well known repositories of real-world data. These repositories' graphs are based on actual data from areas such as social networking, communications networks, and geographical mapping; these graphs are not artificially generated test data. Such repositories include:

1. *Stanford Network Analysis Package* (SNAP) [88]

   This package of applications supplies graph datasets from 14 areas including social networks, communications networks, and citation networks. Hong, et al. made use of the repository's *soc-LiveJournal1*[9] dataset in their 2011 article on BFS processing with the Cray MTA-2 [42]. It has 4,847,571 nodes and 68,993,773 edges. Other SNAP datasets/graphs are even larger. The *memetracker9* has 96 million nodes and 418 million links(edges). The social network, directed graph *soc-LiveJournal1* is used in the BFS experiments described below.

2. *Center for Discrete Mathematics and Theoretical Computer Science* (DIMACS) [20] The repository provided by DIMACS, as part of the *9th DIMACS Implementation Challenge - Shortest Paths*, contains 12 USA road networks. Each network (a network being a graph with additional values attached to nodes or edges) is available in two forms: one form gives the distance on each edge and the other gives the transit time. The networks vary in size from the New York City network with 246,346 nodes and 733,846 edges to the Full USA network with 23,947,347 nodes and 58,333,344 edges. Harish and Narayanan [38] use graphs from this repository in their work on All Points Shortest Path.

[9] We denote graph names, OpenCL kernel names, source code variable names, and similar objects in italics.

3. *University of Florida Sparse Matrix Collection* [25, 24]

   This collection is maintained at the University of Florida and contains sparse matrices that arise in real applications. The collection is used by the numerical linear algebra community for the evaluation of sparse matrix algorithms.

5.2.3 Application Software Design

The DEFG-based breadth-first search application from Chapter III, based on the breadth-first search benchmark from OpenDwarfs [31], was designed to work with a single GPU device. We refer to this application as BFS. In order to utilize more than one GPU device, we modified this existing application's kernels, introduced additional kernels, and enhanced the application's DEFG code. This new application has the moniker BFSDP2GPU. Our basic design approach was to continue using the original BFS application's Harish techniques and enhance them to be utilized over two GPUs. This two-GPU execution required that shared buffers, containing the edges on BFS frontier, be moved between the GPUs. These buffers were shared between GPU threads, as well as, between GPUs. We used the prefix scan approach, as previously described, to manage these shared buffers.

5.2.3.1 The BFS Single-GPU Application

We first describe the processing done in the basic BFS application. The BFS application's DEFG code and kernel code are listed in the Appendix, Section B.4. Here, we will show only significant code snippets. Figure 5.8 shows lines 25 through 41 of the BFS application's main processing loop. Line 26 executed *kernel1* and line 35 executed *kernel2*. The *STOP* variable was used to control the loop; the loop was exited when the frontier emptied, wherein the second kernel did not update the *STOP* value to 1. This loop utilized the two kernels to move through the tree, which was held in the *graph_nodes* and *graph_edges* buffers, and to update the breadth-first search

```
25.   loop
26.    execute part1 kernel1 ( graph_nodes(in)
27.                            graph_edges(in)
28.                            graph_mask(in)
29.                            updating_graph_mask(inout)
30.                            graph_visited(in)
31.                            cost(inout)
32.                            NODE_CNT(in)
33.                            )
34.    set STOP (0)
35.    execute part2 kernel2 ( graph_mask(inout)
36.                            updating_graph_mask(inout)
37.                            graph_visited(inout)
38.                            STOP(inout)
39.                            NODE_CNT(in)
40.                            )
41.   while STOP eq 1
```

Figure 5.8: BFS Application's DEFG Loop

frontier, which was held in the *graph_mask* buffer.

Figures 5.9 and 5.10 list the kernels used. They are taken, with only cosmetic changes, from the OpenDwarfs Benchmark.[10] The first kernel was given a node to process and if the node had a *g_graph_mask* non-zero value, meaning the node was on the current frontier, the node's edges were processed. If the node pointed to by an edge had a zero *g_graph_visited* value, the node was put on the new frontier and the updated cost was carried forward. The second kernel served two purposes: it copied the new frontier, stored in *g_updating_graph_mask*, to the frontier, stored in *g_graph_mask*, and it set the *g_over* variable, which maps to the *STOP* variable in the DEFG code, to 1. Setting this variable to 1 caused the loop to be repeated. These two kernels, along with the associated DEFG code, performed the breadth-first search starting with a node preset in the frontier.

5.2.3.2 The BFSDP2GPU Two-GPU Application

Enhancing this approach to use two GPUs entailed several new operations. The graph was shared between the GPUs, and the GPUs had to actively exchange the lists of the nodes on the frontier.

[10]OpenDwarfs Benchmark: Copyright July 29, 2011 by Virginia Polytechnic Institute and State University All rights reserved.

```
01. __kernel void kernel1(__global const Node* g_graph_nodes,
02.                       __global int* g_graph_edges,
03.                       __global int* g_graph_mask,
04.                       __global int* g_updating_graph_mask,
05.                       __global int* g_graph_visited,
06.                       __global int* g_cost,
07.                       int no_of_nodes)
08. {
09.   unsigned int tid = get_global_id(0);
10.   if(tid < no_of_nodes && g_graph_mask[tid] != 0)
11.   {
12.     g_graph_mask[tid] = 0;
13.     int max = (g_graph_nodes[tid].no_of_edges + g_graph_nodes[tid].starting);
14.     for(int i = g_graph_nodes[tid].starting; i < max; i++)
15.     {
16.       int id = g_graph_edges[i];
17.       if(!g_graph_visited[id])
18.       {
19.         g_cost[id] = g_cost[tid] + 1;
20.         g_updating_graph_mask[id] = 1;
21.       }
22.     }
23.   }
24.}
```

Figure 5.9: BFS Application's *kernel1*

```
01. __kernel void kernel2(__global int* g_graph_mask,
02.                               __global int* g_updating_graph_mask,
03.                        __global int* g_graph_visited,
04.                        __global int* g_over,
05.                        int no_of_nodes)
06. {
07.          unsigned int tid = get_global_id(0);
08.          if(tid < no_of_nodes && g_updating_graph_mask[tid] == 1)
09.   {
10.     g_graph_mask[tid] = 1;
11.     g_graph_visited[tid] = 1;
12.     *g_over = 1;
13.     g_updating_graph_mask[tid] = 0;
14.   }
15. }
```

Figure 5.10: BFS Application's *kernel2*

In order to support 2-GPU graphics operations, we provided callable CPU C++ functions to partition the graph into two data structures, with one structure for each GPU, and other functions to reassemble the final cost buffer from the partitions. The graph was first loaded by the *init_input()* function and then the *ArrayPartition2GPU2()* function was called to partition the graph. *ArrayPartition2GPU2()* reformatted the *graph_nodes* buffer and its related buffers; this step involved renumbering the nodes and storing buffer location information in the DEFG_GLOB C++

structure. The DEFG_GLOB structure was used internally by DEFG to manage the presentation of each buffer to the GPUs. For the sake of brevity, we will limit ourselves to showing only highly significant code segments and omit segments of lesser importance.

We note that DEFG did not perform dynamic workload balancing; the allocation of work given to the GPUs, and GPU threads, was determined entirely by the DEFG application implementation. Our experience has been that server nodes with multiple GPU cards were often configured with matching pairs of GPU cards. For this reason, we considered it sufficient to have the application split the workload approximately into equal parts. Our graph application gave half of the nodes to each GPU. When this graph partitioning was performed by the *ArrayPartition2GPU2()* function, no attempt was made to group the nodes in some manner to minimize the cross-GPU communications. The additional work to optimize the graph traversal could have meant unacceptable resource usage within this preprocessing step. This graph partitioning approach was similar to that used in the Merrill work [59].

The DEFG use of more than a single GPU is engaged by replacing the execute statements with the multi_exec statements and changing the declare gpu statement to select multiple GPUs. Once these changes were in place, DEFG would automatically assign the work to the selected GPUs. Of course, the DEFG application and kernels must have been designed to function with multiple GPUs. In the case of the previously-described image filtering applications, the use of more than a single GPU required very little code enhancement. However, for this breadth-first search application, the DEFG code needed to be significantly enhanced to dynamically share the BFS frontier between GPUs. We note that the added complexity was not in the management of the additional GPUs within OpenCL or in the marshaling of buffers to the correct GPU; instead, the additional complexity was rooted in the allocation and population of the shared buffers. We used the DEFG Prefix-Allocation design

pattern to manage this buffer sharing.

Our original BFS application required two kernels: *kernel1* and *kernel2*. The new BFSDP2GPU application required six kernels. The BFS *kernel1* was replaced by five kernels: *bermanPrefixSumP1*, *bermanPrefixSumP2b*, *getCellValue*, *kernel1a2*, and *kernel1b*. The *bermanPrefixSumP1*, *bermanPrefixSumP2b*, and *getCellValue* kernels were provided by the DEFG Prefix-Allocation design pattern. The *kernel2* kernel was retained from the original BFS application. The remaining two kernels, *kernel1a2*, and *kernel1b*, were new and unique to this application.

So as to perform the prefix sum processing, the *bermanPrefixSumP1* kernel was executed once and then the *bermanPrefixSumP2b* was executed $log2(buffer\_size)$ times. These kernels were based on the work by Berman and Paul [14] and have time complexity of $O(n \log n)$. The *getCellValue* kernel was used to quickly obtain the GPU-specific, run-time length of the shared buffers.[11] In an effort to achieve improved performance, we considered using other prefix sum algorithms, besides Berman and Paul. The prefix sum algorithm described in Harris, et al. [40] achieved time complexity of $O(n)$. However the Harris, and other reviewed algorithms, had a power-of-2 buffer length requirement, which was not acceptable here as the sizes of the buffers to be managed vary at run time. We considered extending the buffers, at run time, to a power-of-2 size and using a Harris-like approach. Ultimately, we decided against this buffer-expansion approach, due to the performance implications; the size of the managed buffers can be quite large. The size exceeded 9 million items in our test graphs.

With the allocation of the shared buffers provided by the kernels described above, the *kernel1a2* and *kernel1b* kernels then provided the functionality of the original *kernel1*. Population of the buffers to be moved and shared was done by *kernel1a2*. After *kernel1a2* completed and the cross-GPU data movements occurred, *kernel1b*

[11] This total buffer size information was generated as a side effect of the prefix sum operations.

did the basic breadth-first search processing of the original *kernel1* kernel. After these steps were completed, the original *kernel2* was executed.

The *kernel1a2* kernel source code is shown in Figure 5.11. As stated above, its primary purpose was to traverse the frontier for each GPU and place the active edges and costs into their assigned location in the shared *g_frontier* and *g_payload* buffers. The *g_payload* buffer contained the accumulated BFS tree traversal costs. We note in passing that Harish-based approach is not a particularly good high-performance GPU software design, as this kernel may induce GPU hardware thread divergence when the number of edges per thread is highly varied; each graph node is processed by one, and only one, GPU thread. A node with a large number of edges can be processed much more slowly than a node with a small number of edges. Nonetheless, we felt that the Harish approach was sufficient to test our prefix sum-based DEFG Prefix-Allocation design pattern.

The original *kernel1* was given a node to process and if the node had a *g_graph_mask* non-zero value, then the node's edges were processed. In the 2-GPU version, this work was done by *kernel1b*, which is shown in Figure 5.12. Note that some lines have been omitted from the figure; these omitted lines are very similar to lines 13 to 26, except that the shared buffers from the other GPU was processed. This kernel was given two sets of buffers, one from each GPU, and it scanned these buffers, processing only the edges for the currently-running GPU.

We will now discuss the BFSDP2GPU application's DEFG code. Due to the code length, we will describe it as DEFG pseudo code; we have included only a few DEFG statements in full, those being the statements with a specialized syntax or purpose. Most of the DEFG statements are presented in a less verbose, minimized form. The BFSDP2GPU pseudo code is shown in Figure 5.13. An ellipsis was used to indicate omitted statements and parameters. The references to minor scalar variables have been omitted. However, due to their key functions, we showed the important

```
01. __kernel void kernel1a2(__global const Node* g_graph_nodes,
02.                         __global int* g_graph_edges,
03.                                 __global int* g_graph_mask,
04.                                         __global int* g_graph_offset,
05.                                 __global int* g_cost,
06.                                         __global int* g_frontier,
07.                                         __global int* g_payload,
08.                                 int no_of_nodes)
09. {
10.   unsigned int tid = get_global_id(0);
11.          if(tid < no_of_nodes && g_graph_mask[tid] != 0)         // in range with edges
12.          {
13.            g_graph_mask[tid] = 0;
14.            if (g_graph_nodes[tid].no_of_edges > 0)
15.            {
16.       int cost = g_cost[tid];
17.              int max = (g_graph_nodes[tid].no_of_edges + g_graph_nodes[tid].starting);
18.              int index = g_graph_offset[tid];
19.              for (int i = g_graph_nodes[tid].starting; i < max; i++)
20.              {
21.                int id = g_graph_edges[i];
22.                g_frontier[index] = id;
23.                g_payload[index] = cost;
24.                index++;
25.              }
26.            }
27.   }
28. }
```

Figure 5.11: BFSDP2GPU Application's *kernel1a2*

```
01. __kernel void kernel1b(
02.                 __global int* g_frontier0,
03.                 __global int* g_payload0,
04.                         int list_size0,
05.                         __global int* g_frontier1,
06.                         __global int* g_payload1,
07.                         int list_size1,
08.                 __global int* g_updating_graph_mask,
09.                         __global int* g_graph_visited,
10.                 __global int* g_cost,
11.                         int gpu_id)
12. {
13.   int index = get_global_id(0);
14.   if (index < list_size0)
15.   {
15.            int id = g_frontier0[index];
16.            if (MAP_DEVICE(id) == gpu_id)
17.     {
19.              int nid = MAP_NODE(id);
20.              if(!g_graph_visited[nid])
21.              {
22.                g_cost[nid] = g_payload0[index] + 1;
23.                g_updating_graph_mask[nid] = 1;
24.              }
25.            }
26.   }
--    similar if stmt code for list_size1, g_frontier1, etc. ommited
38. }
```

Figure 5.12: BFSDP2GPU Application's *kernel1b*

references to the *KCNT* and *STOP* variables.

This code shows the C++ functions and OpenCL kernels being utilized. Lines 11 and 12 contain the calls that obtained the input graph and performed its partitioning. Likewise, calls used to merge the cost array and output the final node-by-node costs are in lines 27 and 28. Lines 14 and 26 mark the boundaries of the "outer" loop. Line 16 shows the "inner" loop execution of the *bermanPrefixSumP2b* kernel having been done KCNT times. The *bermanPrefixSumP1* and *bermanPrefixSumP2b* kernels were used at the start of each "outer" loop iteration to establish the shared buffer offsets for each node. At lines 18 through 20, the *getCellValue* kernel was used to obtain the size of the shared buffer for each GPU. Note that the amount of shared buffer space utilized by each GPU was likely different.

The "@" <GPU-ID> notation, shown on lines 19 and 20, is used to indicate which buffer is presented to a given GPU. The first GPU selected is assigned an ID of 0, and the ID is incremented for each additional GPU in use. This specialized technique is used when the default DEFG approach to multiple-CPU buffer management is not applicable. The default DEFG approach of simply splitting the buffer is not applicable because the size of the shared areas used on each GPU is variable at run time and each shared buffer list must be named uniquely. Although this "@" <GPU-ID> notation is not elegant, we feel it is within the design goals of DEFG, as it is a declaration of what to do and not coding for how to do it.

The broadcast statements on line 22 caused the listed buffers to be made available to the other GPU. Due to OpenCL limits, these DEFG broadcast statements merely cause the named buffers to be copied to the CPU's memory and made available, from there, to the other GPU(s). As will be covered in our run-time performance discussion, this approach to broadcasting data to the other GPU(s), unfortunately, brings about somewhat unimpressive performance.

Lines 21 and 23 show the execution of the two kernels, which was derived from the

```
01. declare application bfsdp2gpu
02.  declare integer NODE_CNT (0) ...
03.  declare gpu gpugrp ( all )
04.  declare kernel bermanPrefixSumP1  bfsdp_kernelv3 ( [[ 1D,NODE_CNTt2 ]] )
05.          kernel bermanPrefixSumP2b bfsdp_kernelv3 ( [[ 1D,NODE_CNTt2 ]] )
06.          kernel getCellValue       bfsdp_kernelv3 ( [[ 1D,1 ]] )
07.          kernel kernel1a2          bfsdp_kernelv3 ( [[ 1D,NODE_CNT ]] )
08.          kernel kernel1b           bfsdp_kernelv3( [[ 1D,EDGE_CNT ]] )
09.          kernel kernel2            bfsdp_kernelv3 ( [[ 1D,NODE_CNT ]] )
10.  declare integer buffer graph_edges (EDGE_CNT )  nonpartable ...
11.  call init_input (graph_nodes(out) ... )
12.  call ArrayPartition2GPU2( graph_nodes (inout) ... )
13.  // set misc. control variable calculations, eg. KCNT
14.  loop
15.    multi_exec run2 bermanPrefixSumP1( offset (out) ... )
16.    sequence KCNT times
17.      multi_exec run2 bermanPrefixSumP2b (offset2 (inout) ... )
18.    multi_exec run3 getCellValue( offset2 (in)
19.                                  NODE_CNT0 @0 (in)  NODE_CNT1 @1 (in)
20.                                  listused0 @0 (out) listused1 @1 (out) )
21.    multi_exec s2 kernel1a2( graph_nodes(in) ... )
22.    broadcast (frontier0 @0) ... broadcast (payload1 @1)
23.    multi_exec s3 kernel1b(frontier0(in) ... )
24.    set STOP (0)
25.    multi_exec s4 kernel2 ( graph_nodes(in) ... STOP ... )
26.  while STOP eq 1
27.  call MergeCost2GPU2(cost(inout) ... )
28.  call disp_output (cost(in) ...)
29. end
```

Figure 5.13: BFSDP2GPU DEFG Pseudo Code

original BFS *kernel1*. Lines 14 and 26 utilized the same loop management approach as used in the original BFS DEFG code. The kernel executed on line 25 was the same *kernel2* as used in BFS.

5.2.4 Approach to Research

The development of our DEFG-based BFSDP2GPU application was done with two main research aims in mind: to further show the viability of DEFG and to demonstrate a two-GPU, breadth-first search application that utilizes prefix sum for shared buffer management. In order to test our new BFS application within these aims, we compared the results and run-time performance of our original BFS application against our new two-GPU BFS application. We used selected graphs from the SNAP repository, and Rodinia/OpenDwarfs Benchmark, in these comparisons [85, 88]. Our new BFSDP2GPU application produced correct results. We note that we compared

Table 5.6: Rodinia Graph Characteristics

Graph	Nodes	Edges
g65536	65,536	393,216
g1M	1,000,000	6,001,836
g2M	2,000,000	11,999,346
g3M	3,000,000	18,003,170
g4M	4,000,000	23,999,338
g5M	5,000,000	29,995,814

breadth-first search results from the existing BFS application, over a subset of the Rodinia/OpenDwarfs graphs, with our new application. The results matched exactly.[12] Unfortunately, the performance of our new BFSDP2GPU application was not impressive relative to that of the BFS application.

5.2.5 Experimental Results

We first explore the BFSDP2GPU run-time performance using graphs from the Rodinia Benchmark. We then run this application with two graphs from the SNAP Package. Seeing similar run-time results from both sets of graphs, we then perform an analysis to find the explanations for the performance we observed.

5.2.5.1 Run-Time Performance Results from Rodinia Graph Data

These tests were run comparing the BFS and BFSDP2GPU run-time performance, using a set of the Rodinia BFS Benchmark graphs. The benchmark is provided with a number of generated test graphs and we used the included tool, graphgenr.cpp, to generate additional large graphs. The 65,536 node graph, *g65536* was included with the benchmark and the larger graphs were generated with the Rodinia tool. The graphs are listed Table 5.6; it contains the name we gave each graph, and the number of nodes and edges. All of these are *directed* graphs.

Table 5.7 shows the average run times of three BFSDP2GPU runs for each of the

[12]The BFS application results had been previously compared with the original BFS results from the OpenDwarfs OpenCL software.

Table 5.7: Run Times of BFS Versus BFSDP2GPU, in Seconds

Graph	BFS Application	BFSDP2GPU Application	Slowdown Factor
g65536	0.008	0.140	17.5
g1M	0.074	0.762	10.3
g2M	0.158	2.345	14.8
g3M	0.276	1.650	6.0
g4M	0.377	2.490	6.6
g5M	0.505	4.089	8.1

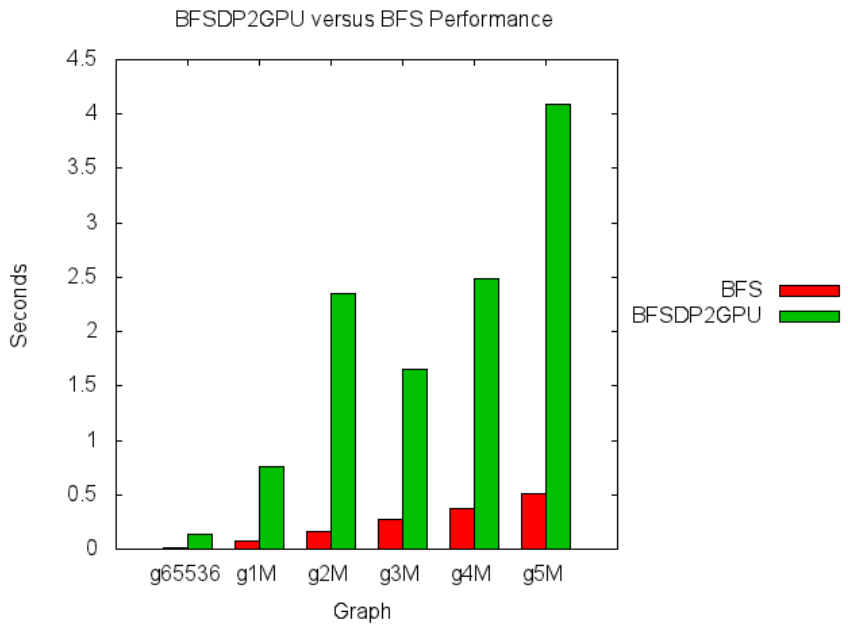

Figure 5.14: BFS Versus BFSDP2GPU Run Times with Rodinia Graphs

graphs. As an example, the 0.008 in the first data row and second column means that the BFS application processed the *g65536* graph in an average of 0.008 seconds. The BFS processing was started with the graph root set to first graph node. These same run-time results are presented in bar-chart form in Figure 5.14. Given that our intention was for BFSDP2GPU to be a high-performance application, these results were disappointing. Our new application's run times ranged between 6 and 17.5 times those of the existing DEFG BFS application.

Table 5.8: SNAP graph characteristics

Graph Details	Nodes	Edges	Description
soc-Slashdot0811	77,360	905,468	Slashdot social network from November 2008
soc-LiveJoural1	4,847,571	68,993,773	LiveJournal online social network

Table 5.9: Run Times from SNAP Graphs, BFS Versus BFSDP2GPU, in Seconds.

Graph	BFS Application	BFSDP2GPU Application	Slowdown Factor
soc-Slashdot0811	0.011	0.075	7.1
soc-LiveJoural1	0.292	3.571	12.2

5.2.5.2 Run-Time Results from Two SNAP Graphs

We performed a second set of performance tests, using the SNAP *soc-LiveJournal1* and *soc-Slashdot0811* graphs, to help determine if the application's slower then expected performance with the Rodinia data was perhaps due to some trait of the data. Two SNAP graphs were chosen because of their vastly different sizes and sources. Table 5.8 provides a detailed description of these directed graphs. We considered the second graph, *soc-LiveJoural*, to be a good example of a VLI graph.

In Table 5.9, we observe similar poor performance as was observed with the Rodinia graphs. Rather than further note the performance of our new application with additional run-time testing results, we turn our attention to the reasons for the observed performance outcomes.

5.2.5.3 Run-Time Performance Analysis

The aforementioned Merrill article demonstrated good breadth-first search performance with multiple GPUs, using NVIDIA's CUDA [59]. Our decision to include the prefix sum-based buffer management in DEFG, as a design pattern, was largely based on the good performance shown by Merrill. Given the performance we saw with BFSDP2GPU, our task became finding explanations for the large performance gap between Merrill's results and ours.

After executing numerous additional performance tests and reviewing the designs of DEFG and the BFSDP2GPU application, we outline four potential causes for the unimpressive run-time results of our application: (1) OpenCL's lack of direct GPU-to-GPU communications; (2) DEFG's lack of support for variable-length buffers; (3) the mixture of sparse-array and list-based data structures in the application; and, (4) the application's GPU work allocation method. We now explore each of these.

5.2.5.4 OpenCL-Provided GPU-to-GPU Communications Limitations

The DEFG broadcast statement is used to provide the GPU-to-GPU communications capability. Since OpenCL does not provide for actual GPU-to-GPU communications, the DEFG broadcast causes two major OpenCL API calls to be generated for each transfer. The first API call, *clEnqueueWriteBuffer()*, moves the buffer to the CPU and the second, *clEnqueueReadBuffer()*, brings the buffer to the requesting GPU.

The GPU-to-GPU communications capability provided by NVIDIA's CUDA uses the PCIe hardware bus for direct GPU-to-GPU communications and is presented as having attractive transfer rates [68]. CUDA tests run on the Hydra server showed CUDA inter-card transfer rates of 6.05GB/second. With 6GB as an actual baseline figure, we produced a DEFG-based OpenCL application, called DiagBR, to time the movement of buffers between the two Hydra GPUs, without any kernel processing involved. Our aim was to compare the CUDA and DEFG/OpenCL inter-card data transfer speeds on Hydra. DiagBR produced a transfer rate of around 286MB/second. The CUDA rate was approximately 21 times the OpenCL rate.

With this fact established, we turned on the BFSDP2GPU application's DEFG logging so as to profile the run time consumed by the two broadcast statement's OpenCL API calls. When the application was rerun with the *g1M* SNAP graph, we found that a total of 0.590 seconds was used by the broadcasting. This is 76.6% of the total run time of 0.771 seconds. We note that this total run-time value, obtained

with logging enabled, is slightly different than the 0.762 value shown in Table 5.7; we attribute this difference to the run-time overhead added by the DEFG logging. It is clear that the lack of an OpenCL-provided GPU-to-GPU direct communications facility drastically impacts the application's run-time performance. The lack of this capability in OpenCL has a substantial performance impact on DEFG. This loss of performance is not due to an error in DEFG; instead, it is due to the OpenCL lack of direct GPU-to-GPU communications support.

5.2.5.5 DEFG Variable-Length Buffers

The performance numbers we have shown so far for BFSDP2GPU included a manual step of tuning the size of the application's broadcast-list buffers for each differently sized input graph. With the *g1M* graph, this broadcast-list buffer size was manually set to 2M elements; with the *g6M* graph it was set to 9M elements.

When a trial test run was done with a 9M size and the *g1M* graph, the average run-time increased from 0.762 seconds to 2.634 seconds. This is a very noticeable change and it indicates how sensitive this application is to the broadcast-list buffer size. This DEFG performance issue can be handled, in the future, by enhancing DEFG to be able to declare that a given scalar variable holds the current buffer transfer size and using this variable's value in the call to the OpenCL buffer movement functions. We note that in our run-time tests, even these manually tuned transfer buffer sizes overstate the buffer size, as they are sized for the largest lists used. Enhancing DEFG to utilize the transfer size of the actual list would prevent the slowdowns associated with this undesirable overstating of the volume of data to be transferred.

5.2.5.6 Data Structures and Threading Model

The data structure design and threading model used in this application have an impact on its run-time performance. If we assume that the DEFG broadcast processing

took no time at all, then an optimistic estimate of the BFSDP2GPU processing time for the g1M graph could be 24% of 0.762, or 0.183 seconds. The time for the BFS application processing the same graph was 0.074 seconds. Clearly, the slow broadcast performance does not, by itself, explain the entire performance issue.

In additional profiled executions, we measured that about 5% of the BFSDP2GPU run time was spent doing the *ArrayPartition2GPU2*() and *MergeCost2GPU2*() pre-processing and post-processing operations and another 3.5% was spent in the prefix sum operation. If we subtract the pre-processing and post-processing 5%, and the prefix sum processing of 3.5% from the 0.183 seconds $(0.183 - (0.085 * 0.762))$, we are then left with an execution time of about 0.118 seconds. This 0.118 second figure is a crude estimate of the time needed to move the graph buffers to the GPUs, to populate the frontier data and to perform the actual BFS processing. This crude estimate exceeds the equivalent BFS run time of 0.074, by 0.044 seconds, nearly 60%. Some aspect of the performance difference is not yet exposed.

Unfortunately, the Linux timers we could use on Hydra are accurate only to milliseconds and many of the DEFG-logged times with this *g1M* graph dropped to the 1ms level, and below. This limitation severely affected our ability to further analyze the performance differences. Nevertheless, in our opinion, the BFSDP2GPU use of two different data structure types and the one-node-per-thread work management model were not providing sufficiently good performance. The performance lost to the lack of OpenCL GPU-to-GPU direct communications significantly overshadowed other performance issues. However, the mixing of the dense-style array structure, for the BFS frontier and cost array, with the list-style structure, for the shared lists, also caused extra work to be done and, we suspect, a loss of performance. The use of the classic Harish GPU threading model may also be suspect, in terms of performance, because of the potential thread divergence issues, previously mentioned, and the unfortunate characteristic that threads assigned to nodes not on the frontier have

very little valid work to perform.

5.2.6 BFSDP2GPU Goals and Run-Time Performance

These explanations, and views, bring us back to the Merrill article and our previously established aims for this application and DEFG. The two main goals for BFSDP2GPU were: (1) implement this reasonably complex application in DEFG, and (2) utilize the DEFG Prefix-Allocation design pattern. We achieved both of these. In order to have Prefix-Allocation be a general design pattern in DEFG, it must be able to be used in an independent manner and not be highly integrated with a specific application or a certain GPU threading model. The Merrill work used a very sophisticated GPU-mapping approach, not the Harish GPU threading model. In hindsight, it appears the high level of performance observed with the Merrill work, compared to the Harish BFS approach, can only occur if the added prefix sum and marshaling overhead can be offset by a decrease in the run time needed to do the actual BFS processing. This was not the case with BFSDP2GPU. The performance of our BFSDP2GPU application did not meet expectations; however, it helped show the range of applications DEFG can handle and it helped us further understand the limits of OpenCL-based processing with multiple GPUs.

5.3 Application: Sorting Roughly Sorted Data

5.3.1 Problem Definition and Significance

General comparison sorting has an $O(n \log n)$ optimal performance bound. This performance level can be improved upon in specific cases; for example, when the sequence to be sorted is already partially sorted. We refer to this sorting of partially sorted data as *roughly sorting*. Formally, roughly sorting pertains to sorting nearly sorted sequences. In this context, a roughly sorted sequence is k-sorted if no sequence element is more than k steps out of sequence. With the sequence k-sorted, Altman,

et al. show that this sequence can be sorted with an $O(n \log k)$ performance bound [9, 10].

The sorting of sequences is a primitive operation in many algorithms and data manipulation operations including binary searching, closest pair determination, uniqueness determination, identifying outliers, database acceleration, and data mining [22, 51, 41, 35]. Roughly sorted sequences can occur when a previously sorted sequence is perturbed with relatively minor changes [10]. An example of a roughly sorted sequence would be a list of all the cities in a region, sorted constantly by city population, where population updates are performed very frequently. The population of each city may change constantly, but it is unlikely that a city's position in the sorted list would significantly change in a very short interval of time.

We have developed GPU-based parallel roughly sorting in DEFG. The implementation uses the high-level algorithm outlined by Altman; it involves three distinct phases. The first phase is the determination of the degree of disorder in the sequence, that is determining the k value. The second phase is the partitioning of the input sequence into fixed-length blocks, using the k value to size the blocks. In the third phase, these blocks are individually sorted, in parallel. When all of these blocks are sorted, the entire k-sorted sequence is sorted. The emphasis of this work is on phases one and two. We use the *comb sort* [76] to satisfy phase three. With k values less than 200[13], our GPU-based rough sort showed impressive run-time results compared to those of the standard CPU Quick Sort. Our rough sort shows increased performance, when k is small relative to n because $O(n \log k)$ is substantially less than $O(n \log n)$.

[13]The 200 value is an estimate. With some of the datasets we used this value was as high as 2000.

5.3.2 Related Work and GPU-based Sorts

Roughly sorting is an instance of an *adaptive* sorting algorithm. Estivill-Castro and Wood provide a dated survey of Adaptive Sorting Algorithms [29]. Their work identifies a number of different measures of disorder. In a later work, Estivill-Castro identify *Dis* and *Max* as the two most common measures of disorder used with adaptive sorting [19]. Dis is equivalent to the measure of k used here for roughly sorting.

The third phase of roughly sorting requires that sorting be performed on the individual blocks of roughly sequenced data. This means a GPU-based sort is required. Satish, et al. describe their CUDA-based radix and merge sorts where they claim a 2-4 times speedup over previous GPU-based sorts. They also claim it to be 23% faster than even a highly optimized CPU sorting routine [79]. Harris, et al. [40] describe the use of CUDA GPU-based prefix scan in radix-based parallel sorting and based on the prefix scan work by Blelloch [15]. Merrill and Grimshaw provide an in-depth summary of sorting on GPU architectures in his technical report [60].

As our work is OpenCL-based, and not CUDA-based, we could not directly use the Satish-produced sorts. We decided not to use a radix-based sort due to the limits associated with the format of the sort-key values and other issues [22]. The ubiquitous Quick Sort is not usable here, as OpenCL 1.1 does not support recursive calls in kernel code [22]. Therefore, for our work, we needed a different sort. We searched for a sort that was an in-place sort and was not recursive, and chose the *comb sort* [62, 76, 97].

The comb sort was first designed by Dobosiewicz [18] in 1980 and was later rediscovered by Lacey and Box in 1991 [76]. The Lacey article provides a good overview of the comb sort. The comb sort is a variant of the bubble sort, but with much-improved performance. The comb sort has outer and inner loops like the bubble sort; however, instead of comparing adjacent values, the comparisons are between elements some *gap* apart. The starting gap is set to the number of items to sort and at the start of each sorting iteration *gap* is set to: $gap/shrink$. The *shrink* value is normally set to

1.3. Lacey discusses how the value of 1.3 is obtained [76]. The *gap* value, therefore, drops with each iteration. The iterations stop when the *gap* value is 1 and the just-completed iteration resulted in no comparison swaps. The comb sort is also used in the GPU work by Nagao and Mori [62]. They use the comb sort in their language analysis work for the same reasons we do: this sort has a sort-in-place design, avoids the use of recursion, and provides good performance.

5.3.3 Application Software Design

In this section, we describe our roughly sorting applications. We produced two DEFG implementations of roughly sorting; we call our single-GPU roughly sorting version RSORT and the other version is RSORTM, our multiple-GPU implementation.

The Altman roughly sorting approach specifies the LR, RL, and DM steps [10]. The pseudo code for steps LR, RL, and DM is given in Figure 5.15. After these steps are completed, a parallel sort is performed. We use the comb sort for this parallel sorting step. Our implementation of the RSORT application requires the DEFG-created code for the CPU-side, the GPU-based comb sort kernel and the kernels for the LR, RL, and DM steps. In addition, a simple kernel is required to find the upper limit on the DM-supplied distance measures. We call this additional step UB. In total, five kernels are used by RSORT. The result of the DM step is an array of distances; this array's maximum value is the desired k value for the partially-sorted data.

The kernels for the LR, left-to-right maximum, and RL, right-to-left minimum, steps require parallel implementations. In a parallel-processing environment, these cannot be written as common serial for loops. We have implemented them as OpenCL kernels using parallel prefix scan. These two kernels are based on the Berman [14] prefix scan approach, which was also used in the DEFG Prefix-Allocation design pattern.

While describing this RSORT application, and unlike the previous DEFG application descriptions, we show the complete source code. No code is omitted. This tends to make the code figures larger, but it makes it possible to see how much can be accomplished with less than a hundred lines of DEFG code and the associated kernels. In addition, inclusion of the full source code provides solid examples of DEFG code-insertion morsels.

Figure 5.16 shows the *LRmax* and *RLmin* kernels. The *LRmax* kernel computed the running maximum, moving left to right and the second kernel computed the running minimum, moving right to left. Both kernels functioned in a similar manner and we will summarize only the inner workings of *LRmax*. Lines 29 through 38 of

```
procedure LR(α, B[1...n]);
begin
   B[1] := a_1;
   for i := 2 to n
      if B[i − 1] < a_i then B[i] := a_i
      else B[i] := B[i − 1]
end.

procedure RL(α, C[1...n]);
begin
   C[n] := a_n;
   for i := n − 1 downto 1
      if C[i + 1] < a_i then C[i] := a_i
      else C[i] := C[i − 1]
end.

procedure DM(B[1...n], C[1...n], D[1...n]);
begin
i := n;
for j := n downto 1 do
  while (j ≤ i) and (i > 0) and (C[i] ≤ B[j])
       and ((j = 1) or (C[i] ≤ B[j − 1]) do
     begin
        D[i] := i − j;
        i := i − 1;
     end
end.
```

Figure 5.15: *LR*, *RL*, and *DM* Pseudo Code

```
01. __kernel void LRmax(__global int src[], __global int dst[], uint stride)
02. {
03.    uint block = get_global_id(0);
04.    uint size = get_global_size(0);
05.    uint arnold = size - stride;
06.    if (block >= arnold) return;
07.    uint js = block + stride;
08.    if (js >= size) return;
09.    int src_j_item = src[block];
10.    int src_js_item = src[js];
11.    if (block < stride) { // copy already processed
12.      dst[block] = src_j_item;
13.    }
14.    if (src_js_item < src_j_item) {
15.      dst[js] = src_j_item;
16.    } else {
17.      dst[js] = src_js_item;
18.    }
19. }

01. __kernel void RLmin(__global int src[], __global int dst[], uint stride)
02. {
03.    uint block = get_global_id(0);
04.    uint size = get_global_size(0);
05.    if (block < stride) return;
06.    int js = block - stride;
07.    if (js < 0) return;
08.    if (block >= (size - stride)) {        // copy already processed
09.      dst[block] = src[block];
10.    }
11.    if (src[js] > src[block]) {
12.      dst[js] = src[block];
13.    } else {
14.      dst[js] = src[js];
15.    }
16. }
```

Figure 5.16: *LRmax* and *RLmin* Kernels

the CPU-side DEFG code, in Figure 5.19, utilized this kernel. This DEFG CPU-side code provided the *src*, *dst*, and *stride* parameter values to the kernels; the value of *stride* began with one and was doubled on each successive iteration.

In Figure 5.16's *LRmax* kernel, lines 3 through 10 setup the index values used to access the *src* and *dst* arrays, obtained the values to be processed, and performed boundary checking. The following lines, 11 through 18, copied the already-processed values from the *src* to *dst* array and processed the current values. The *RLmin* kernel worked in a similar manner, putting items in a decreasing sequence while it moved right to left.

The *DM* and *UB* kernels are shown in Figure 5.17. The *DM* kernel used the maximum and minimum arrays, produced by the previous two kernels, to establish

the distance that each value, to be sorted, was out of sequence. The for loop used in the *DM* algorithm, shown in Figure 5.15, was replaced by the GPU execution of this function at line 49 of Figure 5.19. The while loop in the *DM* algorithm was replaced by the kernel's for and if statements on lines 04 and 05.[14] The second kernel in Figure 5.17 was straight-forward; its function was to determine if any value in the *D* array was larger than the scalar value *d*. If any element was larger, the *again* value was set to 1. The value of *d*, the radius in the DEFG code, was started at one and doubled with each additional invocation. The purpose of kernel *UB* was to quickly establish an upper bound on the maximum value in the *D* array.

Next, we discuss the DEFG program that was used to drive these kernels. Altman describes the overall roughly sorting process in three phases and we will describe our DEFG code in terms of these three phases. Our DEFG code is listed in two Figures 5.18 and 5.19. The DEFG declarations used by our implementation are shown in Figure 5.18. We note that five kernels were declared in lines 14 through 18 and that six equal-sized buffers were declared in lines 19 through 24. We also note that

[14] Algorithm *DM* is deceptively complex and converting it to an OpenCL kernel was *interesting* and *challenging*.

```
01. __kernel void DM(__global int B[], __global int C[], __global int D[])
02. {
03.    int i = get_global_id(0);
04.    for (int j = i; j >= 0; j--) {
05.      if ((j <=i) && (i>=0) && C[i]<=B[j] && ((j==0) || (C[i]>=B[j-1]))) {
06.        D[i] =  i-j;
07.        break;
08.      }
09.    }
10. }

01. __kernel void UB( __global int D[], uint size, int d, __global uint *again)
02. {
03.    if (*again == 1) return;
04.         int i = get_global_id(0);
05.         if ((D[i]) <= d) {
06.        // good
07.       } else {
08.                *again = 1;
09.     }
10. }
```

Figure 5.17: *DM* and *UB* Kernels

```
01. declare application  RSort
02.  declare integer stride (1)
03.          integer size   (64)
04.          integer sizeDB (0)
05.          integer genK   (0)
06.          integer bufSize (0)
07.          integer radius (1)
08.          integer groups (0)
09.          integer again  (0)
10.          integer offset (0)
11.          integer offset2 (0)
12.          integer logSize (0)
13.  declare gpu gpuone ( * )
14.  declare kernel LRmax RSort_Kernels  ( [[ 1D,size ]] )
15.          kernel RLmin RSort_Kernels  ( [[ 1D,size ]] )
16.          kernel DM RSort_Kernels  ( [[ 1D,size ]] )
17.          kernel UB RSort_Kernels  ( [[ 1D,size ]] )
18.          kernel comb_sort RSort_Kernels  ( [[ 1D,groups ]] )
19.  declare integer buffer arrayS (bufSize)
20.          integer buffer LR     (bufSize)
21.          integer buffer LRout  (bufSize)
22.          integer buffer RL     (bufSize)
23.          integer buffer RLout  (bufSize)
24.          integer buffer DMbuf  (bufSize)
```

Figure 5.18: RSORT DEFG Declare Statements

the *comb_sort* kernel was declared somewhat differently concerning the *work-group* size. The work-group dimension, or width, is the number following the "1D," within the double brackets. The first four kernels had a work-group width of *size* and the *comb_sort* kernel had a work-group width of *groups*. The *size* variable contained the number of items to be sorted and the *groups* variable contained the number of parallel sorts to be done by the comb sort. Given a fixed number of items to sort, the larger the value of *groups*, compared to *size*, the better this application performs, because a larger number of parallel smaller sort processes complete in less time than a smaller number of larger parallel sort processes. There was one GPU thread allocated to each sort group and, hence, each *comb_sort* instance.

We continue by describing the first-phase processing. The *LRmax* kernel was driven by lines 29 through 38 of the DEFG code, shown in Figure 5.19. The kernel was called once from line 30 to start the prefix-maximum processing and to move the results to the *LR* array. The loop shown in lines 33 through 38 continued iterating the kernel, with the stride being increased on each iteration. Looping was terminated

when the *again* value exceeded $\log 2$ of the items to be sorted.

The purpose of line 36, containing the DEFG interchange statement, requires some explanation. The *LRmax* kernel does not use atomic locking or synchronization. It strictly read data from the *LR* array and wrote to the *LRout* array. The purpose of the interchange statement was to perform a high-performance swap of the two arrays. Lines 39 through 48 functioned similarly to lines 29 through 38, except the *RLmin* kernel was used. The *DM* kernel was executed from line 49. The resulting *DMbuf* array was processed by lines 50 to 54 to obtain an upper-bound radius value.

The second phase consisted of using the new radius value to determine the size of the sorting blocks and to test various special conditions. These operations, implemented with DEFG morsels, are shown in lines 55 through 57. With the *groups* value having been set in line 57, the third processing phase could begin.

Phase three consisted of two calls to the *comb_sort* kernel. This kernel was first called on line 58 and then again on line 61. The code on lines 59 and 60 did offset the array to be sorted by a *radius* value and lowered the *groups* value by one. This action overlapped the previously-sorted data with new sort blocks and was done to re-sort the previously sorted groups. This subtle step made it possible for out-of-sequence items to be moved between the original sort groups.

Line 63 called the *putMergeArray()* function to write the sorted data to disk. The call to the *sync()* function[15] informed the DEFG optimizer to transfer the updated *arrayS* array contents to the CPU and the DEFG code morsel on line 63 performed the call to output the results to the *sorted.txt* file. This function was called from a morsel, and not a DEFG call statement, because DEFG does not directly support string literals. Strings are outside the proper domain of DEFG.

[15]The sync() function is made available in the defg_loader.h header file.

```
25.  code [[ char* arg = "16"; if (argc > 1) {arg = argv[1];} ]]
26.  code [[ if (argc > 2) { size = (int) pow(2.0, (double) atoi(argv[2])); } ]]
27.  code [[ getArray(arg, arrayS, size); bufSize = size; ]]
28.  code [[ logSize =  int(log(double(size))/log(2.0));  ]]
29.  set stride (1) // start LR processing
30.  execute LR1 LRmax (arrayS(in) LR(out) stride(in))
31.  call times2(stride(*))
32.  set again (1)
33.  loop
34.   execute LR2 LRmax (LR(inout) LRout(out) stride(in))
35.   call times2(stride(*))
36.   interchange(LR LRout)
37.   code [[ again++; ]]
38.  while again lt logSize
39.  set stride (1)  // start RL processing
40.  execute RL1 RLmin (arrayS(in) RL(out) stride(in))
41.  call times2(stride(*))
42.  set again (1)
43.  loop
44.    execute RL2 RLmin (RL(in) RLout(out) stride(in))
45.    call times2(stride(*))
46.    interchange(RL RLout)
47.    code [[ again++; ]]
48.  while again lt logSize
49.  execute DM1 DM (LR(in) RL(in) DMbuf(out)) // DM processing
50.  loop // start UB processing
51.   set again (0)
52.   call times2(radius(*))
53.   execute UB1 UB (DMbuf(in) size(in) radius(in) again(inout))
54.  while again ne 0
55.  code [[ radius *= 2; ]] // determine block sizes
56.  code [[ if (radius > size) { radius = size; } ]]
57.  code [[ groups = (int) ceil( ((double) size / (double) radius)); ]]
58.  execute SORT1 comb_sort(arrayS(inout) radius(in) offset(in) groups(in))
59.  code [[ offset2 = radius / 2; ]]
60.  call dec(groups(*))
61.  execute SORT2 comb_sort(arrayS(inout) radius(in) offset2(in) groups(in))
62.  call sync (arrayS(in))
63.  code [[ putMergeArray("sorted.txt", arrayS, size); ]]
64. end
```

Figure 5.19: RSORT DEFG Executable Statements

5.3.4 Experimental Results

5.3.4.1 Approach to Experimentation

In this section, we compare the RSORT and RSORTM run-time performance relative to the performance of the ubiquitous CPU-based Linux Quick Sort. We used Quick Sort as our sort run-time performance baseline. We considered several other options for the baseline sort. The AMD Application SDK [1] includes a GPU sorting example, but this is based on a radix-style sorting algorithm. This sort was rejected, because we did not want to work around limits of radix sorting [22]. As noted in Section

5.3.2 on related work, most available GPU-based sorts are written to use CUDA and not OpenCL. Therefore, we settled on using the ubiquitous Linux CPU-based Quick Sort[16] as our baseline sort. Quick Sort is a commonly used general-purpose sort and we view it is a valid measurement "yardstick."

We found it difficult to locate usable, real-world, roughly sorted data for our performance testing. Our solution to this problem was to write a tool to artificially create the needed sorting test cases. The tool was given the desired radius value and the number of items to be generated; it then generated the requested data. The numbers in the generated data were integers between one and number of desired items; these numbers having been perturbed to achieve the requested radius. The resultant data could either have one perturbation or have each segment of size $radius + 1$ items perturbed, depending on what was requested. A highly perturbed data set of 16 items with a radius of 4 would contain:

```
5 4 3 2 1 10 9 8 7 6 15 14 13 12 11 16
```

We most often used test data that was highly perturbed, and noted when the run-time performance test being discussed was based on minimally perturbed data. The datasets, of test data, were named with the radius, which we called a "k value," and the size. For our tests with large numbers of items, the size was given as a power of 2. We performed a number of different run-time tests. The datasets used had either 2^{23}, 2^{26}, or 2^{27} elements.

5.3.4.2 Single-GPU Performance

Single-GPU Experiment One

Table 5.10 shows the run-time comparison results from executing our RSORT application against Quick Sort (QSORT). These were executed on Hydra and the numbers

[16]By this, we mean using the qsort() function from the Linux C/C++ run-time library.

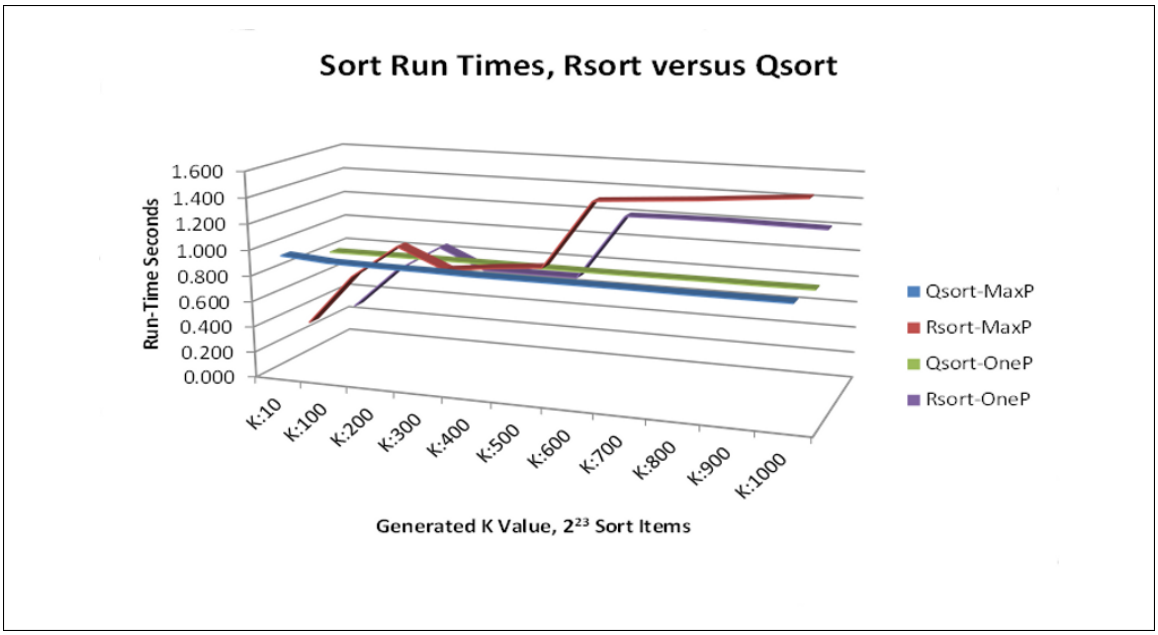

Figure 5.20: Plot of Sort Run Times for 2^{23} (8,388,608) Items

shown are the average seconds computed from three runs, performed for each combination of k value, perturbation, and sort type. For example, the literal "K:10" means the performance tests for the corresponding table line were done with data generated using the radius k value set to 10. In the column headings, "MaxP" denotes highly perturbed test data and "OneP" denotes data with a single perturbation. "Qsort" corresponds to Quick Sort and "Rsort" to our RSORT application. The data used to generate these results was of size 2^{23} or 8,388,608 items. A plot of these results is shown in Figure 5.20.

Table 5.10: Run Times, in Seconds, for Sorting 2^{23} (8,388,608) Items

Gen. K Value	Qsort MaxP	Rsort MaxP	Qsort OneP	Rsort OneP
K:10	0.948	0.329	0.822	0.285
K:100	0.919	0.734	0.823	0.619
K:200	0.912	0.998	0.823	0.840
K:300	0.910	0.842	0.822	0.678
K:400	0.904	0.886	0.821	0.678
K:500	0.904	0.919	0.821	0.678
K:600	0.904	1.443	0.820	1.195
K:700	0.903	1.470	0.821	1.202
K:800	0.900	1.496	0.822	1.205
K:900	0.900	1.528	0.821	1.200
K:1000	0.901	1.558	0.822	1.196

The blue and green lines show the run times for QSORT; blue for the highly perturbed data and green for minimally perturbed data. The results for these two are similar.

The red and purple lines show the run times for RSORT. The red line is for the highly perturbed data and purple is for the minimally perturbed data. The RSORT run-time performance was better up to K:200 and again at K:300, K:400 and one-perturbation K:500. We note the bump in the red and purple lines at K:200. Our suspicion is that this "bump" was due to the non-linear sorting performance of the comb sort. A simple, serial comb sort test was separately executed, entirely on a CPU. We performed the comb sort over differently-sized, highly-perturbed number sequences; this showed that the ratio of sorting-comparisons-count over items-sorted was not monotonic. Although the comb sort showed good performance, the comb sort's run-time performance was not linearly related to the number of items sorted. After the K:300 to K:500 range, the QSORT performance remained superior.

We note that the RSORT maximally perturbed results, the red line, climbed slightly from K:600 onwards and the RSORT minimally perturbed data, the purple line, stays rather flat. This behavior was not a surprise. The RSORT maximally perturbed data was much more out of sequence than the minimally perpetuated

data; the parallel comb sorts had significantly more work to perform.

Perhaps a more interesting observation is that the purple line stays flat from K:300 to K:500 and then flat again from K:600 onwards. This behavior was associated with the *UB* kernel's upper-bounds processing and its search for an upper bound that was a power of 2. The RSORT executions performed for K:300, K:400, and K:500 were done with a radius of 512; the K:600 to K:1000 range was performed with 1,024. With these datasets of 2^{23} items, this testing showed that roughly sorting is faster, for the k values tested, up to 200.

Single-GPU Experiment Two

We ran an additional set of tests using just the highly perturbed data, over a larger range of k values, and on two servers: Hydra and Rabbit. The hardware and software specifications for Hydra and Rabbit are described in the Appendix, Section B.1. The results of this performance testing are listed in Table 5.11 and shown graphically in Figure 5.21; the blue and green lines correspond to the Qsort processing; the red and purple lines correspond to Rsort. Both sort applications were run three times on each server and averages computed. Our earliest performance tests with these higher k values were executed only on Hydra. However, we could find absolutely no reasonable explanation for the substantial jumps in Hydra's run times at K:1100 and K:2100. After looking at low-level logs and profiling data, our next step was to add the Rabbit server to the testing environment; we wanted to observe these same substantial jumps on a second server. On Rabbit, they did not occur.

We note that the jumps tend to occur at the powers of 2: 512, 1,024 and 2,048; we also know that these were the radii values produced by the *UB* kernel and that they were used in allocation and sizing of the application's sorting blocks. Therefore, these radii values directly impacted the size of the OpenCL *work-groups*. In order to help further understand this, we again engaged the DEFG logging of the major OpenCL API calls and executed additional tests. We processed this log data into

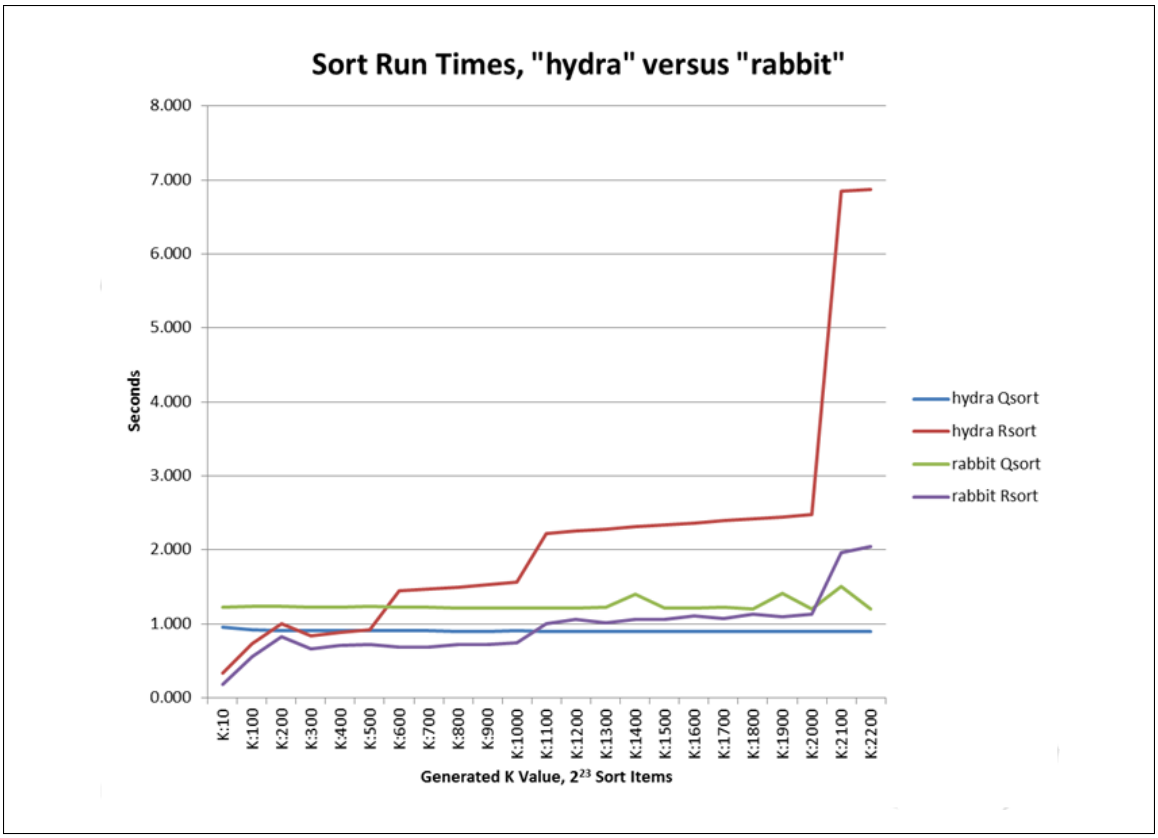

Figure 5.21: Two Server Plot of Sort Run Times with 2^{23} Items

summarized profile data. The kernel execution steps are shown in Table 5.12 and graphed in Figure 5.22. The run times of the sorting step, the comb_sort execution, for Hydra at the k value of 2100 had clearly increased. Whereas the Rabbit run-time increase from K:2000 to K:2100 was times 1.82, the Hydra increase was 3.44. At a k of 2100, the two servers were clearly behaving very differently, in terms of run-times.

Our basic explanation for this difference is that the two servers are using different GPU technologies. The Hydra server used older NVIDIA GPUs and an older OpenCL driver. The Rabbit server used smaller, but newer, AMD GPU cards with current OpenCL drivers. We suspect that the older NVIDIA OpenCL driver, present on Hydra, was not organizing it's local GPU work-groups as well as the newer, AMD OpenCL driver, used on Rabbit. With the multiple-GPU testing, where larger

datasets were used, we observed that this Hydra anomaly was *not present*. We suspect that we encountered a "bug" in the NVIDIA OpenCL driver.

Before discussing the multiple-GPU performance, we note that we analyzed the green-line-upward bumps for the Rabbit server at K:1400, K:1900, and K:2100. We attributed them to a lack of CPU memory resources. The Rabbit server had a rather small CPU, with limited RAM, and two mid-range GPU cards. Our suspicion was that "garbage collection" of CPU memory was occurring at these "bump" times.

Table 5.11: Two Server Run Times with 2^{23} Items, in Seconds

Gen. K Value	Hydra Qsort	Hydra Rsort	Rabbit Qsort	Rabbit Rsort
K:10	0.948	0.329	1.229	0.177
K:100	0.919	0.734	1.232	0.557
K:200	0.912	0.998	1.231	0.823
K:300	0.910	0.842	1.220	0.661
K:400	0.904	0.886	1.222	0.707
K:500	0.904	0.919	1.233	0.715
K:600	0.904	1.443	1.222	0.689
K:700	0.903	1.470	1.223	0.690
K:800	0.900	1.496	1.215	0.719
K:1900	0.900	1.528	1.213	0.724
K:1000	0.901	1.558	1.208	0.747
K:1100	0.900	2.221	1.209	0.995
K:1200	0.899	2.249	1.213	1.061
K:1300	0.898	2.280	1.226	1.012
K:1400	0.898	2.317	1.404	1.059
K:1500	0.898	2.335	1.212	1.059
K:1600	0.898	2.363	1.207	1.107
K:1700	0.898	2.393	1.228	1.076
K:1800	0.895	2.421	1.203	1.128
K:1900	0.895	2.442	1.408	1.100
K:2000	0.897	2.475	1.202	1.133
K:2100	0.895	6.845	1.506	1.965
K:2200	0.897	6.868	1.202	2.040

5.3.4.3 Multiple-GPU Performance

As we have previously mentioned, our view is that the DEFG provision of multiple-GPU support is valuable. We, therefore, added this capability to RSORT, forming

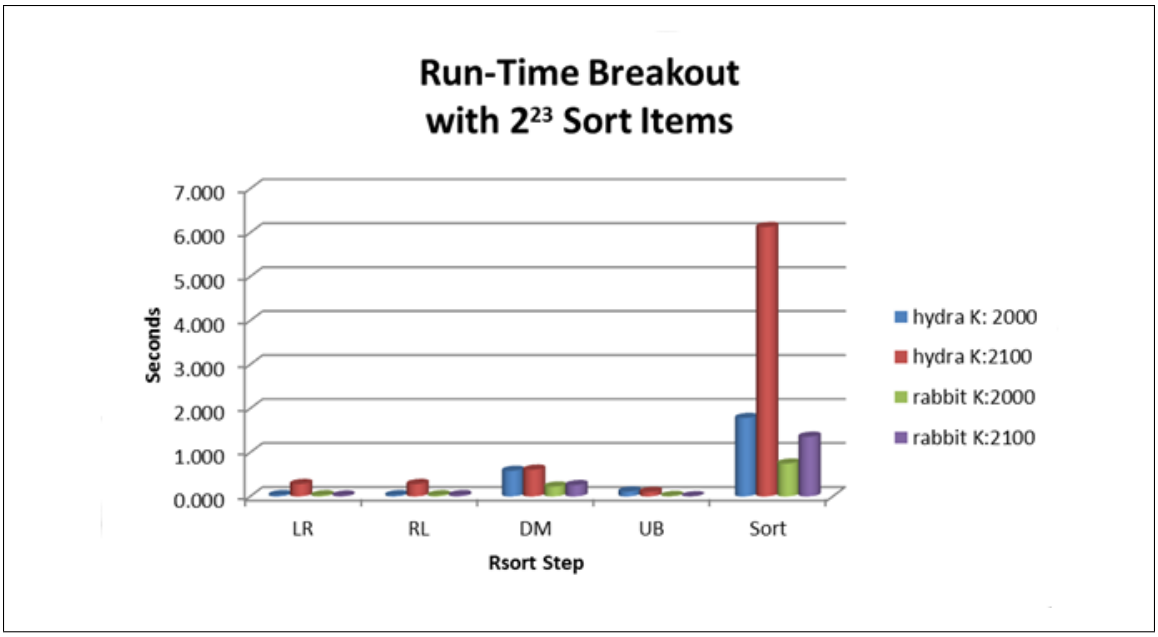

Figure 5.22: Plot of Run-Time Breakout with 2^{23} Items

Table 5.12: Run-Time Breakout with 2^{23} Items, in Seconds

Server	Data	LR	RL	DM	UB	Sort
Hydra	K:2000	0.029	0.028	0.582	0.110	1.782
Hydra	K:2100	0.280	0.280	0.607	0.100	6.127
Rabbit	K:2000	0.025	0.025	0.223	0.010	0.746
Rabbit	K:2100	0.018	0.028	0.256	0.004	1.355

RSORTM. The use of two GPUs with roughly sorting allowed us to obtain sorted results faster and to sort larger data sets. The multiple GPU support was implemented using the Divide-Process-Merge design pattern.

Figure 5.23 shows a very abbreviated listing of DEFG code used in RSORTM. RSORTM is built upon RSORT; most of the lines of code shown in this figure are additions to the code shown in Figures 5.18 and 5.19. Line 22 selected multiple GPUs, lines 27 and 28 declared the additional kernels used in the UB loop, and line 36 declared the extra buffer also used in the UB loop processing. Lines 46 through 69 were a repeat of the original left-to-right maximum code with the DEFG execute statements replaced by multi_exec statements.

Continuing with lines 71 to 77, we observe that the UB processing in RSORTM

differs from the approach used in RSORT. Here, we encounter a small DEFG design restriction. The DEFG approach used to split buffers into equal portions, with a portion given to each GPU, does not work with scalar variables. We cannot reasonably split a scalar variable in half. The upshot of this shortcoming was that the *again* variable used to manage this loop had to be set with the values returned from the *againPart* buffer. The morsel code at line 76 combined the two values from the buffer. The *UB* kernel, used in RSORT, was replaced by the new *UBreset* and *UBsplit* kernels. Kernel *UBreset* did reset the buffer data and *UBsplit* compared the *radius* upper bound with the *DMbuf* contents. The OpenCL code for these simple kernels is shown in the Appendix, Section B.4.

The *putMergeArray()* function, not shown in this abbreviated listing, was called to copy the two sorted data segments to disk. They were merged into one sorted segment as they were copied. This was the merge step inherent in the DEFG Divide-Process-Merge design pattern.

Next, we compare the performance of RSORTM against both RSORT and QSORT. The testing was performed on Hydra using test data files of 2^{26} and 2^{27} sort items.[17] Tables 5.13 and 5.14 show the testing results and these are shown graphically in Figures 5.24 and 5.25. We note that the unexpected drop in Hydra performance seen in the previous testing was not present with these larger test datasets.

Looking at Figure 5.24, we note the RSORTM green line is always below the RSORT red line. With this test data, RSORTM was consistently faster than RSORT. We also note that the RSORTM performance was faster than QSORT up to a k value of 2000. The addition of the second GPU made roughly sorting the faster solution for a wider range of k values. Looking back at the image filtering applications discussed previously, and comparing the applications, RSORTM clearly has enough work to utilize both of the GPUs.

[17]The Rabbit server was not able to handle data files of this large size, due to disk and memory size limits.

```
    ...
04. declare application  RSortm
    ...
22.   declare gpu gpugrp ( all )
    ...
27.      kernel UBsplit   RSort_Kernels  ( [[ 1D,size ]] )
28.      kernel UBreset   RSort_Kernels  ( [[ 1D,size ]] )

    ...
36.      integer buffer againPart (againSize)
    ...
46.   multi_exec LR1 LRmax (arrayS(in) LR(out) stride(in))
    ...
50.   loop
51.     multi_exec LR2 LRmax (LR(inout) LRout(out) stride(in))
    ...
55.   while again lt logSize
    ...
69.   multi_exec DM1 DM (LR(in) RL(in) DMbuf(out))
    ...
71.   loop
72.     multi_exec UB1 UBreset (againPart(inout))
73.     call times2(radius(inout))
74.     multi_exec UB2 UBsplit (DMbuf(in)
                                 againPart(inout) size(in) radius(in))
75.     call sync (againPart(in))
76.     code [[again = againPart[0] + againPart[2]; ]]
77.   while again ne 0
    ...
80.   code [[ if (groups<2) {printf("sort end, too few sort groups, ... ]]
    ...
82.   code [[ groupsMulti = groups / DEFG_GPU_COUNT; ]]
83.   multi_exec SORT1 comb_sort(arrayS(inout) ... groupsMulti(in))
    ...
88.   multi_exec SORT2 comb_sort(arrayS(inout) ... groupsMulti(in))
    ...
91. end
```

Figure 5.23: Abbreviated RSORT DEFG Executable Statements

Due to lack of memory on a single Hydra GPU card, RSORT cannot be executed with the 2^{27}-sized dataset. RSORTM, with access to twice as much GPU memory, was able to process this large dataset. The performance results are presented in Figure 5.25. We observe that the cross-over point with this dataset was similar to the 2^{26} dataset, the performance crossover was at the k value of 2000.

5.3.4.4 RSORT Run-Time Performance

As expected, using our artificially generated, partially sorted datasets, RSORT was shown to produce good performance results, relative to CPU-based QSORT, when the k-value was approximately 100, or less. This high-level of performance occurred

with both the single-GPU and multiple-GPU RSORT usage modes; when more than a single GPU was used by RSORT, sorted results were produced even more quickly. The use of multiple-GPUs also permitted the sorting of larger datasets. For the vast majority of our run-time tests, highly perturbed datasets were used. For this reason, our view is that most other datasets, with equal k values, would be likely to experience faster performance, compared to our generated datasets. For k values over about 100, we suggest that the size and distribution of the data must be taken into consideration when looking at the general use of roughly sorting.

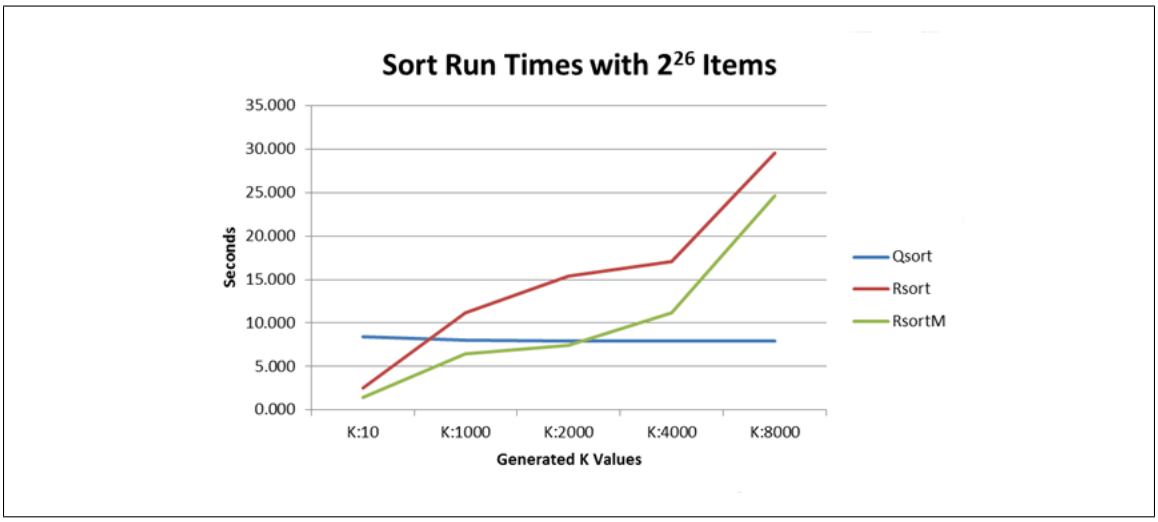

Figure 5.24: Plot of Sort Run Times with 2^{26} Items

Table 5.13: Sort Run Times on Hydra with 2^{26} Items, in Seconds

Program Name	Gen K: 10	Gen K: 1000	Gen K: 2000	Gen K: 4000	Gen K: 8000
Qsort	8.394	8.008	7.972	7.922	7.890
Rsort	2.527	11.216	15.360	17.120	29.556
RsortM	1.459	6.487	7.400	11.189	24.682

Table 5.14: Sort Run Times on Hydra with 2^{27} Items, in Seconds

Program Name	Gen K: 10	Gen K: 1000	Gen K: 2000	Gen K: 4000	Gen K: 8000
Qsort	17.317	16.539	16.460	16.389	16.318
RsortM	2.912	11.896	16.447	19.613	31.243

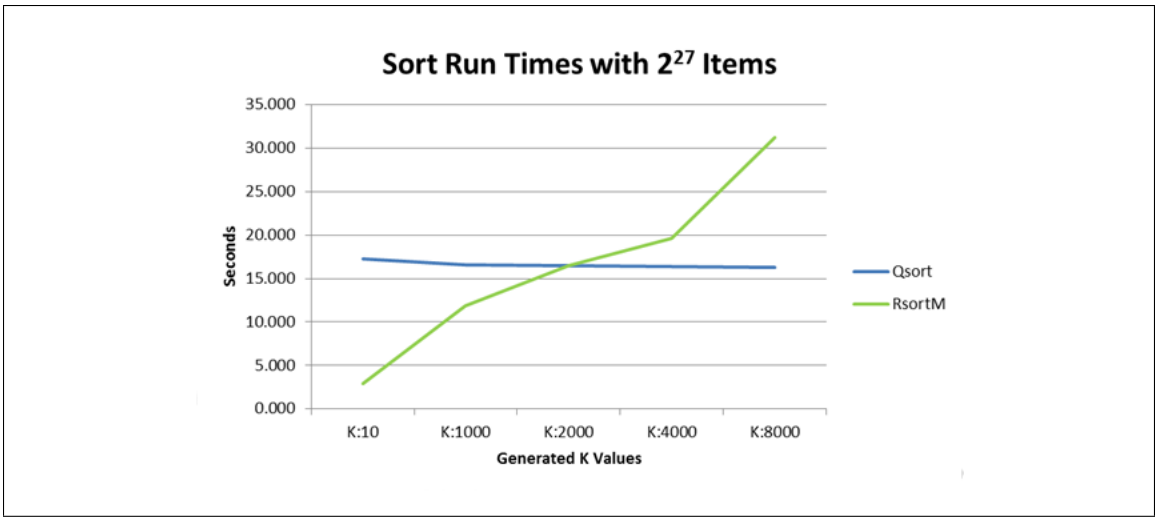

Figure 5.25: Plot of Sort Run Times with 2^{27} Items

5.4 Application: Altman Method of Matrix Inversion

5.4.1 Problem Definition and Significance

Matrix inversion is a well-studied problem with application to many endeavors, including the solution of simultaneous equations in Engineering, Economics, and Chemistry [72, 13, 5, 74]. Further, simultaneous equations need to be solved in real-time robotics and these solutions are required in a limited time interval [46]. The use of the *iterative matrix inversion*, in the context of *anytime* algorithms [77], has the potential to supply the solutions for simultaneous equation in real-time environments. The use of simultaneous equations in robotics is demonstrated in numerous works [28, 44, 63].

There are many methods for inverting matrices. One such method is M. Altman's iterative method; it has the potential for inverting matrices with high degrees of adaptability, scalability and robustness, due to the iterative nature of its design [7, 8]. Iterative inversion consists of making an initial estimated inversion and then performing additional algebraic operations to improve the quality of the inversion. In this work, we use the notion of anytime algorithms to manage our GPU-based Altman inversion processing. Anytime algorithms use a *well-defined quality measure*

to monitor the solution progress and then allocate resources effectively [101]. This anytime algorithm approach makes it possible to offer a tradeoff between solution quality and computational time [36].

Assume we have a matrix A and we wish to find its inverse, R. The Altman method requires we have an R_0 such that $||I - AR_0|| < 1$ and then we iteratively apply $R_{n+1} = R_n(3I - 3AR_n + (AR_n)^2)$. We implemented a GPU-based anytime algorithm version of Altman's iterative method using DEFG and OpenCL, and then performed an analysis of the inversion results.

Our efforts had a focus on DEFG and the demonstration of DEFG's capabilities. As such, we provided a DEFG-based, GPU implementation of the iterative Altman numerical method, with added anytime processing. This was done to help show the range of problems and applications handled by DEFG. We did not explore the numerical intricacies of the actual Altman method. Matrix multiplication was used as a building block, and an existing OpenCL matrix multiplication facility, clMath [2], was used to satisfy our requirement for a high-speed matrix multiplication capability.

This Altman inversion method has some interesting adaptive characteristics [7, 8]. It is tolerant of errors in computation and initial settings. The inversion results are dependent on the initial R_0 and on the number of iterations performed. More iterations tend to compensate for a poorly set R_0 or poorly computed R_n. If R_0 is not otherwise available, it can be formed with $R_0 = \alpha I$, where one possible value of α is $1/||A||$, with $||A||$ being the Euclidean norm of A. Different values of α lead to different convergence characteristics. We use the Euclidean norm of $I - AR_{n+1}$ as our inversion quality measure. This scalar value decreases as the quality of the inversion increases. These characteristics provide the ability to use an anytime algorithm to limit the number of iterations, enabling the desirable option to choose between solution quality and computational time.

5.4.2 Related Work

5.4.2.1 Anytime Algorithms

Boddy and Dean coined the term "anytime algorithm" in their work on time-dependent planning [17]. They describe anytime algorithms as, "algorithms that can be interrupted at any point to supply an answer whose quality increases with increasing computational time." The work of Zilberstein brings the characteristics of anytime algorithms into clearer focus with his list of desirable anytime traits: *measurable quality, recognizable quality, monotonicity, consistency, diminishing returns, interruptibility,* and *preemptibility.* Zilberstein makes an additional point, perhaps best given as a direct quote, "Anytime computation extends the traditional notion of computational procedure by allowing it to return many possible approximate answers to any given input" [101].

Anytime algorithms have previously been implemented on GPUs by Sab and Manharam [77]. They demonstrate the use of their *PAP** parallel search algorithm in which they compose a pool of CUDA kernels in advance and then at run time, depending on the load and quality-level goals, select an appropriate set of kernels to launch.

5.4.2.2 Iterative Approach

This iterative approach distinguishes itself from direct, decomposition inversion approaches such as Cholesky Decomposition and Gauss-Jordan by producing useful intermediate inversion results [52, 90]. The iterative approach used here was outlined by M. Altman in his work: *An optimum cubically convergent iterative method of inverting a linear bounded operator in Hilbert space* [7]. A few years later, Petryshyn improved the error estimates for the M. Altman method [73]. More recent works in this area include the Stanimirovic article: *Self–correcting iterative methods for*

computing {2}–inverses [89]. Stanimirovic pointed out that this class of iterative inversion algorithms will self-correct only if the matrix is in fact invertible.[18]

5.4.2.3 GPU-based High-Performance Matrix Libraries

The Altman algorithm requires numerous matrix operations, including matrix addition, subtraction, and multiplication. We used an existing GPU high-performance library: clMath, formerly called APPML [4]. clMath is the *Accelerated Parallel Processing Math Libraries* which contains the Basic Linear Algebra Subprograms (BLAS) and the Fast Fourier Transform (FFT) functions. It is written for OpenCL and designed to run on AMD GPUs [2] and can be used with OpenCL GPUs from other vendors. In 2013, AMD re-branded APPML as "clMath" and converted it to an Open Source product. NVIDIA supplies cuBLAS. cuBLAS is the NVIDIA Basic Linear Algebra Subroutines library for use with their hardware. Unfortunately, cuBLAS is not compatible with OpenCL. This is unfortunate since cuBLAS has been available longer than clMath, is vendor supported, and is likely a more stable and mature linear algebra library.

5.4.3 Application Software Design

5.4.3.1 The Implementation of Altman's Method in DEFG

Our DEFG implementation of Altman's iterative method of matrix inversion has gone through a number of versions. Each version was verified to be sure that correct inversion results were produced.[19] In our earlier versions, we used the clMath capabilities for all matrix operations. Testing, and periodic reviews of our IMI versions, revealed that our early DEFG IMI source code was not easily read and visualized by

[18] A fact that we can attest to after accidentally passing a singular matrix to our first GPU version of the Altman method!

[19] The results were verified by comparing the matrix inversion results from our IMI applications with corresponding MATLAB inversion results.

the developer, and the general application performance was somewhat slower than we had expected. Because of these issues, we changed our IMI application design and implementation. Our final IMI version, called IMIFLX, used the clMath library only when absolutely needed. For the simpler operations, such as establishing the identity matrix or multiplying a matrix by a constant, we used a different approach. The new approach used specialized GPU kernels to multiply a matrix by a scalar and used CPU-side DEFG morsels to initialize matrices.

The final version, IMIFLX, has the "FLX" name suffix because of the flexible manner in which it obtained the square matrix to be inverted. The matrix was loaded from an input file or was internally generated. The test matrices generated by IMIFLX were Hilbert matrices [99], identity matrices, or "invertible" matrices. The invertible matrices had their non-principle diagonal values set to 1 and their principal diagonal values set to the integer values $1, 2, \ldots, N$; N being the width of the matrix [75].

For the sake of brevity, we will limit our presentation of the IMIFLX application's code to an abbreviated listing, which shows most of the application's non-declarative code. The code is shown in Figure 5.26. The figure's comment lines tend to describe the linear algebra step performed next; the following statements beginning with blas or execute perform the corresponding DEFG operations. The code statements, which are DEFG morsels, inserted the associated C++ code into the application. As an example, the code statements at lines 2 and 3 established an identity matrix multiplied by the previously established α value. There was one blas statement before the loop, and each loop iteration required the execution of three blas statements. The small kernels executed at lines 11, 13, and 21 performed matrix multiplication by a constant. The kernel executed at line 19 performs a matrix (buffer) copy. The code for these simple kernels is shown in the Appendix, Section B.4. The earlier IMI versions used blas statements instead of these kernels. The performance implications of this change

are discussed in the next section.

The *loop_escape* statement, at line 29, demonstrates an example of DEFG *anytime* processing. DEFG maintained an internal timer that tracked the run time of the loop and the loop_escape statement was used to exit the loop when a run time threshold had been exceeded. Since the Altman algorithm provided an inversion result that improved monotonically with each iteration, this algorithm was appropriate for anytime algorithm usage.

The three kernel executions listed together, at line 23 through 25, warrant some explanation. The calculation of the required Euclidean norm, used here to evaluate the quality of the inversion, required that the squares of all matrix values be summed. Parallel prefix scan provided a high performance approach to obtaining this sum [40]. However, many of the available high-performance prefix scan algorithms require an

```
01. // mA holds the matrix to be inverted; mRn = Identity Matrix * alpha
02. code [[ for (int i=0; i < mSIZEt2; i++) {  mRn[i] = 0.0; } ]]
03. code [[ for (int i=0; i < mSIZE; i++) {   mRn[i*mSIZE +i] = alpha; } ]]
04. // mP = mA * MRn
05. blas (dOne * mA * mRn + dZero * mP -> mP)
06. loop
07.   // desired result: mRnp1 = mRn * (mI*3 - mP*3 + mP2)   mP2 is mP*mP
08.   // mW = mP * mP
09.   blas (dOne * mP * mP + dZero * mW -> mW)
10.   // mW += mI * 3
11.   execute k8 PlusIdentityThree(mW(inout) mSIZE(in))
12.   // mW -= mP * 3
13.   execute k9 MinusMatThree(mW(inout) mP(in) mSIZEt2(in))
14.   // mRnp1 = mRn * mW
15.   blas (dOne * mRn * mW + dZero * mRnp1 -> mRnp1)
16.   // mP = mA * MRnp1
17.   blas (dOne * mA * mRnp1 + dZero * mP -> mP)
18.   // copy mP to mW;
19.   execute k10 CopyArray(mW(out) mP(in) mSIZEt2(in))
20.   // mW -= mI
21.   execute k11 MinusIdentity(mW(inout) mSIZE(in))
22.   // result = norm(mW)
23.   execute k12 SweepSquares(mW(in) mSIZEt2(in) mBasket(inout) basketSize(in))
24.   execute k13 prefixSum(mS (out) mBasket(in) mLocal(*) basketSize(in))
25.   execute k14 ReadLastSqrt(mS(in) basketSize(in) result(out))
26.   release (result)   // gets value onto CPU
27.   // cpu: compare epsilon and result
28.   code [[ if (result <= epsilon) LCNT = cycles;  ]]
29.   loop_escape at 6 secs   // "anytime" processing
30.   // mRn <==> mRnp1
31.   interchange(mRn mRnp1)
32.   call inc(LCNT(inout))
33. while LCNT lt cycles
```

Figure 5.26: IMIFLX Application Processing Loop

input data array size that is a power of 2. This prefix scan and power-of-2 topic was mentioned in Section 5.3, when discussing BFSDP2GPU. With our IMI application, we did not want to limit our processing to matrices that have a power-of-2 number of elements and we wanted to have improved performance over the Berman Prefix Scan algorithm's work upper bound of $O(n \log n)$. We, therefore, split the norm processing into three kernels and we introduced a power-of-two-sized intermediate buffer to hold partial sums. Our improved approach was possible because the IMI norm processing only needed the final sum of squares and did not need any of the preceding partial sums.

Our approach stored the partial sums in the *mBasket* buffer, which contained *basketSize* elements. The *basketSize* value was set with a power-of-2 value that was somewhat larger than the maximum number of threads the GPU provides; for Hydra we used a value of 1,024. Each GPU thread produced a small number of sums and these sums were stored in *mBasket*. This work was performed by the *SweepSquares* kernel; it summed the *squares* of the values in the matrix. The code for this kernel, *SweepSquares*, is shown in Figure 5.27. The for loop, in the kernel, incremented the k index value by a stride of *basket_length*, to minimize the GPU hardware's global memory accesses.

The *prefixSum* kernel was taken from the AMD OpenCL SDK. It was modeled after the approach taken by Harris, et al. [39] and Blelloch [15]. Here we used it to sum the partial sums harbored in the *basketM* array. The simple *ReadLastSqrt* kernel returned the square root of the last sum produced by the *prefixSum* kernel. The upper bound on work for this approach is $O(n + \log(basketSize))$ and since $basketSize$ is a constant, the upper bound with a large matrix is actually $O(n)$. The source code for the *prefixSum* kernel is available from the AMD OpenCL SDK and the *ReadLastSqrt* kernel code is available in the Appendix, Section B.4.

```
01. __kernel void SweepSquares(
02.                     __global double* input,    // buffer of data values
03.                     const int length,        // full length of buffer 03.
04.                     __global double*  basket, // basket of partial sums
05.                     const int basket_length)  // full length of basket
06. {
07.   double d;
08.   double sum = 0.0;
09.   if (length < 1) return;
10.   unsigned int tid = get_global_id(0);
11.   if (tid >=length) return;
12.   // make strides of basket_length width
13.   for (int k=tid; k < length; k += basket_length) {
14.     d = input[k] * input[k];
15.     sum += d;
16.   }
17.   basket[tid] = sum;
18.   return;
19. }
```

Figure 5.27: *SweepSquares* Kernel Source Code

Table 5.15: Comparison of M1000 Run Times

IMI Version	BLAS Seconds	BLAS Count	NDR Seconds	NDR Count	Summed Seconds	Actual R.T. Average Seconds
IMIFLX	0.047	43	0.0005	101	2.02	2.113
IMIB	0.047	99	0.0009	45	4.65	4.711

5.4.4 Experimental Results

5.4.4.1 Run-Time Performance of IMI Versions

In order to obtain a sense for the cost of each BLAS API call and the relative performance of our final IMIFLX version, we performed run-time comparisons between an earlier version of our application and the final version. These tests were run on Hydra with the DEFG logging engaged, using an invertible matrix of width 1000, called *M1000*. We used this logging information to compute the average time consumed in BLAS API calls and in OpenCL *clEnqueueNDRangeKernel()* calls. The results, which are the averages of three runs for each case, are shown in Table 5.15.

The earlier version of IMIFLX, called IMIB, used DEFG blas statements for all matrix operations, except initializing the matrices. The IMIFLX version avoided blas statements whenever possible; instead it used the *PlusIdentityThree*, *Minus-*

MatThree, *CopyArray*, and *MinusIdentity* kernels. With the *M1000* matrix, the measured run times dropped from 4.711 seconds to 2.113, a speedup of 2.22. In Table 5.15, the Summed Seconds column contains the sum of the BLAS and the clEnqueueNDRangeKernel API execution run times. The 2.02 seconds IMIFLX value was calculated using $(0.047 * 43 + 0.0005 * 101)$. We note that the BLAS and NDRange processing account for the lion's share of the actual run time. Looking at the BLAS Count and NDR Count columns, we can see that each application version generated 144 requests. In the case of IMIFLX, 43 were BLAS requests; IMIB had 99. Our IMIFLX application version was faster than IMIB because it made many fewer BLAS requests. The kernel requests, which replaced the omitted BLAS requests, were much faster, with run times less than 0.001 seconds, versus the 0.047 seconds for each BLAS request. Not surprisingly, the `blas` statements were the dominant factor in the run times.

5.4.4.2 Inversion Results and Anytime Processing

As the IMIFLX application used the Altman iterative approach, it converged to a solution. It stopped when: the computed Euclidian Norm was less than the specified *epsilon* value; a numeric overflow had occurred; the maximum number of iterations count had been matched, or the anytime processing had intervened. The table in Figure 5.28 shows the Euclidean Norm values for processing an *M500* matrix, with an epsilon setting of 0.00001. The plot in Figure 5.28 presents this data; the Iteration count is on the X axis and the Norm value is on the Y axis. We can see that the application's Norm values monotonically decreased from 20 towards zero. On the 13th iteration, the Norm was less than 0.00001 and the processing stopped.

Table 5.16 shows the results from 20 sample executions of IMIFLX, with different matrices. For each execution, the table provides name, type, and size for the matrix and the epsilon value, number of iterations, and total run time from the actual ex-

M500 Iteration	Norm Value
1	19.905700
2	16.298600
3	10.775400
4	6.320110
5	3.687980
6	2.186990
7	1.362090
8	0.943485
9	0.684855
10	0.320215
11	0.032834
12	0.000035
13	0.000000

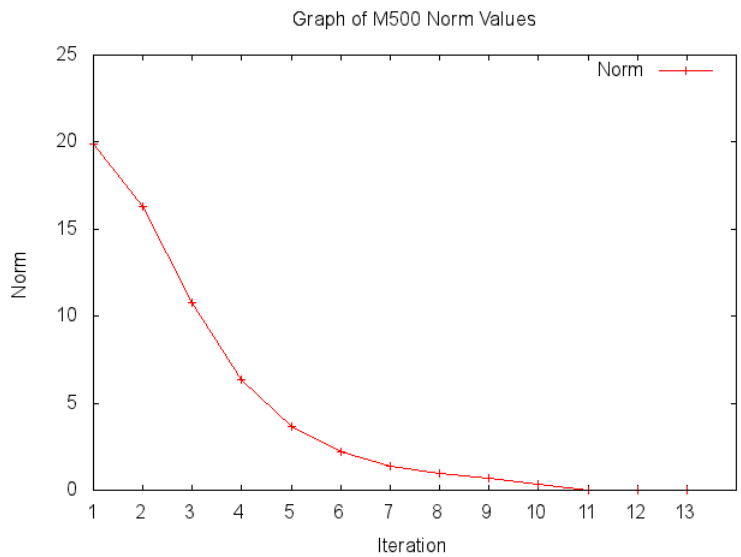

Figure 5.28: Table and Plot of M500 Matrix Norm Values

ecution. Line 13 shows the results of the described *M500* execution; it required 13 iterations and consumed 0.259 seconds. The next line, 13a, is a second execution of the same matrix but with the anytime limit set at 0.2 seconds. Here we see that the processing consumed 0.206 seconds and used 10 iterations. If we look back at the table shown in Figure 5.28, we see that the Norm was 0.320215 at this point. When the execution was terminated by the anytime processing, the application had brought the norm to within 2% of its full-run final value.

5.4.4.3 Range of IMI Inversions

Lines 1 through 12, of Table 5.16, present the results of inverting a set of the notoriously difficult-to-invert Hilbert Matrices [99]. The matrix sizes ranged from 2×2 to 13×13. Our inversion application was able to invert these matrices up to size 12×12. At 13×13, an #INF ("Infinity") C/C++ run-time result occurred. This error was returned when the result of an operation was too large to be accurately stored. Starting at size 10×10, the epsilon value was raised; these larger epsilon

values were the approximate lower bound that the norm value could achieve.[20] Using lower values of epsilon, with these cases, tended to iterate until the max-iteration count was encountered.

In order to determine an approximate upper bound on matrix size that our application could handle on the Hydra server, we inverted matrices of increasing size. Line 16 shows an 8000×8000 matrix being handled successfully; however, the 8500×8500 instance at line 17 fails with an OpenCL error of -4. This OpenCL error was discussed previously in Section 5.1; it occurs when the GPU cannot allocate the requested global memory. The IMI inversion of this matrix exceeded the memory capability of the NVIDIA Tesla T20 on Hydra, which has 2,687M of RAM. Our application required five matrix-sized buffers. With the 8000×8000 matrix, each buffer required 512M bytes, five of these needed 2,560MB. For 8500×8500, each buffer required 578MB of RAM, five of these needed 2,890MB. The larger 8500×8500 matrix exceeded the memory capacity of the Hydra T20 GPU.

The last three matrices listed were obtained from the *University of Florida Sparse Matrix Collection* [24, 25]. Matrix *685_bus* is from a power network problem with 3,249 non-zero values and matrix *1138_bus* is a similar problem with 4,054 non-zero values. The much larger *Kuu* Matrix is from a structural problem and has 340,200 non-zero values. Our iterative inversion application encountered no issues with these real-world matrices.

In this section, we demonstrated DEFG handling our OpenCL numerical application, which used the clMath BLAS library. We obtained improved performance by using kernels for the non-multiplication matrix operations, using the BLAS functions only when absolutely needed. We showed that the M. Altman approach to matrix inversion works well on GPUs, and showed the useful DEFG *anytime* processing performing an early exit of an iterative process, when triggered by an external event.

[20] We note that double-precision data types were used here and that running this OpenCL application on a CPU produced very similar results.

Table 5.16: IMIFLX Inversion Results for Various Matrices

Cnt	Matrix Name	Type	Size	Epsilon	Iterations	Run Time Seconds
1	H2	Hilbert	2x2	0.00001	4	0.018
2	H3	Hilbert	3x3	0.00001	8	0.022
3	H4	Hilbert	4x4	0.00001	12	0.023
4	H5	Hilbert	5x5	0.00001	15	0.030
5	H6	Hilbert	6x6	0.00001	18	0.034
6	H7	Hilbert	7x7	0.00001	21	0.036
7	H8	Hilbert	8x8	0.00001	24	0.037
8	H9	Hilbert	9x9	0.00001	27	0.042
9	H10	Hilbert	10x10	0.001	30	0.035
10	H11	Hilbert	11x11	0.005	40	0.057
11	H12	Hilbert	12x12	0.15	70	0.089
12	H13	Hilbert	13x13	n.a.	n.a	#INF error
13	M500	Invertible	500x500	0.00001	13	0.259
13a	M500	Invt-AnyTime	500x500	0.00001	10	0.206
14	M1000	Invertible	1000x1000	0.00001	14	2.112
15	M5000	Invertible	5000x5000	0.00001	16	329.619
16	M8000	Invertible	8000x8000	0.00001	17	1380.320
17	M8500	Invertible	8500x8500	n.a.	n.a.	error -4
18	685_bus	Repository	685x685	0.00001	12	0.665
19	1138_bus	Repository	1138x1138	0.00001	14	3.262
20	Kuu	Repository	7102x7102	0.00001	9	605.310

CHAPTER VI

ACCOMPLISHMENTS, OBSERVATIONS, AND FUTURE RESEARCH

While envisioning, designing, implementing, testing, and debugging DEFG and developing new DEFG-based applications, we became very familiar with the process of developing GPU applications. Based on this familiarity and our associated expertise, we offer this chapter. It summarizes our DEFG accomplishments, lists our noteworthy observations, and suggests interesting future research.

6.1 DEFG Accomplishments

In this dissertation, we produced the DEFG parser, optimizer, and code generator. These three, along with several DEFG design patterns, give DEFG its unique ability to generate the CPU-side of OpenCL applications, as seen in the aforementioned SOBEL, MEDIAN, BFSDP2GPU, RSORT, and IMIFLX applications. Each of these DEFG applications explored a different aspect of GPUs and OpenCL. The SOBEL and MEDIAN image filter applications used DEFG with image filtering, with a focus on multiple-GPU processing. Our 5×5 median image filter showed a speed up with multiple GPUs. Our graph-processing application, BFSDP2GPU, showed the DEFG ability to process large very irregular data structures with multiple GPUs. The GPU sorting RSORT application implemented the novel roughly sorting algorithm for partially sorted data. It demonstrated good performance in both single-GPU and

multiple-GPU versions. The last application, our iterative matrix inversion implementation IMIFLX, exhibited DEFG's ability to implement iterative, GPU-based, numerical processing.

DEFG is designed to make the development of the CPU-side of OpenCL applications less work for the set of applications that follow one, or more, of the DEFG design patterns. In Chapter III, we showed that DEFG-produced applications can match the speed of equivalent hand-written applications.

The value of DEFG is clear: it enables the implementation of OpenCL applications using the declarative approach. This approach to application development is advantageous; it is likely to consume far less developer time, compared to writing hand-written C/C++, because the developer has to write many fewer lines of source code, and the DEFG code that is written is simpler, relative to the same hand-written C/C++.

6.2 Some Noteworthy Observations

6.2.1 Usefulness of a DSL for GPU Software Development

What is it about the style of software development used with GPUs that makes the use of a DSL attractive? Our answer to this question is based on Farber's suggested rules for GPGPU programming [30], paraphrased here: (1) get the data on the GPU and leave it there, (2) give the GPU ample work to do, and (3) focus on data reuse within the GPU to avoid global memory limitations.

This high-performance GPU methodology comes down to putting the computational work on the GPU and utilizing the CPU mainly to manage the GPU's operations. GPU developers strive to minimize the transfer of data to and from the GPUs, due to the transfer's significant consumption of time. The application work is done in the kernels; kernels need to be efficient, and often, highly optimized. The CPU application code facilitates the GPU's execution; it manages the *movement* of data

to and from the GPU, and it handles *scheduling* of the GPU kernels.

Our view is that these CPU-based data movement and GPU scheduling operations can be quite often expressed with a set of predefined design patterns. Our DSL, DEFG, supplies these design patterns. DEFG makes it possible to declare the characteristics of the GPU-required data and, using design patterns, manage the needed GPU operations. With the declarations made and design patterns chosen, DEFG then produces the corresponding CPU-side C/C++ program. Our experience has been that most application variability and complexity tend to be concentrated in the GPU kernels.

6.2.2 Declarative-Approach for Kernel Code Generation

Having observed that DEFG's declarative approach can work well for generating the CPU-side of OpenCL applications, we explored using a similar declarative approach for GPU kernel generation. We are not as optimistic that the declarative approach will work well for the GPU kernels due to the wide variations we observed in kernel processing.

In our view, one reason that DEFG works well is because of the supplied design patterns. As stated above, many GPU applications require their CPU-side code to perform similar actions; these similar actions often map well into patterns.

Our point here is that with GPU kernels, we did not find the same similarity of actions. We reviewed the kernels used in our port of existing applications and our newly developed applications, and we observed very few common kernel usage patterns that are significant and substantial. The few similar patterns seen on the GPU-side were trivial actions, such as locating the index to a required buffer element or making certain a given index offset value was within the buffer size. Different applications tended to use substantially different kernel processing.

6.3 Conflicting DEFG Aims and Static Optimization

At the outset of this project, two of our principle objectives for DEFG were: (1) to simplify the GPU software development process, and (2) to generate human-readable code. It became obvious that these objectives were somewhat in opposition to each other. Generating human-readable code implies producing modules that are reflective of the programming logic being used.

In other words: to be readable, the modules need to be coded in a straightforward manner and uncluttered. However, as we introduced significant performance optimizations, we saw that the static optimization step required inserting additional code for many special cases. For example, when optimizing the buffer transfers inside a looping structure containing multiple loop exit points, the optimizer should consider the buffer transfer requirements for each exit point, and generate the appropriate code. With static optimization, these special cases may force the generation of significant amounts of specialized logic, at many locations within the code. The insertion of this specialized code can make the generated code hard to understand and seem cluttered. One reason we decided to construct an *external* DSL was to generate human-readable code. To our surprise, we found that it is not always possible to generate uncluttered, readable code, when high-levels of performance is also a major concern.

6.4 Future Research

6.4.1 Additional DEFG Design Patterns

DEFG has demonstrated the ability to generate standalone applications and callable C/C++ functions. DEFG has the potential to be even more useful if two additional design patterns are implemented: multiple-GPU load balancing and resource sharing.

Our current DEFG implementation supports multiple GPUs but does not, in and of itself, make any attempt to balance the workload between the selected GPUs. The

workload assignment is a function of how the application is written. This approach works well on a system with matched GPU devices. However, if the selected GPUs are not matched, or the application does not assign the workload to the devices appropriately, DEFG-supplied load balancing could be of great value. A DEFG design pattern that dynamically allocates work to the selected GPUs could compensate for mismatched GPUs and hard-to-predict workloads.

Similar to load balancing, resource sharing could become an issue. This issue may arise when a DEFG-generated program is utilized as a C/C++ function and the function is used from within a loop. The current DEFG obtains and releases its resources on every invocation. This behavior could be problematic if the DEFG-generated function is constantly re-invoked, after performing only a small amount of work. A design pattern that allowed for the holding, and sharing, of resources could be useful in preventing this loss of performance associated with repeated allocation and release of resources.

6.4.2 DEFG Support for CUDA

We suggest designing and implementing a version of DEFG for NVIDIA's CUDA. CUDA is a widely used platform for generating NVIDIA-specific GPU applications. As noted earlier in this work, CUDA has much better support for direct GPU-to-GPU communications than OpenCL provides. While this is a worthwhile goal, we see two major obstacles to implementing CUDA support in DEFG; the first is rather obvious and the second is more subtle.

First, the CUDA environment is similar to the OpenCL environment but definitely not equal. For example, the CUDA terminology greatly differs from that of OpenCL; the GPU kernels have a different syntax; and, the actual CPU-side call-level APIs have a different *granularity* compared to OpenCL [65, 70]. In our experience, for equals problems, CUDA-based application solutions require fewer API calls. As

a result of these types of differences, the DEFG code generator would require significant refactoring to elegantly support CUDA. The parser and optimizer would not require such refactoring. The BFSDP2GPU application has the potential to perform much better with CUDA, due to the CUDA's superior GPU-to-GPU communication capabilities.

Additionally, the OpenCL environment does not require that the *local work-group* value be supplied on OpenCL *clEnqueueNDRangeKernel()* API calls; setting this parameter is optional. DEFG makes significant use of this OpenCL feature. With CUDA, the equivalent to this parameter, *threads per block*, is required and it is limited to certain values depending on the CUDA block-size value provided. A version of DEFG for CUDA would have to find an appropriate automated way to set this parameter or force the DEFG application software developer to provide a reasonable value.

While challenging, implementing a CUDA version of DEFG would be worthwhile because of CUDA's wide-scale use and the added performance it could give to applications, such as BFSDP2GPU.

6.4.3 DEFG Re-factored as an Internal DSL

DEFG is currently designed in the style of an *external* DSL; it consists of a parser, optimizer, and code generator. The generated program is compiled by a standard C/C++ compiler. After making significant use of the current DEFG, we suggest that producing an *internal* DSL, using many of the DEFG components, would produce a very useful GPU development tool.

Martin Fowler suggests that DSLs can be categorized in two ways: *internal* DSLs and *external* DSLs [32]. An *internal* DSL is constructed inside a standard programming language through the use of objects, macros, and other programming language extensions. This new DSL could be implemented as a set of specialized objects in

an object-oriented language such as C++, C# or Java.[1] The items declared in the current DEFG design could become programming objects, and the high-value DEFG buffer transfer optimizations could be implemented as programming objects centered upon the basic OpenCL API functions. Great care in designing this new internal DSL would need to be taken, so as to ensure that the data structures used are consistent with the constraints of the OpenCL API functions. In order to maintain a high level of performance, this new implementation cannot simply copy data buffers between template-style, object-oriented arrays and the simple, continuous, "flat" buffers of memory required by the OpenCL APIs. If this were done, the run-time cost of this additional copying would likely to be prohibitive, when the buffers being used are large.

In this approach, many DEFG design patterns would become higher level objects, making use of the just-described programming objects. With this new DSL in place, the current DEFG could be re-implemented and greatly simplified. The DEFG parser would remain largely as is.

6.5 DEFG Technical Improvements

As a result of our work, we discovered a number of technical improvements and enhancements that could be added to DEFG. They are paraphrased here, in summarized form, and fully described in Section B.2 of the Appendix.

1. Add a DEFG optimizer step to verify the in/out/inout option settings.

2. Enhance the DEFG code statement to include a list of variables and buffers used.

3. Optimize DEFG to release over-allocated CPU memory.

4. Add DEFG interchange statement functionality to the execute statement.

[1] Providing Java support has the potential to enable DEFG use with the mobile Android Platform.

5. Improve DEFG's ability to collect run-time statistics.

6. Consider use of *dynamic*, instead of *static*, optimization in DEFG buffer transfer operations.

Even without these improvements, DEFG has shown itself to be a capable and efficient tool for creating OpenCL-based GPU applications.

BIBLIOGRAPHY

[1] Advanced Micro Devices, Inc. Accelerated Parallel Processing (APP) SDK. Website, 2013. http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/.

[2] Advanced Micro Devices, Inc. Accelerated Parallel Processing Math Libraries (APPML). Website, 2013. http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-math-libraries/.

[3] Advanced Micro Devices, Inc. AMD Radeon HD 7990 Graphics. Website, 2013. http://www.amd.com/us/products/desktop/graphics/7000/7990/Pages/radeon-7990.aspx.

[4] Advanced Micro Devices, Inc. clMath (formerly APPML). Website, 2014. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-math-libraries/.

[5] R. Aggarwal and K. Jacques. The impact of fdicia and prompt corrective action on bank capital and risk: Estimates using a simultaneous equations model. *Journal of Banking & Finance*, 25(6):1139–1160, 2001.

[6] Altera, Inc. Altera webpage on OpenCL. Website, 2013. http://www.altera.com/products/software/opencl/opencl-index.html.

[7] M. Altman. An optimum cubically convergent iterative method of inverting a linear bounded operator in hilbert space. *Pacific Journal of Mathematics*, 10(4):1107–1113, 1960.

[8] T. Altman. A method of inexact steepest descent for systems of linear equations. *Computers and Mathematics with Applications*, 19(12):65 – 69, 1990.

[9] T. Altman and B. Chlebus. Sorting roughly sorted sequences in parallel. *Information processing letters*, 33(6):297–300, 1990.

[10] T. Altman and Y. Igarashi. Roughly sorting: Sequential and parallel approach. *Journal of Information Processing*, 12(2):154–158, 1989.

[11] ANTLR3. Antlr3. Website, 2013. http://www.antlr3.org/.

[12] D. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society.

[13] R. Barr, T. Pilkington, J. Boineau, and M. Spach. Determining surface potentials from current dipoles, with application to electrocardiography. *Biomedical Engineering, IEEE Transactions on*, (2):88–92, 1966.

[14] K. Berman and J. Paul. *Fundamentals of Sequential and Parallel Algorithms*. PWS Publishing Co., Boston, MA, USA, 1st edition, 1996.

[15] G. Blelloch. Prefix sums and their applications. *A Carnegie Mellon University Research Showcase Report*, 1990.

[16] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4):175–308, 2006.

[17] M. Boddy and T. Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–285, 1994.

[18] B. Brejová. Analyzing variants of shellsort. *Information Processing Letters*, 79(5):223–227, 2001.

[19] V. Castro and D. Wood. An adaptive generic sorting algorithm that uses variable partitioning. *International journal of computer mathematics*, 61(3-4):181–194, 1996.

[20] Center for Discrete Mathematics and Theoretical Computer Science. Dimacs. Website, 2010. `http://www.dis.uniroma1.it/challenge9/download.shtml`.

[21] P. Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, volume 73. Springer, 2011.

[22] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to algorithms (the third edition), 2009.

[23] R. Couturier. *Designing Scientific Applications on GPUs*. CRC Press, 2013.

[24] T. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[25] T. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. Website, 2014. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[26] F. Dehne and K. Yogaratnam. Exploring the limits of gpus with parallel graph algorithms. *arXiv preprint arXiv:1002.4482*, 2010.

[27] M. Dinneen, M. Khosravani, and A. Probert. Using opencl for implementing simple parallel graph algorithms. In *Proceedings of the 17th annual conference on Parallel and Distributed Processing Techniques and Applications (PDPTA11), part of WORLDCOMP*, volume 11, pages 1–6, 2011.

[28] J. Edd, S. Payen, B. Rubinsky, M. Stoller, and M. Sitti. Biomimetic propulsion for a swimming surgical micro-robot. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2583–2588. IEEE, 2003.

[29] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)*, 24(4):441–476, 1992.

[30] R. Farber. *CUDA Application Design and Development*. Elsevier Science, Burlington, 2011.

[31] W. Feng, H. Lin, T. Scogland, and J. Zhang. Opencl and the 13 dwarfs: a work in progress. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, pages 291–294. ACM, 2012.

[32] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[34] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1*. Morgan Kaufmann, 2012.

[35] N. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 611–622, New York, NY, USA, 2005. ACM.

[36] E. Hansen and S. Zilberstein. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, 126(1):139–157, 2001.

[37] Hardkernel Co., Ltd. ORDOID Platforms. Website, 2013. `http:http://www.hardkernel.com/main/products/prdt_info.php?g_code=G138745696275`.

[38] P. Harish and P. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th international conference on High performance computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[39] M. Harris and M. Garland. Optimizing parallel prefix operations for the fermi architecture. *GPU Computing Gems Jade Edition*, page 29, 2011.

[40] M. Harris, S. Sengupta, and J. Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.

[41] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[42] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[43] T. Huang, G. Yang, and G. Tang. A fast two-dimensional median filtering algorithm. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 27(1):13–18, 1979.

[44] K. Ikuta, H. Ishii, and M. Nokata. Safety evaluation method of design and control for human-care robots. *The International Journal of Robotics Research*, 22(5):281–297, 2003.

[45] T. K. G. Inc. OpenCL Reference Pages. Website, 2010. `http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/`.

[46] M. Ishii, S. Sakane, M. Kakikura, and Y. Mikami. A 3-d sensor system for teaching robot paths and environments. *The International journal of robotics research*, 6(2):45–59, 1987.

[47] L. Jordan and G. Alaghband. *Fundamentals of parallel processing*. Prentice Hall Professional Technical Reference, 2002.

[48] D. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[49] A. Klckner. PyOpenCL. Website, 2010. `http://mathema.tician.de/software/pyopencl`.

[50] A. Klckner. PyCUDA. Website, 2012. `http://mathema.tician.de/software/pycuda`.

[51] D. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[52] A. Krishnamoorthy and D. Menon. Matrix inversion using cholesky decomposition. *arXiv preprint arXiv:1111.4144*, 2011.

[53] C. Lejdfors. PyGPU - Python for the GPU. Website, 2007. `http://fileadmin.cs.lth.se/cs/Personal/Calle_Lejdfors/pygpu/`.

[54] litecoin.info. Mining hardware comparison. Website, 2014. https://litecoin.info/Mining_hardware_comparison.

[55] L. Luo, M. Wong, and W. Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55. ACM, 2010.

[56] C. Ma, L. Yang, W. Gao, and Z. Liu. An improved sobel algorithm based on median filter. In *Mechanical and Electronics Engineering (ICMEE), 2010 2nd International Conference on*, volume 1, pages V1–88. IEEE, 2010.

[57] Mathworks. Parallel Computing Toolbox. Website, 2013. http://www.mathworks.com/products/parallel-computing/.

[58] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[59] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 117–128. ACM, 2012.

[60] D. Merrill and A. Grimshaw. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 545–546. ACM, 2010.

[61] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg. *OpenCL programming guide*. Addison-Wesley Professional, 2011.

[62] M. Nagao and S. Mori. A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of japanese. In *Proceedings of the 15th conference on Computational linguistics-Volume 1*, pages 611–615. Association for Computational Linguistics, 1994.

[63] Y. Nagasaka, K.and Kuroki, S. Suzuki, Y. Itoh, and J. Yamaguchi. Integrated motion control for walking, jumping and running on a small bipedal entertainment robot. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 4, pages 3189–3194. IEEE, 2004.

[64] NVIDIA Corporation. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, NVIDIA Corporation, 2011.

[65] NVIDIA Corporation. *CUDA C Programming Guide*, 4.2 edition, 2012.

[66] NVIDIA Corporation. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110i. Technical report, NVIDIA Corporation, 2012.

[67] NVIDIA Corporation. Cuda zone – opencl. Website, 2013. https://developer.nvidia.com/opencl.

[68] NVIDIA Corporation. Nvidia gpudirect. Website, 2014. https://developer.nvidia.com/gpudirect.

[69] NVIDIA Corportation. NVIDA webpage on GPGPU. Website, 2013. http://www.nvidia.com/object/what-is-gpu-computing.html.

[70] K. Opencl and A. M. The opencl specification version: 1.2 document revision: 15, 2012.

[71] J. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[72] G. Petrie and T. Kennie. Terrain modeling in surveying and civil engineering. *Computer-aided design*, 19(4):171–187, 1987.

[73] W. Petryshyn. On the inversion of matrices and linear operators. *Proceedings of the American Mathematical Society*, 16(5):893–901, 1965.

[74] R. Porra. The chequered history of the development and use of simultaneous equations for the accurate determination of chlorophylls a and b. *Photosynthesis Research*, 73(1-3):149–156, 2002.

[75] Rosenzweig, M. How to Construct an Invertible Matrix? Just Choose Large Diagonals. Website, 2013. http://matthewhr.wordpress.com/2013/09/01/how-to-construct-an-invertible-matrix-just-choose-large-diagonals/.

[76] L. S. and R. Box. A Fast Easy Sort. Website, 1991. http://cs.clackamas.cc.or.us/molatore/cs260Spr03/combsort.htm.

[77] A. Saba and R. Mangharam. Anytime algorithms for gpu architectures. *AVICPS 2010*, page 31, 2010.

[78] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[79] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009.*, pages 1–10. IEEE, 2009.

[80] R. Senser and T. Altman. DEF-G: Declarative Framework for GPU Environment. In *Proceedings of the 19th annual conference on Parallel and Distributed Processing Techniques and Applications (PDPTA13), part of WORLDCOMP*, volume II, pages 490–496, 2013.

[81] R. Senser and T. Altman. A second generation of DEFG: Declarative Framework for GPUs. In *Proceedings of the 20th annual conference on Parallel and Distributed Processing Techniques and Applications (PDPTA14), part of WORLDCOMP*, volume t.b.d., page t.b.d, 2014. To be available November 2014.

[82] R. Senser and T. Altman. Poster: DEFG, Declarative Framework for GPUs - ID P4194. NVIDIA GPU Technology Conference, GTC 2014, March 24-27, 2014, San Jose, CA, 2014. http://on-demand-gtc.gputechconf.com/gtcnew/on-demand-gtc.php?searchByKeyword=senser&searchItems=&sessionTopic=&sessionEvent=2&sessionYear=2014&sessionFormat=5&submit=&select=+\#sthash.v9BEY0N0.dpuf.

[83] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of superscalar processors, beta edition*. McGraw-Hill Science/Engineering/Math, 2003.

[84] M. Sid-Ahmed. *Image processing*. McGraw-Hill, 1994.

[85] K. Skadron. Rodinia: Accelerating compute-intensive applications with accelerators. Website, 2013. http://lava.cs.virginia.edu/Rodinia/.

[86] I. Sobel and G. Feldman. A 3x3 isotropic gradient operator for image processing. *a 1968 talk at the Stanford Artificial Project*, 1968.

[87] M. Sonka, V. Hlavac, and R. Boyle. Image processing, analysis, and machine vision. 1999. *Champion & Hall*, pages 2–6, 1998.

[88] Stanford SNAP Group. Stanford network analysis project. Website, 2014. http://snap.stanford.edu/.

[89] P. Stanimirovic. Self–correcting iterative methods for computing {2}–inverses. *Arch Math (Brno)*, 39:27–36, 2003.

[90] R. Tewarson. A direct method for generalized matrix inversion. *SIAM Journal on Numerical Analysis*, 4(4):499–507, 1967.

[91] TINYXML2. Tinyxml2. Website, 2013. http://www.grinninglizard.com/tinyxml2/index.html.

[92] N. Tuck. Bacon: A gpu programming language with just in time specialization (draft). *University of Massachusetts Lowel, Lowel MA 01854*, 2012.

[93] O. Vincent and O. Folorunso. A descriptive algorithm for sobel image edge detection. In *Proceedings of Informing Science and IT Education Conference (InSITE)*, pages 97–107, 2009.

[94] W. Wang. Reach on sobel operator for vehicle recognition. In *Artificial Intelligence, 2009. JCAI'09. International Joint Conference on*, pages 448–451. IEEE, 2009.

[95] Z. Wei and J. JaJa. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *IPDPS*, pages 1–8. IEEE, 2010.

[96] S. Wesolkowski, M. Jernigan, and R. Dony. Comparison of color image edge detectors in multiple color spaces. In *Image Processing, 2000. Proceedings. 2000 International Conference on*, volume 2, pages 796–799. IEEE, 2000.

[97] Wikipedia. Comb sort. Website, 2014. http://en.wikipedia.org/wiki/Comb_sort.

[98] Wikipedia. Embarrassingly parallel. Website, 2014. http://en.wikipedia.org/wiki/Embarrassingly_parallel.

[99] Wikipedia. Hilbert matrix. Website, 2014. http://en.wikipedia.org/wiki/Hilbert_matrix.

[100] J. Xiong, J. Johnson, R. Johnson, and D. Padua. Spl: A language and compiler for dsp algorithms. In *ACM SIGPLAN Notices*, volume 36, pages 298–308. ACM, 2001.

[101] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.

APPENDIX A

DEFG User's Guide

A.1 Introduction

DEFG is a Domain Specific Language (DSL) that uses declarative statements to provide much of the information needed to generate the CPU-portion of OpenCL GPU applications. It is not a general-purpose programming language, nor is it intended to be. DEFG gives up some of the power that a universal language, such as C++ or Java, provides. In exchange for decreased power, DEFG is able to perform certain GPU-related activities efficiently with minimal software developer effort. This efficiency relates both to the degree of effort required by the developer to create an OpenCL application, and the performance achieved by the generated OpenCL application.

While not a general-purpose language in itself, DEFG does have the ability, with its `code` statement, to insert arbitrary C/C++ code into the applications it generates. These inserted C/C++ code snippets are referred to as DEFG *morsels*, which are intended to allow the developer to insert small snippets to do minor calculations, to format display output, and to assist in debugging.

This User's Guide provides DEFG source code examples and summary descriptions of the common DEFG design patterns. These examples and design pattern descriptions are followed by the detailed DEFG Language Reference Section. Also covered in this DEFG are advanced features and error handling.

A.2 Intended Audience

The intended audience for this User's Guide is both experienced developers who are starting to use GPUs, and seasoned GPU developers. It is assumed that the reader has a basic understanding of GPU technology, the OpenCL Specification, and GPU kernel programming.

A.3 DEFG Examples

Two DEFG example programs are presented here. They provide a glimpse of how the DEFG language is utilized, and they show the CPU-side operations needed for GPU implementation of two common applications. The first example provides for the execution of a single GPU kernel, which is an image filter. The second one shows more of the DEFG power in the form of the looping operation used in a GPU-based Breadth-First Search (BFS) implementation.

A.3.1 Sobel Image Filter Example

This is a simple DEFG application. It loads an image, copies it to the GPU, processes the image with a GPU kernel named sobel_filter, and then brings the image back to the CPU for display.

```
// SobelRef.txt: Sobel algorithm in DEFG syntax
  declare application  sobel
  declare integer Xdim (0)
          integer Ydim (0)
          integer BUF_SIZE (0)
  declare gpu gpuone ( * )
  declare kernel  sobel_filter SobelFilter_Kernels
                  ( [[ 2D,Xdim,Ydim ]] )
  declare integer buffer image1 ( Xdim Ydim )
          integer buffer image2 ( Xdim Ydim )
  call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
  execute run1 sobel_filter ( image1(in) image2(out) )
  call disp_output (image2(in) Xdim (in) Ydim (in) )
 end
```

The DEFG declare statements define the characteristics of the DEFG application. In this case the application name is sobel; it has three integer scalar variables; it uses any available GPU and the OpenCL kernel name is sobel_filter, from the OS file named "SobelFilter_Kernels.cl". There are two GPU integer buffers: image1 and image2. The C++ function init_input is called to load the image, and the C++ function disp_output is called to display the resulting image. Between these two C++ functions, the DEFG execute statement schedules the execution of the GPU kernel and arranges for the movement of the two buffers. The (in) and (out) denotations allow DEFG to optimize the movement of the integer buffers between the CPU and GPU.

A.3.2 Breadth-First Search Example

The second DEFG example shows multiple kernel execution and the looping capabilities of DEFG. Here, a DEFG looping construct is used to repeat the set of kernels, and DEFG internally optimizes the loop's data buffer movements between the CPU and GPU devices.

```
// bfsRef.txt: BFS (Harish version)
  declare application  bfs
  declare integer NODE_CNT (0)
          integer EDGE_CNT (0)
          integer MAX_DEGREE (0)
          integer STOP (0)
  declare gpu gpuone ( * )
  declare kernel kernel1 bfs_kernel ( [[ 1D,NODE_CNT ]] )
          kernel kernel2 bfs_kernel ( [[ 1D,NODE_CNT ]] )
  declare struct (8) buffer graph_nodes ( NODE_CNT )
          integer buffer graph_edges (EDGE_CNT )
          integer buffer graph_mask ( NODE_CNT )
          integer buffer updating_graph_mask ( NODE_CNT )
          integer buffer graph_visited (NODE_CNT )
          integer buffer cost  (NODE_CNT)
  call init_input (graph_nodes(out)
                   graph_edges(out)
                   graph_mask(out)
```

```
                    updating_graph_mask(out)
                    graph_visited (out)
                    cost (out)
                    NODE_CNT(out)
                    EDGE_CNT(out)
                    MAX_DEGREE(out))
  loop
    execute part1 kernel1 ( graph_nodes(in)
                            graph_edges(in)
                            graph_mask(in)
                            updating_graph_mask(inout)
                            graph_visited(in)
                            cost(inout)
                            NODE_CNT(in)
                            )
    set STOP (0)
    execute part2 kernel2 ( graph_mask(inout)
                            updating_graph_mask(inout)
                            graph_visited(inout)
                            STOP(inout)
                            NODE_CNT(in)
                            )
  while STOP eq 1
  call disp_output (cost(in) NODE_CNT(in))
 end
```

Here, the DEFG declare, call, and execute statements function as was described in the first example. This example introduces the loop, set, and while DEFG statements to manage the required looping behavior. The loop statement denotes the beginning of a processing loop. The while statement denotes the end of the processing loop and provides the loop-exit condition. The set statement assigns the given value to the scalar variable, STOP. In this example, the value previously set in STOP is changeable by the second kernel executed and thus forms the loop exit control, which is managed by the second kernel.

Although this appears to be just a procedural programming loop, the DEFG optimizer processing uses very carefully the characteristics of this loop to generate the OpenCL code to move data buffers. For instance, the *graph_edges* buffer, which is used by the first kernel within the loop, is only moved once from the CPU to the

GPU. If the (in), (inout), and (out) denotations were coded differently, DEFG could generate code to move this buffer in each iteration of the loop.

A.4 Common DEFG Design Patterns

DEFG relies heavily on a set of templates to generate the needed OpenCL code. When these templates are combined with certain DEFG coding techniques, the results are the *DEFG design patterns*. This section describes the most commonly used patterns. Some of them are engaged via the use of certain DEFG statements or keywords. Other design patterns are invoked by using the common DEFG statements in specific groupings. By their very nature, these design patterns can be overlapped, and in certain cases, limited by the presence of other patterns. Most of the complexity in using these design pattern arises when using the DEFG multiple-GPU card support. Use of single-GPU DEFG design patterns tends to be straight-forward.

A.4.1 Execution-Flow Design Patterns

DEFG provides the three execution-flow design patterns, described in this section. These patterns provide the basic processing template that the DEFG code generator uses to form the C/C++ code.

A.4.1.1 Sequential-Flow Design Pattern

The Sequential-Flow design pattern is always present. It causes the statements following the DEFG declare statement to be executed from top to bottom. The Single-Kernel Repeat Sequence and Multiple-Kernel Loop Design Patterns, discussed below, can substantially alter this top-to-bottom ordering.

A.4.1.2 Single-Kernel Repeat Sequence Design Pattern

The Single-Kernel Repeat Sequence is a looping design pattern that allows a single DEFG execute or multi_exec statement to be executed a fixed number of times. The DEFG statement to utilize this pattern is the sequence. It can be used with both the Sequential-Flow and Multiple-Kernel Design Patterns.

A.4.1.3 Multiple-Kernel Loop Design Pattern

The Multiple-Kernel Loop design pattern is a conditional looping pattern. It is used within the Sequential-Flow design pattern. The DEFG statements needed to form this design pattern are the loop and while statements. The Multiple-Kernel Loop design pattern may not be embedded within itself. In other words, a DEFG loop/while construct may not contain another loop/while construct. The Single-Kernel Repeat Sequence design pattern may be used within the Multiple-Kernel Loop design pattern.

A.4.2 Anytime Processing Design Pattern

The Anytime Processing design pattern is used with the Multiple-Kernel Loop design pattern. This pattern is used to cause a premature exit of the design pattern's loop. The criteria to exit the loop can be run-time based, through the use of the DEFG anytime statement, or data-comparison based, through the use of the DEFG code morsel.

A.4.3 Application versus Module Design Patterns

Standalone applications, with a C/C++ **main()** entry function, and callable functions can both be created with DEFG.

A.4.3.1 Application Design Pattern

The Application design pattern generates a standalone application and is indicated by using a DEFG declare application statement.

A.4.3.2 Module Design Pattern

The Module design pattern generates a callable C/C++ function and is indicated by using the DEFG declare module statement, sometimes followed by one or more optional DEFG declare parameter statements.

A.4.4 BLAS Design Pattern

The clMath Basic Linear Algebra Subprograms (BLAS) library is provided for OpenCL [2]. The clMath library was formerly known as APPML. This set of subprograms provides a set of OpenCL-callable BLAS capabilities. DEFG makes use of the double-precision matrix multiply API, *clAmdBlasDgemm()*, from this library with its blas statement.[1] The main difference between execute and blas is that the execute statement executes a specific OpenCL kernel, and the blas statement executes the desired subprogram from the library which then calls its own set of OpenCL kernels.

A.4.5 Virtual-Pointer/Prefix Sum Design Pattern

DEFG is supplied with several OpenCL kernels that perform parallel prefix sum. We have found that prefix sum is a common operation performed in DEFG applications. These kernels can be used by any OpenCL application that needs prefix-sum processing. They are specifically included in DEFG to facilitate the developer-use of prefix sum in assigning offsets in a shared buffer, without the use of low-level synchronization. These kernels are named: *bermanPrefixSumP1*, *bermanPrefixSumP2b*,

[1] Support for additional clMath functions may be added to DEFG at a later date.

and *getCellValue*. They are used in the Breadth-First Search Application, discussed in Section 5.2.

A.4.6 Multiple GPU Support Design Patterns

When the design of the GPU algorithms used in a DEFG program permit the concurrent use of more than one GPU device, DEFG supports design patterns to enable the use of multiple GPU devices.

A.4.6.1 Multiple-Execution Design Pattern

With certain applications, it is desirable to split the workload over more than a single GPU device. This may be done to obtain faster performance, handle larger problems, or both. DEFG makes this possible with the use of the multi_exec statement. With this capability, DEFG will execute the workload over the group of GPU devices declared in the declare gpu statement. The basic algorithms in use must be intended for this high degree of parallelism, and the OpenCL kernels in use must also have been designed for this mode of use. This pattern is used in the image filters applications, discussed in Section 5.1. The design pattern described next is often used with this Multiple-Execution design pattern.

A.4.6.2 Managed Buffer Design Pattern

When the just-described Multiple-Execution design pattern is used, it is often necessary to segment the data passed to each GPU. DEFG provides a number of declare buffer options to make it possible for DEFG to automatically segment the data. These options are halo, multi, and nonpartable. In the single-GPU context, these options have no effect; they are ignored. When the multi_exec is used, DEFG buffers are considered *partable*, unless a buffer option is supplied. The default *partable* behavior is to split the data into equal segments for passing to the selected GPUs. When the nonpartable

option is used, the buffer is not segmented and is given, in full, to each GPU. The halo option is often used with images, because it notifies DEFG that the data contains small edges. The result of using halo is that edge data is sent to each GPU using the edge. The multi option is used with the DEFG_GLOB variable to handle complex segmentations of the input data. With the DEFG_GLOB approach, the C/C++ function that brings the data into the DEFG-generated program controls the splitting of the data. It also notifies DEFG-generated program, via DEFG_GLOB, how the data is allocated to the GPU buffers. This option is often needed when graphical data needs to be processed over more than a single GPU. Use of DEFG_GLOB is shown in the Breadth-First Search application, discussed in Section 5.2.

A.5 DEFG Language Reference

A.5.1 DEFG Statements

This section lists the DEFG statements in alphabetical order. Each DEFG statement is described, followed by a syntax description and then a very small code example. The "<" and ">" meta characters, used in the syntax descriptions, are used to denote names, variables, constants, and literals; the single "[" and '] " meta characters are used to denote optional repeating sequences; and the "|" meta character is used to denote options. Note that the double "[[" and "]]" character sequences are literal characters and are part of the actual DEFG syntax.

1. blas Statement

   DEFG is able to utilize the **clAmdBlasDgemm()** BLAS function to perform double-precision matrix multiplication. To utilize the DEFG blas statement, the AMD clMath (formerly APPML) library must have already been installed. The DEFG blas statement is a wrapper around the **clAmdBlasDgemm()** C/C++ function. Note that this statement requires that all fields be provided. In most

cases, the scalar variables must be to be set to 0 or 1 to get the desired results. Note that the "->" character sequence is part of the actual statement syntax. In the example below, the linear algebra calculation performed is:

$$matrix3 = scalar1 * matrix1 * matrix2 + scalar2 * matrix3$$

Syntax:

```
blas (<scalar1> * <matrix1> * <matrix2> + <scalar2> * <matrix3> ->
                    <matrix3>)
```

Example:

```
set dZero (0.0)
set dOne (1.0)
blas (dOne * mP * mP + dZero * mW -> mW)
```

2. broadcast Statement

When a DEFG program is being used in multiple-GPU mode, the broadcast statement is used to make the contents of a data buffer on a given GPU available to the other GPUs. The statement's GPU number must be a constant and preceded by the "@" character. In the current DEFG implementation, this statement forces the data buffer to be copied back to the CPU for later usage by a requesting GPU. This operation is *not* done as a GPU to GPU copy and, therefore, tends to be relatively slow.

Syntax:

```
broadcast (<variable> @<gpu>)
```

Example:

```
broadcast (frontier0 @0)
```

3. call Statement

It is very common for a DEFG program to rely on C/C++ functions to bring

data into the DEFG program and to forward the GPU-processed data to the CPU for further processing or output. The call statement is used to execute C/C++ functions. The name of the called C/C++ function, and all referenced DEFG variable names, must follow C/C++ naming conventions and not begin with either defg_ or DEFG_. The required options in, inout, and out are described in this manual's DEFG Statement Options Section. These three options provide the DEFG Translator with the data movement directions.

Syntax:

call <function_name> (<variable>(<direction>) [<variable>(<direction>)])

where: <function_name> is a C/C++ function name

and: <variable> is a DEFG scalar or buffer name

and: <direction> is a direction

Example:

```
call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
```

The DEFG Generated C/C++ code for this call statement would be:

```
init_input(image1, Xdim, Ydim, BUF_SIZE);
```

Whereas, the corresponding declaration in actual C/C++ function could be:

```
void init_input (void * image1, int& Xdim, int& Ydim, int& BUF_SIZE);
```

Note: (1) the function is of type void, and (2) the required use of the C++ call-by-reference operator, that is, the & character, on all the scalar values passed between the DEFG-generated code and the C/C++ function. Since the data buffers are passed as void pointers, they are also passed by reference.

4. code Statement

The DEFG code statement may be used to insert C/C++ code into the gener-

ated C/C++ program. These inserted code snippets are called *morsels*. The morsel's code snippet, provided between the "[[" and "]]" markers, is inserted into the generated program without being parsed or verified. Common uses for this statement are outputting scalar values (for debugging) and generating input data (for testing). Note that when outputting GPU results, the GPU's data must be current and valid on the CPU. References to any DEFG fields, from embedded C/C++ code, are not managed by the DEFG optimizer. Therefore, the DEFG release statement can be used to force a copy of the GPU data back to the CPU. Obviously, the DEFG code statement can create hard-to-find software errors; it should only be used with great care.

Syntax:

code [[<native C/C++ code>]]

Example:

```
code [[ printf("KCNT: %d\n", KCNT); ]]
```

5. declare application Statement

Each DEFG application requires a name, which is provided by the declare application. The name is listed in the generated C/C++ code along with a DEFG translation time stamp.

Syntax:

declare application <application_name>

where: <application_name> is the name used in code generation

Example:

```
declare application sobel
```

6. declare buffer Statement

Each data buffer used within a DEFG program must be declared, and all in

a single declare buffer statement. The maximum size of the buffers can be changed with a run-time environment variable, called DEFG_MAX_BUF. The data buffers are treated as C/C++ arrays of type double, float, integer or struct. The DEFG support for struct arrays is limited and these limits are discussed in the DEFG Data Types Section.

Syntax:

declare buffer <buffer_entry> [<buffer_entry>]

where: <buffer_entry> consists of

<data_type> <name> (<size>) [<buffer_option>]

and: <data_type> consists of the data type

and: <name> consists of the buffer name

and: <size> consists of the number of occurrences

and: <buffer_option> consists of halo | local | multi | nonpartable

Example:

```
declare integer buffer data1 ( size )
        integer buffer data2 ( size )
```

The halo, local, multi, and nonpartable options are described in the DEFG Statement Options Section; the DEFG Advanced Features Section provides additional information.

7. declare gpu Statement

The gpu declaration determines which device or devices are selected for use. The devices are normally any GPUs, or the CPU. The GPU selection criteria are defined as follows:

(a) *: Matches any single device, CPU included.

(b) any: Matches any single GPU, CPU excluded.

(c) all: Matches all GPUs.

(d) quoted list: Matches device names listed.

Syntax:

declare gpu <gpu_group_name> (<gpu_criteria>)

where: <gpu_group_name> is the group name

and: <gpu_criteria> consists of * | any | all | < gpu_list>

and: < gpu_list> consists of a sequence of one or more quoted GPU names.

Two examples:

```
declare gpu gpuone ( * )
declare gpu gpuone ( "GeForce GT 220" )
```

8. declare kernel Statement

The names of the kernel or kernels required are declared with the declare kernel statement. Each kernel referenced has an internal name, and in most cases a file name. The internal name must match the actual kernel name and the file name is the name of the text file containing the kernel code. The text file must have an extension of ".cl". Note that there is an option to insert the kernel text into the DEFG code through the use of the insert code [[..]] phrase. This kernel insert code option is intended for debugging purposes; that is, for the temporary execution of small amounts of uncommented kernel code.

Each kernel requires global parameters and may include local sizing parameters. The current DEFG version uses the [[..]] syntax to contain the parameters needed for OpenCL. The first parameter is 1D or 2D, denoting the dimensionality of the data. The next one or two parameters denote the associated dimension size. Note the use of commas, which is not the normal DEFG syntax design, and the optional colon character. The optional local parameters, with one or two values, are preceded by a colon character.

Syntax:

declare kernel <kernel_name> <kernel_file_name> (<kernel_run-time>)

where: <kernel_name> consists of the name of the kernel function

and: <kernel_file_name> consists of the name of the kernel text file

and: <kernel_run_time> consists of the sizing parameters

note that <kernel_file_name> can be replaced by the

insert code [[..]] as shown below.

Three examples:

```
declare kernel mfavg_filter Mfavg_Kernels ( [[ 1D,Dim ]] )
declare kernel local_sample Samples ( [[ 1D,100:10 ]] )
declare kernel tiny_kernel
        insert code [[
        __kernel void tiny_kernel(__global int* p1){p1[0] = 34; return;}
        ]] ( [[ 1D,1 ]] )
```

9. declare module Statement

DEFG has the option to generate C/C++ functions instead of standalone applications with a C/C++ **main** entry point. In order to generate a function, the declare module statement is used. The module name provided becomes the function name of the generated C/C++ code. The arguments to the C/C++ function are provided with the declare parameter statement, discussed in the next section.

Syntax:

declare module <module_name>

where: <module_name> is the name used for the generated function

Example:

```
declare module sobelc
```

10. declare parameter Statement

    When the DEFG declare module statement is used, the arguments to the generated function are defined using the declare parameter statement. Only the parameter name is supplied and the name must reference a declared variable or buffer.

    Syntax:

    declare parameter <parameter_name>

    where: <parameter_name> matches a declared variable or buffer

    Example:

    declare parameter image1

11. declare variable Statement

    Each developer-defined scalar variable used within a DEFG program must be declared. All variables are declared within a single declare statement. These variables are treated as C/C++ variables of type double, float, or integer. Note that the literal "variable" is *not* part of this DEFG statement.

    Syntax:

    declare <variable_entry> [<variable_entry>]

    where: <variable_entry> consists of <data_type> <name>

    Example:

    declare integer num1 data1

    integer data2

12. end_timer Statement

    DEFG automatically provides one timer that the developer can use to compute the run times of DEFG code. See the start_timer statement below for a full description.

13. execute Statement

The DEFG execute statement schedules the execution of the kernel on a single GPU. If the needed input variables and buffers are not already on the GPU, then copies of these are moved to the GPU. The required options in, inout, and out express the direction of data movement and are more fully described in this guide's Statement Options Section. The correct setting of these options is critical, as the DEFG optimizer uses their values to efficiently move the variables and buffers between the devices. The incorrect setting of these options can cause poor performance and/or incorrect results.

Syntax:

execute <run_name> <kernel_name> ([<variable>(<option>)])

where: <run_name> is the unique name of this execution step

and: <kernel_name> consists of the kernel name

and: <variable> consists of a variable name

and: <option> is one of in, inout, or out

Example:

execute run1 sobel_filter (image1(in) image2(out))

14. include Statement

The DEFG include statement may be used to insert C/C++ code into the generated C/C++ program. The code snippet provided, between the '[[' and ']]' markers can define C-style macros, as well as, contain C/C++ #include statements. The DEFG include statement can create hard-to-find software errors; it should only be used after careful consideration as the side effects on its use. This statement differs from the code statement insertions in that the include insertions occur at the very beginning of the generated C/C++ program.

Syntax:

include [[<native C/C++ include and similar statements>]]

Example:

include [[#define INDEX2(xi,xj,xsize,xind) xind = (xi * xsize) + xj;]]

15. interchange Statement

The DEFG interchange statement is used to make DEFG programs easier to understand, faster, and smaller by making it possible to interchange the contents of two GPU buffers. The interchange occurs without copying the contents back to the CPU and actually swapping the buffer contents. This statement is often used when a given kernel is executed more than once and the output of the previous iteration is the input to the next iteration.

Syntax:

interchange (<variable1> <variable2>)

Example:

```
loop
    execute Run2 Kernel (LR(inout) LRout(out) size(in) stride(in) groupSize(in))
    interchange(LR LRout)
while ...
```

16. loop Statement

The DEFG loop statement is used to repeat a sequence of DEFG statements a variable number of times. The loop termination condition is handled by a single variable, checked by the while clause. If more than one condition needs to be checked, then the while clause has to be preceded by a code morsel that processes the multiple conditions and returns the results in one scalar DEFG variable. DEFG loop statements may not be layered or embedded in other loops, but the DEFG loop statement may contain one or more sequence statements. These DEFG domain-specific language limits make it possible for the DEFG

optimizer to efficiently manage the OpenCL buffer transfers within the loop via static optimizations.

Syntax:

loop <DEFG_statements> while <condition>

where: <DEFG_statements> consists of executable statements

and: <condition> consists of

<variable_name> <operator> <numeric_constant>

and: <operator> consists of one of eq ne lt le gt ge.

Example:

```
loop
    //maybe execute some DEFG code
    set again (0)
    //execute some DEFG code that updates again variable
while again eq 1
```

The eq ne lt le gt and ge operators represent $=, \neq, <$, etc.

17. multi_exec Statement

The DEFG multi_exec statement schedules the execution of the kernel on two or more GPUs. The associated declare gpu statement must have provided for at least two GPUs, or the multi_exec statement fails with an error. More information on the use of this statement is available in the execute statement description and in the Multiple GPU Support Section. The required options in, inout, and out are described in this guide's Statement Options Section.

Syntax:

multi_exec <run_name> <kernel_name> ([<variable>(<option>)])

where: <run_name> is the unique name of this execution step

and: <kernel_name> consists of the kernel name

and: <variable> consists of a variable name

and: <option> is one of in, inout, or out

Example:

multi_exec run1 sobel_filter (image1(in) image2(out))

18. output_timer Statement

DEFG automatically provides one timer that the developer can use to compute the run times of DEFG code. See the start_timer statement for a full description.

19. release Statement

In certain instances, the DEFG optimizer needs to be informed that a variable or buffer must contain valid contents on the CPU. The release statement provides this functionality. As an example, a release statement is required for any variable or buffer that is returned to a calling program from a DEFG module; this statement guarantees the data being returned to the caller is valid. This statement may also be used before the end_timer statement to verify that the referenced CPU field has valid, not old, contents.

Syntax:

release (<name>)

where: <name> consists of the name of a variable

Example:

release (image2)

20. sequence Statement

The DEFG sequence statement is a looping construct and must be immediately followed by an associated execute or multi_exec statement. The associated execute or multi_exec statement is re-executed, in a sequence, the number of times

specified. The DEFG_CNT system variable provides an iteration count. This DEFG_CNT system variable is zero-indexed.

Syntax:

sequence <count> times <exec>

where: <count> consists of a variable or constant

and: <exec> consists of an execute or multi_exec statement

Example:

```
sequence NODE_CNT times
    execute run1 FWarshall ( buffer1(inout) buffer2(inout) DEFG_CNT(in))
```

21. set Statement

The DEFG set statement is used to copy the value of a constant to a scalar variable. Note: for scalar-variable-to-scalar-variable copying, a code morsel may be used.

Syntax:

set <name>(<value>)

where: <name> consists of the name of the variable

and: <value> consists of the constant value given to the variable

Example:

```
set STOP (0)
```

22. start_timer Statement

DEFG automatically provides one timer that the developer may use to compute simple run times of DEFG code. Note that to get accurate times, the data being processed or updated likely has to be copied back to the CPU, otherwise the the full time used may not be captured. This may require use of the release statement to force the updated buffers to the CPU.

Three simple statements are used to compute run times: start_timer, end_timer, and output_timer. Here is an example of their use.

Example:

```
start_timer
// do something to update image2
release (image2)
end_timer
// optionally, do something un-timed
output_timer
```

23. while Statement

    The DEFG while statement is always paired with a preceding loop statement. See the loop statement description for the details of the while statement.

A.5.2 DEFG Statement Options

1. halo Option

   When the multi_exec statement is used, DEFG has the ability to share buffer data between the selected GPUs. The halo option is used on the declare buffer statement to provide DEFG with the number of edge cells (for 1D processing) or rows (for 2D processing) that must be sent to both GPUs that are processing the edge. The GPU processing boundaries are at the edges and when halo is used, DEFG manages the needed duplication of data for each GPU. This statement option is most commonly used with image filters where each pixel's processing requires the data values of the neighboring pixels.

2. in/inout/out/* Option

   In this discussion, *field* refers to either a DEFG *buffer* or *variable*. The options

in, inout, out and * are used to inform the DEFG optimizer how a given field on a DEFG call, execute, or multi_exec statement is used. The field can be used for input, output, or both. The * can be used with the call statement to mark a given field as *don't care*, which means the DEFG optimizer does not need to move any data for this field. The in marks a field as *input*; the out marks a field as *output*; and inout marks a field as both *input* and *output*; A given in, inout or out is appropriate for only the DEFG statement it appears with. For example, an in associated with a given field on a call statement means that field is used for input by this called function. Likewise, an in associated with a given field on an execute statement means that field is used for input by this GPU kernel.

A note of caution: if these options are not set correctly, then a given DEFG program may perform poorly, due to unnecessary data moves between the devices. Worse, erroneous results may produced due to the necessary data moves not being performed. The correct results may be contained in the GPU's memory, but if these options are not set correctly, then these results may not be present on the CPU. The DEFG translator does not parse the GPU kernel code to verify passed fields and their option settings.

3. local Option

   Local is a declare buffer statement option that is used to mark a buffer as local to the GPU. Buffers of this type are not transferred between the CPU and GPU, and they are usually restricted in size. Local buffers are normally processed by the kernels more quickly than buffers kept in GPU global storage. Note that each GPU work-group has private local storage.

4. multi Option

   When the multi_exec statement is used, a sharing of the workload between GPUs is implied. In cases where the other declare buffer statement options do not

provide the required data segmentation, the multi option is available. When this option is used, DEFG relies on the C/C++ program loading the given data buffer to set the DEFG_GLOB variable (actually a C/C++ structure) with the information to determine which area of the given buffer goes to a given GPU. This option is complex and not easy to use. As the name implies, this variable is global and shared by all multi buffers. It should be used as a solution of last resort.

5. nonpartable Option

   The nonpartable option makes it possible to mark a buffer as non-segmented when the multi_exec statement is used. A buffer declared with the nonpartable option is passed, in full, to each GPU. This option is very useful when the same read-only data must be passed to each GPU participating in a multi_exec step.

6. [[. . .]] Option

   The [[. . .]] construct is used to pass the exact characters between the "[[" and "]]" delimiters to the DEFG run-time code. In the case of C/C++ code, the code is processed by the CPU C/C++ compiler or the OpenCL driver, as appropriate. In the case of global and optional local sizing parameters in the DEFG declare gpu statement, these parameters are passed to the OpenCL **clEnqueueNDRangeKernel()** function. Note that these NDRange values may be adjusted automatically when multiple-GPU support is active.

A.5.3 DEFG Data Types

This section discusses data type support in DEFG. First, note that the common C/C++ char and string data types are not directly supported by DEFG. The DEFG code morsels make it possible to deal with char and string data types, from within DEFG, but the majority of the DEFG statements do not support them. The remain-

der of this section describes the data types supported by DEFG.

A.5.3.1 Data Type: double

The DEFG double data type maps directly to the C/C++ double data type provided by the C/C++ compiler.

A.5.3.2 Data Type: float

The DEFG float data type maps directly to the C/C++ float data type provided by the C/C++ compiler.

A.5.3.3 Data Type: integer

The DEFG integer data type maps directly to the C/C++ int data type provided by the C/C++ compiler.

A.5.3.4 Data Type: struct

The DEFG direct support of C/C++ structures is limited to copying the contents of the struct variables through DEFG. When accessing the actual fields within a struct, a DEFG code statement must be used.

A.5.4 DEFG System Variables

DEFG provides a number of internal system variables. These variables make it possible for the DEFG developer to access DEFG internal information.

A.5.4.1 DEFG_CNT Variable

DEFG_CNT is a read-only DEFG system variable that provides the current loop count when a DEFG sequence is used. It is a zero-indexed counter. When this variable is read outside of a sequence loop, it has a value of zero.

A.5.4.2 DEFG_GLOB Structure

The DEFG_GLOB variable is a C/C++ structure that is internal to DEFG. It is used to manage the partitioning of data buffers when the multi option is used on a buffer. The values in DEFG_GLOB are normally set by the C/C++ function that populates the data buffers. This structure must be passed as an argument to any such C/C++ function. See the DEFG Advanced Features Section for more details.

A.5.4.3 DEFG_GPU Variable

The read-only DEFG system variable DEFG_GPU provides the ordinal of the current GPU. This variable is intended to be used with the multi_exec statement, when it is necessary to know which of the selected GPUs is active. This variable is zero indexed.

A.5.4.4 DEFG_GPU_COUNT Variable

The read-only DEFG system variable DEFG_GPU_COUNT provides the maximum number of GPUs active for this program. The number of GPUs active is determined by the declare gpu statement and the actual hardware configuration.

A.5.5 DEFG Environment Variables

These DEFG environment variables can be accessed via the operating system. They can be used to influence the behavior of the DEFG Translator compilation and run-time behavior of the generated DEFG program.

A.5.5.1 DEFG_MAX_BUF Environment Variable

The DEFG_MAX_BUF environment variable is used at run time to set the size of CPU buffers used to hold DEFG data buffers. The default value is 4MB. Larger values can be used, depending on the available RAM on the CPU and GPU.

A.5.5.2 DEFG_TIMERS Environment Variable

The DEFG_TIMERS environment variable is used at *compile* time. This is not a run-time setting. When this environment variable is set to a value of "1," many of the low-level OpenCL calls for buffer movement and kernel execution are timed and displayed. This feature can be helpful for DEFG debugging and tuning; it should only be used for debugging and testing, as it impacts the overall performance and stability of DEFG.

A.5.6 DEFG Utilities and Functions

DEFG provides additional added features to assist the developer.

A.5.6.1 rsDevices Utility

rsDevices is a small utility program that can be used to list the available GPUs on a given CPU. The output lists the OpenCL platforms available and then each OpenCL-supported device. Each device name is followed by "|CPU" or "|GPU", depending on its device type. These device names can be used with the DEFG declare gpu statement's quoted list of GPU names.

Here is the output from a sample Linux execution:

```
$ ./rsDevices
platform: Advanced Micro Devices, Inc.
platform: NVIDIA Corporation
Six-Core AMD Opteron(tm) Processor 2427|(CPU)
Tesla S2050|(GPU)
Tesla S2050|(GPU)
$
```

A.5.6.2 Loader Functions

DEFG provides a number of C/C++ functions to input data files, output results, and perform simple calculations. These functions are supplied in the "defg_loaders.h" include file. Additional loader functions can be added to this header file or additional header files listed in DEFG. These new headers can be referenced with a DEFG include statement. A partial list of the commonly used "defg_loaders.h" functions is given in Table A.1.[2]

Facility	Description	Function Name
ImageLoader	From the AMD SDK	init_input()
Floyd Warshall Graph Loader	From the AMD SDK	init_input()
BFS Graph Loader	From Rodinia Benchmark	init_input()
Array Partition	Partition for 2-GPU use	ArrayPartition2GPU()
Array Merge	Merge for 2-GPU use	MergeCost2GPU()
Dump Scalar	Output a scalar variable	dumpScalar()
Dump Buffer	Output a buffer	dumpBuffer()
Debug Exit	Immediate processing stop	debugExit
Increment	Increment a scalar	inc()
Decrement	Decrement a scalar	dec()

Table A.1: A Partial List of DEFG Loaders and Functions

A developer should review the "defg_loaders.h" include file to see exactly which functions are available and the types of data processed. New application-specific functions can be added to this header file.

A.6 DEFG Advanced Features

A.6.1 Direct Insertion of C/C++ Code

The DEFG optimizer uses the limited domain of the DEFG language to assist in the generation of efficient OpenCL code. Specifically, this domain limit makes it possible for the optimizer to manage the scalar variables and data buffers in ways

[2]Duplicate function names are managed through the use of application name C-style macros; see the actual header file for the names and details.

that avoid unneeded OpenCL data movement operations between the CPU and GPU. This optimizer does not process the actions performed in the DEFG code statements, referred to as DEFG *morsels*. Therefore, morsels that modify data, or use assumptions as to the current CPU or GPU location of modified data, should be written with great care. The release statement can be used to notify DEFG that a given buffer needs to be made current on the CPU. The DEFG optimizer may move, or not move, data between the CPU and GPU at expected times; it may have pre-staged the data movement or postponed the data movement, depending on factors unique to the optimizer.

A.6.2 Multiple GPU Support

When an application is written in DEFG, the application is limited by the capabilities of the resources available. A single-GPU DEFG application is limited to the resources of one GPU. If the data being processed is even one byte larger than the available GPU memory, the application likely will not execute. Therefore, DEFG supplies the capability to utilize more than a single GPU. For applications that permit a higher degree of parallelism, this capability makes it possible to obtain faster performance, handle larger problems, or both. When this capability is engaged with the multi_exec statement, the processing is spread over the GPUs selected by the declare gpu statement. It cannot be over emphasized that this multiple GPU support capability is limited by the algorithms in use and the implementation of the OpenCL kernels used. This feature simplifies the CPU coding to support multiple GPUs; it does not automatically turn an arbitrary, single-GPU application into a new one with multiple-GPU support.

A.6.3 Support for Mobile GPU Platforms

DEFG has the potential to be used on mobile platform hardware, when the Linux operating system is used. We do not anticipate DEFG being used with the Android operating system. C/C++ programs generated by DEFG have been built and executed on the ARM Cortex AP processor used with the ORDOID U3 [37]. Unfortunately, the Linux OpenCL driver for the integrated ARM Mali-400 Quad Core 440MHz GPU was not available at the time of our testing.

A.6.4 Use of DEFG_GLOB

Caveat: DEFG_GLOB is a very advanced DEFG feature and should only be used by experienced C/C++ developers who have determined that the other DEFG buffer options will not provide the functionality they need.

When the DEFG multi option is added to a buffer and multi_exec is used, the buffer's layout is then managed by the C/C++ function that populates the buffer. This capability provides a mechanism for the developer to influence what data does to each GPU, with DEFG performing the data transfers to the selected GPUs. The DEFG_GLOB variable is a C/C++ structure that contains offsets and lengths, which are used to manage associated DEFG buffers. The actual structure is declared in "defg_template.txt" and its format is DEFG-version dependent.

A.6.5 Global and Local Range Settings

The DEFG declare kernel statement requires that the OpenCL global range be set. This parameter determines the number of device threads started and how they are to be managed. OpenCL has the capability to set the local range automatically and DEFG routinely uses this capability. In some instances, the developer may wish to set this local range setting directly. In this event, the global range within the "[[" and "]]" phrase is followed by a colon character and then the local range is given. The use

of local storage by a kernel is likely to require the developer to set the corresponding declare kernel statement's local range. See the declare kernel statement description for its full syntax.

A.6.6 A Few DEFG Advanced Techniques

A.6.6.1 Changing the DEFG Run-Time Output Location

DEFG generates informational output and error messages at run time. The generated DEFG code uses the DEFG_PRINTF C-style macro to route output to the **printf()** function. This routing may be changed by updating the DEFG_PRINTF macro content in the "defg_template.txt" file.

A.6.6.2 Changing the DEFG Exit Behavior

At run time, DEFG will sometimes abort the processing due to unrecoverable errors. The generated DEFG code uses the DEFG_EXIT C-style macro to route execution to the **exit()** function. This routing may be changed by updating the DEFG_EXIT macro setting in the generated code.

A.6.6.3 Additional *Anytime* Exit Capabilities

Additional Anytime-like processing can be inserted into a DEFG application with code morsels. This basically involves using a C/C++ **if** statement and **goto** statement to jump out of the DEFG loop.[3] It will be necessary examine at the generated code to determine the label of the C/C++ loop exit. This is a very advanced DEFG feature and is DEFG-version dependent. It should only be used as a solution of last resort.

[3]The use of a **goto** statement is related to the scoping of internal DEFG C/C++ variables.

A.6.6.4 DEFG call statement is preferred over DEFG code morsel

As stated earlier, the code morsel can easily introduce errors. This small section discusses why the call statement is preferred over the code statement.

The DEFG call statement contains the in/out/inout/* settings. These inform the DEFG optimizer how the fields being given to the called function are used. Using this information, the optimizer moves the needed input data to the CPU and moves updated DEFG variables and buffers back to the GPU. The DEFG code morsels lack this capability. DEFG will insert the provided C/C++ code into the generated program at the given location, but DEFG does not monitor the variables and buffers accessed. Data read in the morsel may be invalid and updates done in the morsel may be lost. It is the developer's responsibility to be certain that the data used and updated in the morsel is correctly used.

A.7 How to Execute the DEFG Translator

In order to execute the DEFG translator under Windows, the **translate** batch file is supplied. This batch file will execute the DEFG parser, optimizer, and code generator. The DEFG source code must be in a text file with the extension type of ".txt" and the resulting program will have an extension of type ".cpp". The translate batch file expects one input command-line argument, and that must be the name of the source file to be translated from DEFG to C/C++. The example below translates "sobel.txt" to "sobel.cpp".

Example:

```
translate sobel
```

A.8 DEFG Error Handling

A.8.1 Translator Errors

The DEFG Translator has somewhat limited error reporting capabilities and will only report one translation error at a time. It may take multiple editing sessions and translator executions to find all the errors present. When an error occurs, a corresponding ".cpp" file will not be generated.

Example: Translation with an error on line 21

```
C>translate sobelerr
C>echo off
"******** TRANSLATE ********"
"sobelerr parse"
sobelerr.txt line 21:2 extraneous input 'this_is_an_error' expecting END
C>
```

A.8.2 Run-Time Errors

DEFG translates the DEFG input into a C/C++ program that is then compiled and executed. This C/C++ program can produce run-time errors. When DEFG-generated code senses a run-time error condition, its default behavior is to describe the error with a call to the C/C++ *printf()* function. This behavior can be adjusted via the DEFG_PRINTF C-style macro defined in "defg_template.txt". Here is the display of a run-time error showing an OpenCL failure:

Example from Windows:

```
C>sobel
NOTICE: run 1 CPU
clCreateKernel (332) status: -44
C>
```

The "NOTICE:" line is a DEFG informational message and not an error. The subsequent line is the actual error output message. The "clCreateKernel" is the name of the OpenCL API function that failed, "(322)" is the source code line number displaying this error message, and the -44 is the returned OpenCL error code. These displayed error codes are normally listed and described in the "opencl.h" header file.

A.8.3 Useful Debugging Techniques

The C/C++ programs generated by DEFG, like any other programs, may contain errors. Here are some debugging techniques that are known to be useful with DEFG problems and programming mistakes:

- Have available, and use, an existing test case with known results.
- Debug the DEFG program first on a simple computer using a single DEFG device before debugging it on a complex server with high-powered GPUs. Any Windows personal computer large enough to support C/C++ compilers can run OpenCL, with or without an actual GPU defined to OpenCL.
- If possible, debug each program portion separately. DEFG program lines can be commented out by inserting "//" at the beginning of the program line.
- Use call statements to the *dumpScalar()* and *dumpBuffer()* functions to display intermediate results.
- Use code morsels to output descriptive text, as well as processing results.
- Use additional temporary DEFG data buffers to move intermediate results to the CPU for additional analysis.
- Inspect the generated C/C++ code. It is possible to temporarily modify this code with additional debugging statements.

APPENDIX B

Source Code and Other Items

B.1 Hardware and Software Description

We used three different platforms in our research. The vast majority of our work was done on the University of Colorado Denver, Department of Computer Science and Engineering's Penguin Server, known as *Hydra*. In certain instances, two other computers were used. All three are described below in Table B.1.

Name	**Configuration Data**
Server: Hydra	Penguin Computing Cluster, Linux Cent OS 5.3, AMD Opteron 2427 2.2 GHz, 24 GB RAM, using NVIDIA OpenCL SDK 4.0, two NVIDIA Tesla T20s, each with 14 Compute Units, 1147 MHz and 2687M RAM OpenCL: CUDA vers: 5 (driver) and OpenCL 1.1 CUDA 4.2.1 Compiler: gcc version 4.4.4
Server: Rabbit	Single Server, Linux 2.6.32-5-686, AMD Sempron 145, 2.8 GHz, 2 GB RAM, using AMD SDK 2.8, GPU1: AMD HD 7850 2GB RAM and GPU2: AMD Radeon R9 270X 2GB RAM OpenCL: AMD Catalyst 13.25.5 (driver) and AMD OpenCL 2.8 SDK Compiler: gcc version 4.4.5 (Debian 4.4.5-8)
CPU-only	Windows 7, Intel I3 Processor, 1.33 GHz, 4 GB RAM (no GPU) OpenCL: Support via AMD OpenCL 2.8 SDK x86 CPU driver Compiler: Microsoft VC 2008

Table B.1: Testing Configurations, Hardware and Software

B.2 Suggested DEFG Technical Improvements

DEFG, as with many other complex software implementations, can benefit from added features and existing-feature enhancements. Below is a list of potential DEFG fea-

tures, in no particular order, that we have considered for future addition to DEFG.

1. Add a DEFG optimizer step to verify the in/out/inout option settings. These settings are used with execution, multi_exec and call statements. Currently, DEFG does not cross check the option settings; the option settings coded by the developer might be syntactically valid, but semantically wrong. Using the wrong option settings can cause DEFG to omit required transfer operations, creating hard-to-find failures. Verification would facilitate finding DEFG coding errors.

2. The DEFG code statement, the *morsel*, could be enhanced to include an optional list of DEFG field names and their associated in/out/inout settings. This would make it possible for the DEFG optimizer to assist in always having valid data present in the DEFG variables and buffers. In the present version, this is the responsibility of the developer.

3. DEFG currently allocates a DEFG_MAX_BUF-sized CPU memory segment for each declared buffer. This approach can waste a significant amount of CPU memory. Once the CPU buffer is loaded with data, DEFG could release the unused CPU memory. The DEFG CPU buffers are currently allocated with a C++ *malloc()* call. The unused memory could be carefully released with an associated *realloc()* call. The performance implications of this change would need to be explored. We note that the memory allocated on the GPU is *not* allocated in fixed-size blocks; its allocation size is determined by the width of the data stored.

4. The DEFG interchange statement is often used, after an execute or multi_exec statement, to swap the contents of two DEFG buffers. With a syntax change to the execute and multi_exec statements, the interchange statement functionality

could be included within the execute and multi_exec. This would make the DEFG programs simpler, and likely faster.

5. DEFG has the ability to automatically collect run-time statistics for each major OpenCL API request. This facility could be expanded to collect timing data for all major DEFG actions, including the calls to CPU functions and the execution of DEFG morsels. The added information could then be used to obtain detailed run-time profile statistics, which are often quite helpful in achieving high levels of performance.

6. The DEFG optimization is currently performed statically; it is completed just before program generation. In the future, DEFG could also have the option to utilize dynamic (run-time) optimization of the OpenCL buffer transfers. This change would greatly simplify the static optimization and likely permit DEFG to provide additional looping structures.

B.3 The DEFG Mini-Experiment with Four GPUs

One of our goals is to produce applications that utilize multiple GPUs. In this short discussion, we describe our results from performing a mini-experiment with a small, 4-GPU, DEFG application. We wrote a computationally intense DEFG diagnostic program, called DIAG4WAY; it performed, 26 times, the multi_exec of a small kernel. This kernel mainly executed this step:

```
for (int i=0; i < 1024*512; i++) {
  d = d + i; e = (int) (sqrt( (float) d));  d = d - e; d = d + e;
}
```

The purpose of this contrived workload is obviously to keep the GPU very busy. We would have preferred to run our existing MEDIAN5M application in this 4-GPU environment; however, as we lacked administrator privileges on the 4-GPU server, we were not able to install the needed AMD SDK to provide the image-loading modules

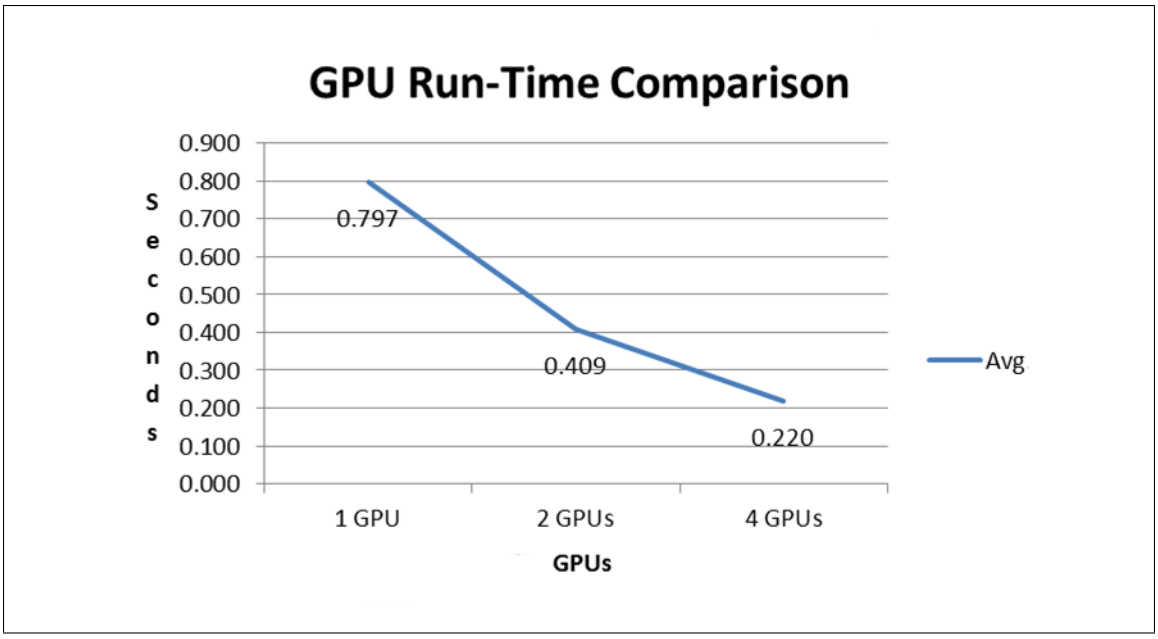

Figure B.1: Run-Time Comparison with 1, 2 and 4 GPUs

needed by our filtering applications. We used DIAG4WAY instead, and we obtained interesting results. The DIAG4WAY source code is in Section B.4.

We ran our diagnostic four times for each of the 1-GPU, 2-GPU, and 4-GPU execution modes and averaged the respective run times. These run-time averages are, 1-GPU: 0.797 secs, 2-GPU: 0.409 secs, and 4-GPU: 0.220 secs. Figure B.1 graphs the execution seconds against the number of GPUs. The 2-GPU test shows a 1.95 speedup and the 4-GPU shows a 3.62 speedup.

DEFG shows a significant multiple-GPU performance gain with this very computationally intense application. A special thanks to Mark Smith, at the Exxact Corporation[1], for providing a weekend of access to one of their GPU-equipped servers. This server had 32 Xeon processors (E5-2660 @ 2.20GHz) and four NVIDA K20, Kepler-generation GPU cards, each with 5119MB of RAM.

[1]Exxact Corporation, Fremont, CA; A distributor of AMD, NVIDIA, and PNY products; URL: www.exxactcorp.com

B.4 DEFG Application Source Code

B.4.1 BFSDP2GPU Application

```
001. // bfsdp2gpu.txt: BFS (Harish-like version) with VPs for 2 GPUs
002. declare application  bfsdp2gpu
003.   declare integer NODE_CNT (0)
004.           integer NODE_CNTt2 (0)
005.           integer NODE_CNT0 (0)
006.           integer NODE_CNT0p1 (0)
007.           integer NODE_CNT1 (0)
008.           integer NODE_CNT1p1 (0)
009.           integer KCNT (0)
010.           integer KCNT0 (0)
011.           integer KCNT1 (0)
012.           integer EDGE_CNT (0)
013.           integer MAX_DEGREE (0)
014.           integer STOP (0)
015.           integer STOP0 (0)
016.           integer STOP1 (0)
017.           integer LIST_WIDTH (150000)
018.           integer listused0 (0)
019.           integer listused1 (0)
020.   declare gpu gpugrp ( all )
021.   declare kernel bermanPrefixSumP1  bfsdp_kernelv3 ( [[ 1D,NODE_CNTt2 ]] )
022.           kernel bermanPrefixSumP2b bfsdp_kernelv3 ( [[ 1D,NODE_CNTt2 ]] )
023.           kernel getCellValue       bfsdp_kernelv3 ( [[ 1D,1 ]] )
024.           kernel kernel1a2          bfsdp_kernelv3 ( [[ 1D,NODE_CNT ]] )
025.           kernel kernel1b           bfsdp_kernelv3( [[ 1D,EDGE_CNT ]] )
026.           kernel kernel2            bfsdp_kernelv3 ( [[ 1D,NODE_CNT ]] )
027.   declare integer buffer graph_edges (EDGE_CNT )  nonpartable
028.           integer buffer frontier0   (LIST_WIDTH) nonpartable // list
029.           integer buffer payload0    (LIST_WIDTH) nonpartable // list
030.           integer buffer frontier1   (LIST_WIDTH) nonpartable // list
031.           integer buffer payload1    (LIST_WIDTH) nonpartable // list
032.           struct (8) buffer graph_nodes      (NODE_CNT) multi
033.           integer buffer graph_mask          (NODE_CNT) multi
034.           integer buffer updating_graph_mask (NODE_CNT) multi
035.           integer buffer graph_visited       (NODE_CNT) multi
036.           integer buffer cost                (NODE_CNT) multi
037.           integer buffer offset              (NODE_CNT) multi
038.           integer buffer offset2             (NODE_CNT) multi
039.           call init_input (graph_nodes(out)
040.                            graph_edges(out)
041.                            graph_mask(out)
042.                            updating_graph_mask(out)
043.                            graph_visited (out)
044.                            cost(out)
045.                            NODE_CNT(in)
046.                            EDGE_CNT(out)
047.                            MAX_DEGREE(out)
048.            )
049.   // partition nodes and some other buffers for multi-GPU use
050.   call ArrayPartition2GPU2 (graph_nodes         (inout)
051.                             graph_visited       (inout)
052.                             cost                (inout)
053.                             graph_mask          (inout)
054.                             updating_graph_mask (inout)
055.                             NODE_CNT0(out)
056.                             NODE_CNT1(out)
057.                             NODE_CNT(in)
058.                             DEFG_GLOB(*)
```

(Continued on next page)

(Continued from previous page)

```
                          )
   code [[ NODE_CNT0p1 = NODE_CNT0 + 1;  NODE_CNT1p1 = NODE_CNT1 + 1;
           NODE_CNTt2 = (NODE_CNT+1) * 2; ]]
   loop
     multi_exec run2 bermanPrefixSumP1 (offset (out)
                                        graph_mask(in)
                                        NODE_CNT0p1 @0 (in)
                                        NODE_CNT1p1 @1 (in)
                                       )
     code [[ KCNT0 = (int) ceil((log((double) NODE_CNT0p1))/log(2.0)); ]]
     code [[ KCNT1 = (int) ceil((log((double) NODE_CNT1p1))/log(2.0)); ]]
     code [[ KCNT = (KCNT0 >= KCNT1) ? KCNT0 : KCNT1; ]]
     sequence KCNT times
       multi_exec run2 bermanPrefixSumP2b (offset2 (inout)
                                           offset (inout)
                                           DEFG_CNT(in)
                                           NODE_CNT0p1 @0 (in)
                                           NODE_CNT1p1 @1 (in)
                                          )
     code [[ if ((KCNT % 2) == 0) {cl_mem s = defg_buffer_offset[0];
                 defg_buffer_offset[0] = defg_buffer_offset2[0];
                 defg_buffer_offset2[0] = s; } ]]
     code [[ if ((KCNT % 2) == 0) {cl_mem s = defg_buffer_offset[1];
                 defg_buffer_offset[1] = defg_buffer_offset2[1];
                 defg_buffer_offset2[1] = s; } ]]
     multi_exec run3 getCellValue (offset2 (in)
                                   NODE_CNT0 @0 (in)
                                   NODE_CNT1 @1 (in)
                                   listused0 @0 (out)
                                   listused1 @1 (out)
                                  )
     call sync (listused0(in))
     call sync (listused1(in))
     code [[ if (listused0 >= LIST_WIDTH || listused1 >= LIST_WIDTH) {
             printf("Error list overflow %d %d.\n",
             listused0, listused1); exit(0);
        } ]]
     multi_exec s2 kernel1a2 (graph_nodes(in)
                              graph_edges (in)
                              graph_mask (inout)
                              offset2 (in)
                              cost (in)
                              frontier0 @0 (out)
                              payload0  @0 (out)
                              NODE_CNT0 @0 (in)
                              frontier1 @1 (out)
                              payload1  @1 (out)
                              NODE_CNT1 @1 (in)
                             )
     broadcast (frontier0 @0)
     broadcast (payload0 @0)
     broadcast (frontier1 @1)
     broadcast (payload1 @1)
     multi_exec s3 kernel1b (frontier0(in)
                             payload0(in)
                             listused0(in)
                             frontier1(in)
                             payload1(in)
                             listused1(in)
                             updating_graph_mask (inout)
                             graph_visited (in)
                             cost (inout)
```

(Continued on next page)

(Continued from previous page)

```
115.                            DEFG_GPU(in)   // which GPU
116.                           )
117.     set STOP0 (0)
118.     set STOP1 (0)
119.     set STOP (0)
120.     multi_exec s4 kernel2 (graph_nodes(in)
121.                            graph_mask(inout)
122.                            updating_graph_mask(inout)
123.                            graph_visited(inout)
124.                            STOP0 @0(inout)
125.                            STOP1 @1(inout)
126                             // STOP (inout) // debug, is wrong
127.                            NODE_CNT0 @0(in)
128.                            NODE_CNT1 @1(in)
129.                           )
130.     call logicalor(STOP(out) STOP0(in) STOP1(in))
131.   while STOP eq 1
132.   // merge costs into 1 array for testing
133.   call MergeCost2GPU2(cost(inout) DEFG_GLOB(*))
134.   call disp_output (cost(*) NODE_CNT(*))
135. end
```

OpenCL Kernels:

```
001. // kernel for DEFG BFSDP2GPU with PrefixSum-based buffer allocation
002. // using "berman" 2-phase PrefixSum version
003. //
004. // macros used to seperate device and node for VP
005. #define MAP_DEVICE(x) (x & 1)
006. #define MAP_NODE(x) (x >> 1)
007. typedef struct
008. {
009.     int starting;
010.     int no_of_edges;
011. }Node;
012. ////
013. //// PrefixSum kernels -- two versions
014. ////
015. // Simple 1-thread prefix sum -- slow but reliable -- unused
016. __kernel void kernelPrefixSum(
017.      __global int* output,      // buffer of sums
018.      __global int* input,       // buffer of values
019.       __global int* block,       // workarea
020.       const int length) // length of buffers
021. {
022.  if (length < 1) return;
023.  // clearly, i must be increasing with each call...
024.  // NOTICE: length+1; goes 1 past end of normal buffer !!!!
025.  for (int k=0; k < length+1; k++) {
026.   if (k == 0) {
027.    output[0] = 0;
028.   } else {
029.    output[k] =  output[k-1] + input[k-1];
030.   }
031.  }
032.  return;
033. }
034. // Berman page 378, part 1
035. // global_work is size
036. __kernel void bermanPrefixSumP1(
037.      __global int* output,      // buffer of partial sums
038.      __global int* input,       // buffer of values
039.                      const int size)           // size of buffer
040. {
041.  int offset = get_global_id(0);
```

(Continued on next page)

(Continued from previous page)

```
042.  if (offset > size) return;
043.  if (offset == 0) {
044.   output[0] = 0;
045.  } else if (offset == 1) {
046.      output[1] = input[0];
047.  } else {
048.   output[offset] = input[offset-2] + input[offset-1];
049.  }
050.  return;
051. }
052. // Berman page 378, part 2
053. __kernel void bermanPrefixSumP2b(
054.      __global int* buffer2,  // buffer of new partial sums
055.      __global int* buffer1,  // buffer of partial sums
056.                      const int CNT,
057.                      const int size)     // size of buffer
058. {
059.  int offset = get_global_id(0);
060.  if (offset >= size) return;
061.  int k = 1 << (CNT + 1);
062.     if ((CNT %2) == 0) {
063.  // buffer1 --> buffer2
064.  buffer2[offset] = buffer1[offset];
065.  if (offset < (k + 1)) return;
066.  buffer2[offset] = buffer1[offset - k] + buffer1[offset];
067.  } else {
068.  // buffer2 --> buffer1
069.  buffer1[offset] = buffer2[offset];
070.  if (offset < (k + 1)) return;
071.  buffer1[offset] = buffer2[offset - k] + buffer2[offset];
072.  }
073.  return;
074. }
075. ////
076. //// getCellValue:
077. ////
078. __kernel void getCellValue(
079.             __global int* buffer,
080.                             const int offset,
081.          __global int* value
082.          )
083. {
084.     *value = buffer[offset];
085. }
086. ////
087. //// kernel1a2
088. ////
089. __kernel void kernel1a2(__global const Node* g_graph_nodes,
090.           __global int* g_graph_edges,
091.           __global int* g_graph_mask,
092.     __global int* g_graph_offset,
093.           __global int* g_cost,
094.     __global int* g_frontier,
095.     __global int* g_payload,
096.           int no_of_nodes) {
097.
098.     unsigned int tid = get_global_id(0);
099.  // in range, in frontier, and has edges
100.  if(tid < no_of_nodes && g_graph_mask[tid] != 0)
101.  {
102.   g_graph_mask[tid] = 0;
103.   if (g_graph_nodes[tid].no_of_edges > 0)
104.   {
105.    int cost = g_cost[tid];
```

(Continued on next page)

(Continued from previous page)

```
106.    int max = (g_graph_nodes[tid].no_of_edges + g_graph_nodes[tid].starting);
107.    int index = g_graph_offset[tid];
108.    for(int i = g_graph_nodes[tid].starting; i < max; i++)
109.    {
110.     int id = g_graph_edges[i];
111.     g_frontier[index] = id;
112.     g_payload[index] = cost;
113.     index++;
114.    }
115.   }
116.  }
117. }
118. ////
119. //// kernel1b
120. ////
121. __kernel void kernel1b(
122.    __global int* g_frontier0,
123.    __global int* g_payload0,
124.    int list_size0,
125.    __global int* g_frontier1,
126.    __global int* g_payload1,
127.    int list_size1,
128.          __global int* g_updating_graph_mask,
129.    __global int* g_graph_visited,
130.          __global int* g_cost,
131.    int gpu_id)
132. {
133.  int index = get_global_id(0);
134.  if (index < list_size0) {
135.   int id = g_frontier0[index];
136.   if (MAP_DEVICE(id) == gpu_id) {
137.       int nid = MAP_NODE(id);
138.    if(!g_graph_visited[nid])
139.    {
140.     g_cost[nid] = g_payload0[index] + 1;
141.     g_updating_graph_mask[nid] = 1;
142.    }
143.   }
144.  }
145.  if (index < list_size1) {
146.   int id = g_frontier1[index];
147.   if (MAP_DEVICE(id) == gpu_id) {
148.    int nid = MAP_NODE(id);
149.    if(!g_graph_visited[nid])
150.    {
151.     g_cost[nid] = g_payload1[index] + 1;
152.     g_updating_graph_mask[nid] = 1;
153.    }
154.   }
155.  }
156. }
157. __kernel void kernel2(__global const Node* g_graph_nodes,
158.     __global int* g_graph_mask,
159.              __global int* g_updating_graph_mask,
160.     __global int* g_graph_visited,
161.     __global int* g_over,
162.     int no_of_nodes)
163. {
164.  unsigned int tid = get_global_id(0);
165.  if(tid < no_of_nodes && g_updating_graph_mask[tid] == 1)
166.  {
167.   g_graph_mask[tid] = g_graph_nodes[tid].no_of_edges; // was 1;
168.   g_graph_visited[tid] = 1; *g_over = 1;
169.   g_updating_graph_mask[tid] = 0;
170.  }
171. }
```

B.4.2 IMIFLX Application

```
001. // imiflx.txt: Altman iterative matrix inversion
002. // * input <pgm> <file>.txt | <file>.mtx | I<size> | M<size> | H<size> [epsilon maxCycles newAlpha]
003. //
004.  declare application  imiflx
005.   include [[
006.  #define INDEX2(xi,xj,xsize,xind) xind = (xi * xsize) + xj;
007.  #ifdef _WIN32
008.  #define isfinite _finite
009.  #endif
010.  ]]
011.   declare integer mSIZE (0)
012.           integer mSIZEt2 (0)
013.           integer LIMIT (0)
014.           integer cycles (200)
015.     integer localWork  (1024)
016.     integer localSize  (2048)   // 2 * localWork
017.     integer basketSize (2048)   // 2 * localWork
018.           double epsilon (0)
019.           double result (0)
020.     double newAlpha (0)
021.           double norm (0)
022.           double alpha (0)
023.           integer LCNT (0)
024.     integer ITR (0)
025.           double dOne (1.0)
026.           double dZero (0.0)
027.           double dThree (3.0)
028.           double dmThree (-3.0)
029.           declare gpu gpuone ( * )
030.           declare kernel CopyArray imiflx ( [[ 1D,mSIZEt2 ]] )
031.                   kernel PlusIdentityThree imiflx ( [[ 1D,mSIZE ]] )
032.                   kernel MinusMatThree imiflex ( [[ 1D,mSIZEt2 ]] )
033.           kernel prefixSum imiflx ( [[ 1D,localWork:localWork ]] )
034.           kernel MinusIdentity imiflx ( [[ 1D,mSIZE ]] )
035.           kernel SweepSquares imiflx ( [[ 1D,basketSize ]] )
036.           kernel ReadLastSqrt imiflx ( [[ 1D,1 ]])
037.   declare double buffer mA    ( mSIZE mSIZE )
038.           double buffer mP    ( mSIZE mSIZE )
039.           double buffer mRn   ( mSIZE mSIZE )
040.           double buffer mRnp1 ( mSIZE mSIZE )
041.           double buffer mW    ( mSIZE mSIZE )
042.           double buffer mS    ( basketSize )
043.           double buffer mBasket (basketSize)
044.           double buffer mLocal  (localSize) local
045.   call defg_matloader( mA(out) mSIZE(out))
046.   code [[ mSIZEt2 = mSIZE * mSIZE; ]]
047.   code [[ if (mSIZEt2 >= ((int) (DEFG_MAX_BUF * sizeof(double))))
                { printf("buffer overflow error\n"); exit(0); } ]]
048.   code [[ cycles = mSIZEt2; ]]
049.   code [[ newAlpha = -1.0; ]]
050.   code [[ epsilon = 0.00001; ]]
051.   code [[ if (argc > 2) { epsilon = atof(argv[2]); } ]]
052.   code [[ if (argc > 3) { cycles = atoi(argv[3]); } ]]
053.   code [[ if (argc > 4) { newAlpha = atof(argv[4]); } ]]
054.   code [[ LIMIT = (mSIZEt2 < 16) ? mSIZEt2 : 16; ]]
055.   code [[ printf("imiflx GPU version using SweepSquares and prefixSum\n"); ]]
056.   code [[ printf("SQ MAT size: %d, epsilon: %lf, max cycles: %d, newAlpha: %g\n",
                    mSIZE, epsilon, cycles, newAlpha); ]]
057.   // compute norm
058.   execute k1 SweepSquares(mA(in) mSIZEt2(in) mBasket(inout) basketSize(in))
059.   execute k2 prefixSum(mS (out) mBasket(in) mLocal(*) basketSize(in))
060.   execute k3 ReadLastSqrt(mS(in) basketSize(in) norm(out))
061.   release (norm)   // gets value onto CPU
```

(Continued on next page)

(Continued from previous page)

```
062.   code [[ alpha = 1.0 / norm; ]]
063.   code [[ if (newAlpha != -1.0) alpha = newAlpha; ]]
064.   code [[ if (alpha < 0.0 || alpha >= 1.0) { printf("Error, invalid alpha!\n"); exit(0); } ]]
065.   // cpu: mRn = Identity Matrix * alpha
066.   code [[ for (int i=0; i < mSIZEt2; i++) {  mRn[i] = 0.0; } ]]
067.   code [[ for (int i=0; i < mSIZE; i++) {   mRn[i*mSIZE +i] = alpha; } ]]
068.   // mP = mA * mRn
069.   blas (dOne * mA * mRn + dZero * mP -> mP)
070.   loop
071.     // desired result: mRnp1 = mRn * (mI*3 - mP*3 + mP2)   mP2 is mP*mP
072.     // mW = mP * mP
073.     blas (dOne * mP * mP + dZero * mW -> mW)
074.     // mW += mI * 3
075.     execute k8 PlusIdentityThree(mW(inout) mSIZE(in))
076.     // mW -= mP * 3
077.     execute k9 MinusMatThree(mW(inout) mP(in) mSIZEt2(in))
078.     // mRnp1 = mRn * mW
079.     blas (dOne * mRn * mW + dZero * mRnp1 -> mRnp1)
080.     // mP = mA * MRnp1
081.     blas (dOne * mA * mRnp1 + dZero * mP -> mP)
082.     // copy mP to mW
083.     execute k10 CopyArray(mW(out) mP(in) mSIZEt2(in))
084.     // mW -= mI
085.     execute k11 MinusIdentity(mW(inout) mSIZE(in))  // note: mSize not mSizet2
086.     // result = norm(mW)
087.     execute k12 SweepSquares(mW(in) mSIZEt2(in) mBasket(inout) basketSize(in))
088.     execute k13 prefixSum(mS (out) mBasket(in) mLocal(*) basketSize(in))
089.     execute k14 ReadLastSqrt(mS(in) basketSize(in) result(out))
090.     release (result)   // gets value onto CPU
091.     code [[ ITR = LCNT + 1; ]]
092.     code [[ if (!isfinite(result)) { printf("infinite result, exiting\n"); LCNT = cycles; } ]]
093.     code [[ if (result != result) { printf("result is nan, exiting\n"); LCNT = cycles; } ]]
094.     code [[ if (result <= epsilon) LCNT = cycles;  ]]
095.     loop_escape at 6 secs   // "anytime" processing
096.     // mRn <==> mRnp1
097.     interchange(mRn mRnp1)
098.     call inc(LCNT(inout))
099.   while LCNT lt cycles
100.   call defg_write_matrix(mRn(in) mSIZE(in))
101.  end
```

OpenCL Kernels: The prefix_sum kernel is used from the AMD OpenCL 2.8 SDK and is copyrighted by AMD; the full kernel source code can be obtained from this SDK.

```
001-076 contain the prefix_sum kernel and are ommited, see above.
077. //
078. // senser-written kernels start here
079. //
080. __kernel void CopyArray(
081.      __global double* output,   // buffer of out data values
082.      __global double* input,    // buffer of in data values
083.       const int length)         // length of buffer
084. {
085.     unsigned int tid = get_global_id(0);
086.  if (tid >=length) return;
087.  output[tid] = input[tid];
088. }
089. __kernel void SweepSquares(
090.      __global double* input,   // buffer of data values
091.       const int length,        // full length of buffer
092.      __global double*  basket, // basket of partial sums
093.      const int basket_length)  // full length of basket
094. {
095.  double d;
096.  double sum = 0.0;
```

(Continued on next page)

(Continued from previous page)

```
097.  // int index;
098.  if (length < 1) return;
099.     unsigned int tid = get_global_id(0);
100.  if (tid >=length) return;
101.  // strides of basket_length size ....
102.  for (int k=tid; k < length; k += basket_length) {
103.   d = input[k] * input[k];
104.   sum += d;
105.  }
106.  basket[tid] = sum;
107.  return;
108. }
109. __kernel void PlusIdentityThree(
110.      __global double* matrix, // buffer of values
111.       const int length)       // size of diag NOT full length
112. {
113.      unsigned int tid = get_global_id(0);
114.   if (tid >= length) return;
115.   unsigned int offset = tid * length + tid;
116.   matrix[offset] += 3;
117. }
118. __kernel void MinusMatThree(
119.      __global double* baseMatrix,                // buffer of result values
120.                       __global double* otherMatrix, // 2nd Matrix buffer
121.       const int length)                           // size of full matrix
122. {
123.      unsigned int tid = get_global_id(0);
124.   if (tid >= length) return;
125.   baseMatrix[tid] -= 3 * otherMatrix[tid];
126. }
127. __kernel void MinusIdentity(
128.      __global double* matrix, // buffer of values
129.       const int length)       // size of diag NOT full length
130. {
131.      unsigned int tid = get_global_id(0);
132.   if (tid >= length) return;
133.   unsigned int offset = tid * length + tid;
134.   matrix[offset] -= 1.0;
135. }
136. __kernel void ZeroBasketDEAD(
137.      __global double* matrix, // basket of future partial sums
138.       const int length)       // size of basket
139. {
140.      unsigned int tid = get_global_id(0);
141.   if (tid >= length) return;
142.   matrix[tid] = 0.0;
143. }
144. __kernel void ReadLastSqrt(
145.      __global double* matrix,  // array
146.       const int length,        // size of array
147.       __global double* last)   // return value
148. {
149.   *last = sqrt(matrix[length - 1]);
150. }
```

B.4.3 MEDIAN Application

```
01. // mediam.txt: Median algorithm in DEFG syntax
02.  declare application  median
03.   declare integer Xdim (0)
04.           integer Ydim (0)
05.           integer BUF_SIZE (0)
06.   declare gpu gpuone ( * )
07.   declare kernel  median_filter  insert code [[
08.   __kernel void median_filter(__global uint* inputImage, __global uint* outputImage)
09.   {
10.     uint sort_buf[9];
11.     uint h;
12.     int i;
13.     int j;
14.     uint x = get_global_id(0);
15.     uint y = get_global_id(1);
16.
17.     uint width = get_global_size(0);
18.     uint height = get_global_size(1);
19.     int c = x + y * width;
20.     if( x >= 1 && x < (width-1) && y >= 1 && y < height - 1)
21.     {
22.      /* outputImage[c] = inputImage[c]; return; */
23.      sort_buf[0] = inputImage[c - 1 - width];
24.      sort_buf[1] = inputImage[c - width];
25.      sort_buf[2] = inputImage[c + 1 - width];
26.      sort_buf[3] = inputImage[c - 1];
27.      sort_buf[4] = inputImage[c];
28.      sort_buf[5] = inputImage[c + 1];
29.      sort_buf[6] = inputImage[c - 1 + width];
30.      sort_buf[7] = inputImage[c + width];
31.      sort_buf[8] = inputImage[c + 1 + width];
32.      for (i=0; i < 9; i++) {
33.       for (j=i; j < 9; j++) {
34.        if (sort_buf[i] > sort_buf[j]) {
35.         h = sort_buf[i];
36.         sort_buf[i] = sort_buf[j];
37.         sort_buf[j] = h;
38.        }
39.       }
40.      }
41.      outputImage[c] = sort_buf[4];
42.     }
43.   }
44.   ]]  ( [[ 2D,Xdim,Ydim ]] )
45.   declare integer buffer image1 ( Xdim Ydim ) halo (1)
46.           integer buffer image2 ( Xdim Ydim ) halo (1)
47.   call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
48.   start_timer
49.     execute run1 median_filter ( image1(in) image2(out) )
50.   call sync (image2(in)) // helps timer accuracy
51.   end_timer
52.   call disp_output (image2(in) Xdim (in) Ydim (in) )
53.   output_timer
54.  end
```

B.4.4 MEDIAN5 Application

```
01. // mediam5.txt: Median algorithm in DEFG syntax
02. // use 5x5, not 3x3, "window" to compute the median
03.  declare application  median
04.   declare integer Xdim (0)
05.           integer Ydim (0)
06.           integer BUF_SIZE (0)
07.   declare gpu gpuone ( * )
08.   declare kernel  median_filter  insert code [[
09.   __kernel void median_filter(__global uint* inputImage, __global uint* outputImage)
10.   {
11.     uint sort_buf[25];
12.     uint h;
13.     int i;
14.     int j;
15.     int k;
16.     uint x = get_global_id(0);
17.     uint y = get_global_id(1);
18.
19.     uint width = get_global_size(0);
20.     uint height = get_global_size(1);
21.     int c = x + y * width;
22.     if( x >= 2 && x < (width - 2) && y >= 2 && y < (height - 2))
23.     {
24.      k = 0;
25.      for (i=-2; i < 3; i++) {
26.        sort_buf[k++] = inputImage[c + i - width - width];
27.      }
28.      for (i=-2; i < 3; i++) {
29.        sort_buf[k++] = inputImage[c + i - width];
30.      }
31.      for (i=-2; i < 3; i++) {
32.        sort_buf[k++] = inputImage[c + i];
33.      }
34.      for (i=-2; i < 3; i++) {
35.        sort_buf[k++] = inputImage[c + i + width];
36.      }
37.      for (i=-2; i < 3; i++) {
38.        sort_buf[k++] = inputImage[c + i + width + width];
39.      }
40.      for (i=0; i < 25; i++) {
41.        for (j=i; j < 25; j++) {
42.          if (sort_buf[i] > sort_buf[j]) {
43.            h = sort_buf[i];
44.            sort_buf[i] = sort_buf[j];
45.            sort_buf[j] = h;
46.          }
47.        }
48.      }
49.      outputImage[c] = sort_buf[12];
50.     }
51.   }
52.   ]]  ( [[ 2D,Xdim,Ydim ]] )
53.   declare integer buffer image1 ( Xdim Ydim ) halo (2)
54.           integer buffer image2 ( Xdim Ydim ) halo (2)
55.   call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
56.   execute run1 median_filter ( image1(in) image2(out) )
57.   call disp_output (image2(in) Xdim (in) Ydim (in) )
58.  end
```

B.4.5 MEDIAN5M Application

```
01. // median5m.txt: Multi-GPU Median 5x5 algorithm in DEFG syntax
02. // use 5x5 "window" to compute the median
03.  declare application  nedian5
04.  declare integer Xdim (0)
05.          integer Ydim (0)
06.          integer BUF_SIZE (0)
07.   declare gpu gpuone ( all )
08.   declare kernel  median5_filter  insert code [[
09.   __kernel void median5_filter(__global uint* inputImage, __global uint* outputImage)
10.   {
11.   uint sort_buf[25];
12.   uint h;
13.   int i;
14.   int j;
15.   int k;
16.   uint x = get_global_id(0);
17.   uint y = get_global_id(1);
18.
19.   uint width = get_global_size(0);
20.   uint height = get_global_size(1);
21.   int c = x + y * width;
22.   if( x >= 2 && x < (width - 2) && y >= 2 && y < (height - 2))
23.   {
24.    k = 0;
25.    for (i=-2; i < 3; i++) {
26.      sort_buf[k++] = inputImage[c + i - width - width];
27.    }
28.    for (i=-2; i < 3; i++) {
29.      sort_buf[k++] = inputImage[c + i - width];
30.    }
31.    for (i=-2; i < 3; i++) {
32.      sort_buf[k++] = inputImage[c + i];
33.    }
34.    for (i=-2; i < 3; i++) {
35.      sort_buf[k++] = inputImage[c + i + width];
36.    }
37.    for (i=-2; i < 3; i++) {
38.      sort_buf[k++] = inputImage[c + i + width + width];
39.    }
40.    for (i=0; i < 25; i++) {
41.     for (j=i; j < 25; j++) {
42.      if (sort_buf[i] > sort_buf[j]) {
43.       h = sort_buf[i];
44.       sort_buf[i] = sort_buf[j];
45.       sort_buf[j] = h;
46.      }
47.     }
48.    }
49.    outputImage[c] = sort_buf[12];
50.   }
51.  }
52.  ]]  ( [[ 2D,Xdim,Ydim ]] )
53.   declare integer buffer image1 ( Xdim Ydim ) halo (2)
54.           integer buffer image2 ( Xdim Ydim ) halo (2)
55.   call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
56.   multi_exec run1 median5_filter ( image1(in) image2(out) )
57.   call disp_output (image2(in) Xdim (in) Ydim (in) )
58.  end
```

B.4.6 RSORT Application

```
01. // RSort.txt: Altman's roughly sort algorithm in DEFG syntax
02. // useage: pgm <file or genK> <size>
03. // arg1 can be an input file or the genK value if generating data
04. // arg2 is the size, if generating data, 2**size
05. declare application  RSort
06.    include [[ char* vers = "V1.1"; ]]
07.    declare integer stride (1)
08.            integer size   (64)
09.            integer sizeDB (0)
10.            integer genK   (0)
11.            integer bufSize (0)
12.            integer radius (1)
13.            integer groups (0)
14.            integer again  (0)
15.            integer offset (0)
16.            integer offset2 (0)
17.            integer logSize (0)
18.    declare gpu gpuone ( * )
19.    declare kernel LRmax RSort_Kernels  ( [[ 1D,size ]] )
20.            kernel RLmin RSort_Kernels  ( [[ 1D,size ]] )
21.            kernel DM RSort_Kernels  ( [[ 1D,size ]] )
22.            kernel UB RSort_Kernels  ( [[ 1D,size ]] )
23.            kernel comb_sort RSort_Kernels  ( [[ 1D,groups ]] )
24.    declare integer buffer arrayS (bufSize )
25.            integer buffer LR     (bufSize)
26.            integer buffer LRout  (bufSize)
27.            integer buffer RL     (bufSize)
28.            integer buffer RLout  (bufSize)
29.            integer buffer DMbuf  (bufSize)
30.    code [[ char* arg = "16"; if (argc > 1) {arg = argv[1];} ]]
31.    code [[ if (argc > 2) { size = (int) pow(2.0, (double) atoi(argv[2])); } ]]
32.    code [[ if ( ((int) (size * sizeof(int))) >= DEFG_MAX_BUF)
                { printf("Error, buffer too small!\n"); exit(0);}
           ]]
33.    // has to be C-style invocation due to arg being a string ...
34.    code [[ getArray(arg, arrayS, size); bufSize = size; ]]
35.    code [[ logSize =  int(log(double(size))/log(2.0));  ]]
36.    code [[ if (bufSize > 16) sizeDB = 16; else sizeDB = bufSize; ]]
37.    code [[ printf("version %s size: %d, logSize: %d\n", vers, size, logSize); ]]
38.    // ==> LR
39.    set stride (1)
40.    execute LR1 LRmax (arrayS(in) LR(out) stride(in))
41.    call times2(stride(*))
42.    set again (1)
43.    loop
44.      execute LR2 LRmax (LR(inout) LRout(out) stride(in))
45.      call times2(stride(*))
46.      interchange(LR LRout)
47.      code [[ again++; ]]
48.    while again lt logSize
49.    // ==> RL
50.    set stride (1)
51.    execute RL1 RLmin (arrayS(in) RL(out) stride(in))
52.    call times2(stride(*))
53.    set again (1)
54.    loop
55.      execute RL2 RLmin (RL(in) RLout(out) stride(in))
56.      call times2(stride(*))
57.      interchange(RL RLout)
58.      code [[ again++; ]]
59.    while again lt logSize
60.    // ==> DM
61.    execute DM1 DM (LR(in) RL(in) DMbuf(out))
```

(Continued on next page)

(Continued from previous page)

```
62.    // ==> FU
63.    call cpy(groups(*) size(*))
64.    loop
65.      set again (0)
66.      call times2(radius(*))
67.      execute UB1 UB (DMbuf(in) size(in) radius(in) again(inout))
68.    while again ne 0
69.    code [[ radius *= 2; ]]
70.    code [[ if (radius > size) { radius = size; } ]]
71.    code [[ groups = (int) ceil( ((double) size / (double) radius)); ]]
72.    // ==> COMB SORT pass 1
73.    execute SORT1 comb_sort(arrayS(inout) radius(in) offset(in) groups(in))
74.    // ==> COMB SORT pass 2
75.    code [[ offset2 = radius / 2; ]]
76.    // lower groups by one and then put it back
77.    call dec(groups(*))
78.    // groups is used implicitly, see kernel declare ...
79.    execute SORT2 comb_sort(arrayS(inout) radius(in) offset2(in) groups(in))
80.    call inc(groups(*))
81.    call sync (arrayS(in))
82.    code [[ putMergeArray("sorted.txt", arrayS, size); ]]
83. end
```

OpenCL Kernels:

```
001. /*
002.  * For a description of the algorithm and the terms used, please see:
003.  * http://en.wikipedia.org/wiki/Comb_sort
004. */
005. __kernel void comb_sort(__global int* base, uint size, uint offset, uint groups)
006. {
007.  uint block = get_global_id(0);
008.  if (block >= groups) return;  // used with multi-GPU
009.     __global int* input;
010.     const float shrink = 1.3f;
011.     int swap;
012.     uint i, gap = size;
013.     bool swapped = false;
014.
015.     input = base + (block * size) + offset;
016.     while ((gap > 1) || swapped) {
017.         if (gap > 1) {
018.             gap = (size_t)((float)gap / shrink);
019.         }
020.
021.         swapped = false;
022.
023.         for (i = 0; gap + i < size; ++i) {
024.             if (input[i] - input[i + gap] > 0) {
025.                 swap = input[i];
026.                 input[i] = input[i + gap];
027.                 input[i + gap] = swap;
028.                 swapped = true;
029.             }
030.         }
031.     }
032.
033. }
034. __kernel void LRmax(__global int src[], __global int dst[], uint stride)
035. {
036.  // src is input array
037.  // dst is output array
038.  // stride is the offset to the relevant rhs cells
```

(Continued on next page)

(Continued from previous page)

```
039.  uint block = get_global_id(0);
040.  uint size = get_global_size(0);
041.  uint arnold = size - stride;
042.  if (block >= arnold) return;  /* Terminator! */
043.  uint js = block + stride;
044.  if (js >= size) return;   /* Terminator II */
045.  int src_j_item = src[block];
046.  int src_js_item = src[js];
047.  if (block < stride) {  // copy already processed
048.   dst[block] = src_j_item;
049.  }
050.  if (src_js_item < src_j_item) {
051.   dst[js] = src_j_item;
052.  } else {
053.   dst[js] = src_js_item;
054.  }
055. }
056. __kernel void RLmin(__global int src[], __global int dst[], uint stride)
057. {
058.  // src is input array
059.  // dst is output array
060.  // blk_width is number of compares for this thread to perform
061.  uint block = get_global_id(0);
062.  uint size = get_global_size(0);
063.  if (block < stride) return;  /* Terminator! */
064.  int js = block - stride;
065.  if (js < 0) return;   /* Terminator II */
066.  if (block >= (size - stride)) {  // copy already processed
067.   dst[block] = src[block];
068.  }
069.  if (src[js] > src[block]) {
070.   dst[js] = src[block];
071.  } else {
072.   dst[js] = src[js];
073.  }
074. }
075. __kernel void DM(__global int B[], __global int C[], __global int D[])
076. {
077.  int i = get_global_id(0);
078.  for (int j = i; j >= 0; j--) {
079.   if ((j <=i) && (i>=0) && C[i]<=B[j] && ((j==0) || (C[i]>=B[j-1]))) {
080.    D[i] =  i-j;
081.    break;
082.   }
083.  }
084. }
085. __kernel void UB( __global int D[], uint size, int d, __global uint *again)
086. {
087.  if (*again == 1) return;  // speeds up CPU version ??
088.  int i = get_global_id(0);
089.  if ((D[i]+0) <= d) {  // this +1 sets up a better radius for sorting control
090.   // good
091.  } else {
092.   *again = 1;
093.  }
094. }
095. __kernel void UBsplit( __global int D[], __global uint again[],  uint size, int d)
096. {
097.     again[1] = d;
098.     if (again[0] != 0) return;  // speeds up CPU version ??
099.     int i = get_global_id(0);
100.     // debug < to <= ??
101.     if ((D[i]) > d) {  // this +1 sets up a better radius for sorting control
102.        again[0] = D[i]; // 1;
```

(Continued on next page)

(Continued from previous page)

```
103.         again[1] = i;
104.     }
105. }
106. __kernel void UBreset( __global uint again[])
107. {
108.     if (get_global_id(0) > 0) return;
109.     again[0] = 0;
110. }
```

B.4.7 RSORTM Application

```
01. // RSortm.txt: Altman's roughly sort algorithm in DEFG syntax, multi GPU
02.  declare application  RSortm
03.    include [[
04.  char* vers = "MV1.1b";
05.  ]]
06.   declare integer groupSize (1)
07.           integer stride (1)
08.           integer size   (4194304)
09.           integer sizeDB (0)
10.           integer genK   (0)
11.           integer bufSize (0)
12.           integer radius (1)
13.           integer groups (0)
14.           integer groupsMulti (0)
15.           integer again  (0)
16.           integer offset (0)
17.           integer offset2 (0)
18.           integer logSize (0)
19.           integer againSize (4)
20.   declare gpu gpugrp ( all )
21.   declare kernel LRmax     RSort_Kernels  ( [[ 1D,size ]] )
22.           kernel RLmin     RSort_Kernels  ( [[ 1D,size ]] )
23.           kernel SHexit    RSort_Kernels  ( [[ 1D,size ]] )
24.           kernel DM        RSort_Kernels  ( [[ 1D,size ]] )
25.           kernel UBsplit   RSort_Kernels  ( [[ 1D,size ]] )
26.           kernel UBreset   RSort_Kernels  ( [[ 1D,size ]] )
27.           kernel comb_sort RSort_Kernels  ( [[ 1D,groups ]] )
28.   declare integer buffer arrayS (bufSize)
29.           integer buffer LR     (bufSize)
30.           integer buffer LRout  (bufSize)
31.           integer buffer RL     (bufSize)
32.           integer buffer RLout  (bufSize)
33.           integer buffer DMbuf  (bufSize)
34.           integer buffer againPart (againSize)
35.   code [[ char* arg = "16"; if (argc > 1) {arg = argv[1];} ]]
36.   code [[ if (argc > 2) { size = (int) pow(2.0, (double) atoi(argv[2])); } ]]
37.   code [[ if ( ((int) (size * sizeof(int))) >= DEFG_MAX_BUF)
38.              { printf("Error, buffer too small!\n"); exit(0); } ]]
39.   code [[ getArray(arg, arrayS, size); bufSize = size; ]]
40.   code [[ if (bufSize > 16) sizeDB = 16; else sizeDB = bufSize; ]]
41.   code [[ logSize =  int(log(double(size))/log(2.0)) - 1 ;  ]]  // <<---- multi change
42.   code [[ printf("version %s size: %d, logSize: %d\n", vers, size, logSize); ]]
43.   // ==> LR
44.   set stride (1)
45.   multi_exec LR1 LRmax (arrayS(in) LR(out) stride(in))
46.   call times2(stride(*))
47.   // main LR loop
48.   set again (1)
49.   loop
50.     multi_exec LR2 LRmax (LR(inout) LRout(out) stride(in))
51.     call times2(stride(*))
52.     interchange(LR LRout)
53.     code [[ again++; ]]
54.   while again lt logSize
55.   // ==> RL
56.   set stride (1)
57.   multi_exec RL1 RLmin (arrayS(in) RL(out) stride(in))
58.   call times2(stride(*))
59.   // main RL loop
60.   set again (1)
61.   loop
62.     multi_exec RL2 RLmin (RL(inout) RLout(out) stride(in))
```

(Continued on next page)

(Continued from previous page)

```
63.     call times2(stride(*))
64.     interchange(RL RLout)
65.     code [[ again++; ]]
66.   while again lt logSize
67.   // ==> DM
68.   multi_exec DM1 DM (LR(in) RL(in) DMbuf(out))
69.   call cpy(groups(*) size(*))
70.   loop
71.     multi_exec UB1 UBreset (againPart(inout))
72.     call times2(radius(inout))
73.     multi_exec UB2 UBsplit (DMbuf(in) againPart(inout) size(in) radius(in))
74.     call sync (againPart(in))
75.     code [[again = againPart[0] + againPart[2]; ]]
76.   while again ne 0
77.   code [[ radius *= 2; ]]
78.   code [[ groups = (int) ceil( ((double) size / (double) radius)); ]]
79.   code [[ if (groups < 2) { printf("sort ended, too few sort groups, use 1 GPU sort!\n"); exit(1); } ]]
80.   // ==> COMB SORT pass 1
81.   code [[ groupsMulti = groups / DEFG_GPU_COUNT; ]]
82.   multi_exec SORT1 comb_sort(arrayS(inout) radius(in) offset(in) groupsMulti(in))
83.   // ==> COMB SORT pass 2
84.   code [[ offset2 = radius / 2; ]]
85.   call dec(groups(*))
86.   code [[ groupsMulti = groups / DEFG_GPU_COUNT; ]]  // this is critical!!
87.   multi_exec SORT2 comb_sort(arrayS(inout) radius(in) offset2(in) groupsMulti(in))
88.   call sync (arrayS(in))
89.   code [[ putMergeArray("sorted.txt", arrayS, size); ]]
90.  end
```

OpenCL kernels for RSORTM are the RSORT kernels.

B.4.8 SOBEL Application

```
01. // Sobel.txt: Sobel algorithm in DEFG syntax
02.  declare application  sobel
03.   declare integer Xdim (0)
04.           integer Ydim (0)
05.           integer BUF_SIZE (0)
06.   declare gpu gpuone ( * )
07.   declare kernel  sobel_filter SobelFilter_Kernels  ( [[ 2D,Xdim,Ydim ]] )
08.   declare integer buffer image1 ( Xdim Ydim ) halo (1)
09.           integer buffer image2 ( Xdim Ydim ) halo (1)
10.   call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
11.   execute run1 sobel_filter ( image1(in) image2(out) )
12.   call disp_output (image2(in) Xdim (in) Ydim (in) )
13.  end
```

OpenCL Kernel:

The SOBEL kernel is used from the AMD OpenCL 2.8 SDK and is copyrighted by AMD. The full kernel source code can be obtained from this SDK.

B.4.9 SOBELM Application

```
01. // Sobelm.txt: Sobel algorithm in DEFG syntax
02. //             'm' version for 2 GPU execution
03.  declare application  sobelm
04.   declare integer Xdim (0)
05.           integer Ydim (0)
06.           integer BUF_SIZE (0)
07.   declare gpu gpuone ( all )
08.   declare kernel  sobel_filter SobelFilter_Kernels  ( [[ 2D,Xdim,Ydim ]] )
09.   declare integer buffer image1 ( Xdim Ydim ) halo (1)
10.           integer buffer image2 ( Xdim Ydim ) halo (1)
11.   call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
12.   multi_exec run1 sobel_filter ( image1(in) image2(out) )
13.   call disp_output (image2(in) Xdim (in) Ydim (in) )
14.  end
```

OpenCL Kernel:

The SOBEL kernel is used from the AMD OpenCL 2.8 SDK and is copyrighted by AMD. The full kernel source code can be obtained from this SDK.

B.5 DEFG Diagnostic Source Code

B.5.1 diagAT Diagnostic Program

```
01. //
02. // DiagAT.txt: diagnostic to verify anytime escape at in DEFG
03. //
04. declare application  diagat
05.   declare integer BUF_SIZE (2)
06.           integer iMore (1)
07.   declare gpu gpuone ( * )
08.   declare kernel  dummy_kernel insert code [[
09.     __kernel void dummy_kernel(__global int* p1)
10.     {
11.       p1[0] = 34;
12.       return;
13.     }
14.   ]]  ( [[ 1D,1 ]] )
15.          kernel  dummy_kernel2 insert code [[
16.     __kernel void dummy_kernel2(__global int* p1)
17.    {
18.      p1[0] = 99;
19.      return;
20.    }
21.   ]]  ( [[ 1D,1 ]] )
22.   declare integer buffer b1 ( BUF_SIZE )
23.           integer buffer b2 ( BUF_SIZE )
24.   start_timer
25.   loop
26.     execute run1 dummy_kernel ( b1(out) )  // b1[0] = 34
27.     execute run1 dummy_kernel2 ( b2(out) )  // b2[0] = 99
28.     call sync(b1(in))
29.     call sync(b2(in))
30.     loop_escape at 20 ms
31.   while iMore eq 1
32.   end_timer
33.   code [[ if (b1[0] == 34 && b2[0] == 99 )
                printf("ok.\n"); else printf("error.\n");  ]]
34.   output_timer
35. end
```

B.5.2 diagIK Diagnostic Program

```
01. //
02. // DiagIK.txt: diagnostic to verify include kernel code in DEFG
03. //
04.  declare application  diagik
05.   declare integer BUF_SIZE (2)
06.   declare gpu gpuone ( * )
07.   declare kernel  dummy_kernel insert code [[
08.   __kernel void dummy_kernel(__global int* p1)
09.   {
10.     p1[0] = 34;
11.    return;
12.   }
13.   ]]  ( [[ 1D,1 ]] )
14.          kernel  dummy_kernel2 insert code [[
15.   __kernel void dummy_kernel2(__global int* p1)
16.   {
17.     p1[0] = 99;
18.     return;
19.   }
20.   ]]  ( [[ 1D,1 ]] )
21.   declare integer buffer b1 ( BUF_SIZE )
22.           integer buffer b2 ( BUF_SIZE )
23.   start_timer
24.   execute run1 dummy_kernel ( b1(out) )  // b1[0] = 34
25.   execute run1 dummy_kernel2 ( b2(out) )  // b2[0] = 99
26.   call sync(b1(in))
27.   call sync(b2(in))
28.   end_timer
29.   code [[ if (b1[0] == 34 && b2[0] == 99 )
                 printf("ok.\n"); else printf("error.\n");  ]]
30.   output_timer
31. end
```

B.5.3 diag4way Diagnostic Program

```
01. //
02. // Diag4way.txt: diagnostic to verify 4-way operations
03. // vers: C5D4_D6C7D6E2
04.  declare application  diag4way
05.    declare integer BUF_SIZE (262144)
06.            integer N (26)
07.            integer iMore (1)
08.    declare gpu gpuall ( all )
09.    declare kernel  dummy_kernel insert code [[
10.    __kernel void dummy_kernel(__global int* p1)
11.    {
12.      uint index = get_global_id(0);
13.      p1[index] = 2 * p1[index];
14.      int d = 0;
15.      int e;
16.      for (int i=0; i < 1024*512; i++) { d = d + i;
             e = (int) (sqrt( (float) d));  d = d - e; d = d + e; }
17.      p1[index] += d;
18.      p1[index] -= d;
19.      return;
20.    }
21.    ]]  ( [[ 1D,BUF_SIZE ]] )
22.    declare integer buffer b1 ( BUF_SIZE )
23.    code [[ for (int ii=0; ii < BUF_SIZE; ii++) { b1[ii] = ii + 1; } ]]
24.    start_timer
25.    sequence N times
26.      multi_exec run1 dummy_kernel ( b1(inout) )
27.    call sync(b1(in))
28.    end_timer
29.    code [[ int err = 0;
30.      printf("(version 1) N: %d\n", N);
31.      int factor = 1;
32.      for (int ii=0; ii < N; ii++) { factor = factor * 2; }
33.      printf("factor: %d\n", factor);
34.      for (int ii=0; ii < BUF_SIZE; ii++) {
             if (b1[ii] != (factor * (ii + 1)) ) { err = ii+1; break; }
         }
35.      if (err == 0) printf("ok.\n");
         else printf("error. (%d %d)\n", err-1, b1[err-1] );
36.    ]]
37.   output_timer
38. end
```

B.6 DEFG Major Components

B.6.1 ParserV2.g: DEFG Grammar Source Code

DEFG Parser Antler code available from the Department of Computer Science and Engineering, University of Colorado Denver.

B.6.2 DEFGopt.java: DEFG Optimizer Source Code

DEFG Optimizer Java code available from the Department of Computer Science and Engineering, University of Colorado Denver.

B.6.3 defgv2.cpp: DEFG Code Generator Source Code

DEFG Code Generator C++ code available from the Department of Computer Science and Engineering, University of Colorado Denver.