On the Complexity of Robust Source-to-Source Translation from CUDA to OpenCL

Paul D. Sathre

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Wu-chun Feng, Chair
Mark K. Gardner
Eli Tilevich

April 8, 2013
Blacksburg, Virginia

Keywords: Source Translation, Clang, CUDA, OpenCL, GPU, GPGPU, Compilers
©Copyright 2013, Paul D. Sathre

On the Complexity of Robust Source-to-Source Translation from CUDA to OpenCL

Paul D. Sathre

ABSTRACT

The use of hardware accelerators in high-performance computing has grown increasingly prevalent, particularly due to the growth of graphics processing units (GPUs) as general-purpose (GPGPU) accelerators. Much of this growth has been driven by NVIDIA's CUDA ecosystem for developing GPGPU applications on NVIDIA hardware. However, with the increasing diversity of GPUs (including those from AMD, ARM, and Qualcomm), OpenCL has emerged as an open and vendor-agnostic environment for programming GPUs as well as other parallel computing devices such as the CPU (central processing unit), APU (accelerated processing unit), FPGA (field programmable gate array), and DSP (digital signal processor).

The above, coupled with the broader array of devices supporting OpenCL and the significant conceptual and syntactic overlap between CUDA and OpenCL, motivated the creation of a CUDA-to-OpenCL source-to-source translator. However, there exist sufficient differences that make the translation non-trivial, providing practical limitations to both manual and automatic translation efforts. In this thesis, the performance, coverage, and reliability of a prototype CUDA-to-OpenCL source translator are addressed via extensive profiling of a large body of sample CUDA applications. An analysis of the sample body of applications is provided, which identifies and characterizes general CUDA source constructs and programming practices that obstruct our translation efforts. This characterization then led to more robust support for the translator, followed by an evaluation that demonstrated the performance of our automatically-translated OpenCL is on par with the original CUDA for a subset of sample applications when executed on the same NVIDIA device.

This work was supported in part by NSF I/UCRC IIP-0804155 via the NSF Center for High-Performance Reconfigurable Computing (CHREC).

Acknowledgements

Looking back on the winding path that has led me to this achievement, I struggle to put to words the gratitude I feel to the people around me who have helped make it a reality. I can honestly say that without their phenomenal support, I could not have made it to where I am today.

To my advisors and mentors, Dr. Wu Feng and Dr. Mark Gardner: I thank you for inspiring the young undergraduate me who had too much time on his hands to challenge myself and explore the cutting-edge of high performance computing. Throughout the four years since, you have provided me with troves of opportunities and knowledge that have helped shape the researcher and developer I am today. I am immensely grateful for your guidance, understanding, and most importantly, patience as I've struggled to learn focus and how to navigate my graduate work.

To the members of the Synergy Lab: I thank you for our numerous discussions throughout the years, your technical insights, and your humanity. You have been fantastic role models and colleagues, and your support has been invaluable during the stressful and disillusioning periods of my academic growth.

To my teachers, supervisors, and other mentors throughout my life: I thank the multitude of you who've shared your talents and wisdom, and provided me the opportunities, encouragement, and freedom to chase all of my passions.

To my family: I can scarcely begin to express my gratitude for the unwavering support you've provided throughout my life. From my youngest days you showed me the value of books, nature, sound and silence, ideas and dreams, culture, and balance. Thank you for being a light in dark times and the biggest supporters of my dreams. Thank you for all you've done to help mould me into the scientist and human being that I am today.

And to my friends: I thank you for constantly reminding me that life is about more than just where you go and what you do; it's about who you share the journey with. You have all been the source of my freedom, my sanity, and my favorite adventures. Thank you for sharing the small moments, the laughter and tears, that keep me going.

Contents

1 Introduction **1**

1.1 Comparison of CUDA to OpenCL 3

1.2 Related Works 6

1.2.1 CU2CL Translator Prototype 9

1.3 Research Approach 11

1.4 Contributions 12

1.5 Thesis Organization 12

2 Characterization of Translator Capability **14**

2.1 Test Applications 15

2.1.1 CUDA SDK 15

2.1.2 Rodinia 16

2.1.3 Other Large Applications 16

2.2 Test Environment 18

2.3 Translator Performance 18

2.4 Translator Reliability 20

2.5 Translator Coverage 21

2.6 Translated Application Performance 24

3 Improved Robustness **26**

3.1 Features Added 27

3.1.1 Inline Error Reporting 27

3.1.2 Update to Clang 3.2 28
3.1.3 Device-side Builtin Math Functions 29
3.1.4 Partial Support for `cudaSetDevice` 30
3.1.5 Partial Support for Literal Parameter Values to Kernels 31
3.1.6 Scaffolding for Struct Alignment Attribute Handling 34
3.1.7 Re-enabling Timing 35
3.2 Mistranslations Repaired 36
3.2.1 Support for Device Buffers as Members of Host-side Structures 36
3.2.2 Handling of `cudaMemcpyToSymbol` 37
3.2.3 File Overwrites Due to Naming Convention 38
3.2.4 Handling of Device-Side Kernel Parameter Rewrites 39
3.2.5 Removal of Functions Lacking Explicitly-Declared Return Type 40
3.2.6 Inhibited Translation of Implicitly-Defined Functions 41
3.2.7 Host-side Arrays of Device Buffers 42
3.3 Bugs Quarantined 42
3.3.1 Template Handling 43
3.3.2 Launching a Kernel Function Pointer 44
3.3.3 Failure When Main Method is not Present 46
3.3.4 Handling of Separately Declared and Defined Kernels 47
3.3.5 Handling for `#defined` Functions 48

4 Characterization of Translator Limitations 49
4.1 Partially Supported 1-to-1 Mappings 50
4.1.1 Device Buffer `cl_mem` Propagation 50
4.2 Unsupported 1-to-1 Mappings 52
4.2.1 Constant Memory 52
4.2.2 Shared Memory 53
4.2.3 Texture to Image Translation 54
4.3 Functionally Emulatable Mappings 55

4.3.1 Kernel Function Pointer Invocation 56

4.3.2 Warp-level Synchronization 57

4.3.3 OpenGL Interoperability 58

4.4 Device Language Expressivity Limitations 59

4.4.1 Mapping C++ to C 59

4.4.2 Threadfence and Other Intrinsic Device Functions 60

4.5 Inherited Limitations 61

4.5.1 Preprocessor Macros 61

4.5.2 Separate Compilation 62

4.5.3 Precompiled Modules 63

4.6 Engineering Limitations 64

5 Conclusion **66**

Appendix **69**

Bibliography **75**

List of Figures

1.1 Setting CUDA/OpenCL Execution Configuration 3
1.2 CUDA and OpenCL Execution Models 4
1.3 CUDA and OpenCL Memory Models 5

2.1 Translator Performance vs. Source Lines of Code 19
2.2 CU2CL Translator Reliability 20

3.1 Comparison of CUDA Kernel Launch Syntaxes 32
3.2 Comparison of Literal Parameter Handling 33
3.3 Comparison of CUDA/OpenCL Alignment Attributes 34
3.4 Device Buffers as Struct Members 37
3.5 OpenCL Output Naming Conventions 38
3.6 Demonstration of Kernel Parameter Rewrites 39
3.7 Kernel Function Pointer Launches 45

4.1 Dynamically allocated shared memory 53
4.2 Warp-level synchronization in reduce 57

List of Tables

2.1 Coverage of CU2CL Translation 24

2.2 Run Times of CUDA Applications and OpenCL Ports 25

4.1 Frequency of Translation Challenges in Sample Applications 50

A.1 Rodinia Sample Translation Times 69

A.2 CUDA SDK Sample Translation Times 70

A.3 CUDA SDK Sample Translation Times (cont.) 71

A.4 Large Application Translation Times 72

A.5 Rodinia Sample Translation Robustness 72

A.6 CUDA SDK Sample Translation Robustness 73

A.7 CUDA SDK Sample Translation Robustness (cont.) 74

Chapter 1

Introduction

Reminiscent of the days when discrete floating point units remained separate from machines' central CPUs, in recent years graphics processing units (GPUs) — as well as other discrete hardware components, such as field programmable gate arrays (FPGAs) — have seen rapid growth as hardware accelerators for general parallel applications. This growth has been driven largely by the massive performance gains possible given the intrinsic extreme data-parallelism afforded by GPUs. However, at the onset of general purpose GPU (GPGPU) computing, there were no easily-used application programming interfaces (APIs) for developing computational applications for graphics devices. Rather, key computations were labor-intensively "force-fit" into traditional graphics shader languages. There were efforts to provide general-purpose compute languages for GPGPU devices, such as BrookGPU [4], but adoption was somewhat limited. However, in 2006 NVIDIA released a programming environment designed to dramatically simplify the development of data-parallel applications for their graphics hardware, known as the Compute Unified Device Architecture (CUDA) [29]. This development environment allowed application programmers to write highly-parallel "kernels" for execution on NVIDIA GPUs alongside standard CPU code via a relatively easy to grasp

variant of C/C++, known as CUDA C. With access to a more programmable method for utilizing GPUs as accelerators, developers were able to more easily achieve often multiple orders of magnitude acceleration over their serial CPU codes, driving a surge in interest.

Apple, however, wanted a similar means of easily writing data parallel applications but wished for greater flexibility in hardware choices than the proprietary nature of CUDA afforded. Thus, they began the development of OpenCL, which was later transferred to the Khronos Group to steward as a vendor-neutral, open standard API for developing data parallel applications. Notably, it was intended not just for GPUs, but for any other current or future hardware for which a vendor wished to implement the standard. NVIDIA, as well as other high-powered and embedded hardware vendors such as AMD, Intel, and IBM played key roles in the development of the standard, however much of its conceptual foundation draws heavily upon the CUDA model. Therefore, the two APIs show a remarkable similarity, despite the obvious difference of CUDA being officially supported solely on NVIDIA hardware and OpenCL's potential support on a vast array of hardware from multiple vendors. However, due to CUDA's head start as well as its programmability advantage, there remain a significant number of applications written solely in CUDA. These applications cannot attempt to make use of the performance afforded by another vendor's device, without requiring lengthy and error-prone translation to OpenCL or the vendor's own proprietary framework. Further, while there is remarkable similarity between CUDA and OpenCL, the fact remains that the two languages are not semantically identical, and there are many situations in which there is no robust mapping between them. Therefore there is a demand for a robust automatic CUDA-to-OpenCL translator in addition to an improved understanding of the challenges inherent to achieving such translation.

1.1 Comparison of CUDA to OpenCL

CUDA and OpenCL share a number of conceptual and linguistic similarities as both have a basis in GPGPU acceleration. These similarities are one of the primary enabling factors behind efforts to achieve translation from one specification onto the other. For the purposes of translating a majority of applications, the most significant similarities lie within their execution and memory models.

```
dim3 block(BDIM_X, BDIM_Y);
dim3 grid(GDIM_X / block.x, GDIM_Y / block.y);
kernel<<<grid,block>>>(...);
```

(a) CUDA C

```
size_t[3] group_size = {BDIM_X, BDIM_Y, 1}
size_t[3] range_size = {GDIM_X, GDIM_Y, 1}
clEnqueueNDRangeKernel(command_queue, kernel, 3, NULL, range_size, group_size, ...);
```

(b) OpenCL

Figure 1.1: Setting CUDA/OpenCL Execution Configuration

Within CUDA, host- and device-side code share the same source scope, but device functions are annotated with the `__global__` or `__device__` attribute on the function declaration and definition. `__global__` functions may be invoked from the host and are known as *kernels* whereas `__device__` functions may only be invoked by other functions executing on the device. To invoke a kernel, one of several methods can be employed — discussed further in Section 3.1.5 — but the most popular takes the form of an expanded function invocation, specifying the kernel's *execution configuration*. Figure 1.1a provides an example of this syntax in a typical kernel invocation. This allows the programmer to specify a 1- to 3-dimensional *grid* of 1- or 2-dimensional *blocks* of *threads*, which execute the function in parallel across one or more *streaming multiprocessors* (SMs). The conceptual hierarchy of these thread groupings is demonstrated in Figure 1.2. Additionally, on *current* NVIDIA architectures, the threads are executed in lock-step in batches of 32, known as *warps*.

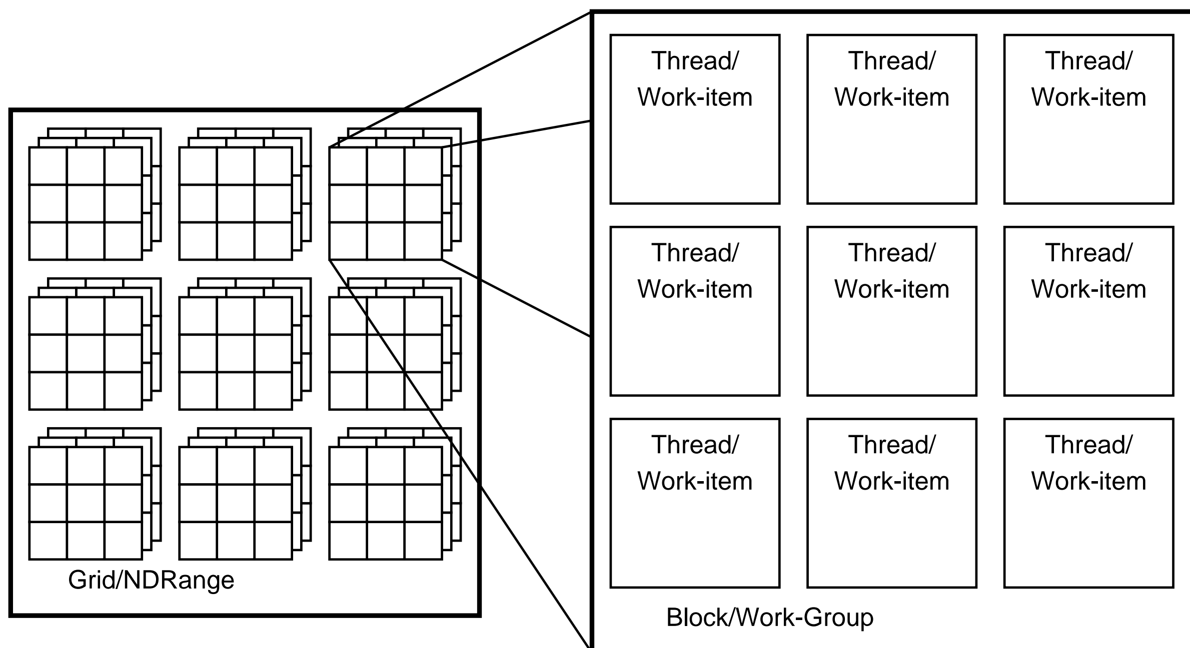


Figure 1.2: CUDA and OpenCL Execution Model
(Multiple labels signify first the CUDA terminology, followed by the OpenCL terminology)

OpenCL maintains a similar conceptual execution model with varying terminology and syntax. A key difference is that OpenCL host- and device-side code **do not** share the same source scope, as OpenCL kernel code is generally compiled just-in-time from a string. This string is either stored as a constant in the host application or read from a separate file on disk at runtime. Thus, device-side accessory functions need no explicit annotation as device code remains isolated from host, and host-invokable kernels in device code are specified via the `__kernel` attribute. OpenCL makes use of a significantly more verbose kernel invocation, discussed in Section 3.1.5, but retains a mechanism of specifying an execution configuration. This requires the programmer to set a global work size specifying the 1- to 3-dimensional *N-dimensional range* of 1- or 2-dimensional *work-groups* of *work-items*. These work-items then execute the kernel function in parallel across one or more *compute units* (CUs), but without the implicit lock-step execution of CUDA's warps, an inconsistency addressed further in Section 4.3.2. The equivalence of these organizational concepts to CUDA is demonstrated

in Figure 1.2, and an example of a typical OpenCL kernel invocation specifying the global and local work sizes is shown in Figure 1.1b.

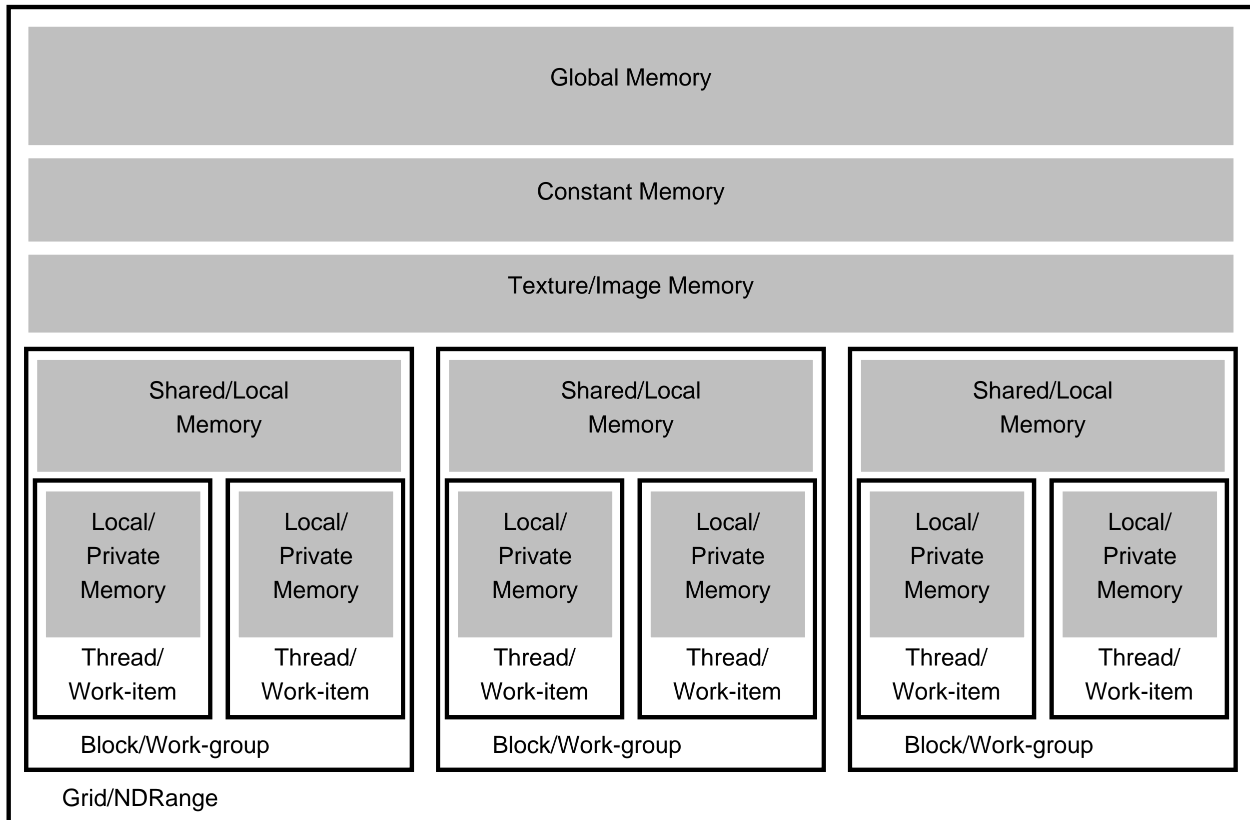


Figure 1.3: CUDA and OpenCL Memory Models
(Multiple labels signify first the CUDA terminology, followed by the OpenCL terminology

Likewise, CUDA and OpenCL's memory models have significant similarities, shown in Figure 1.3. Both have the notion of a bulk *global* memory, accessible by all threads running on the device and declared and initialized by explicit host-side calls. Additionally, both provide access to a small amount of faster, special purpose memory accessible to only those threads/work-items that are in the same block/work-group. This is known in CUDA as *shared* memory and *local* memory in OpenCL. Likewise both support another small region of *constant* memory that is set by the host or as statically defined device program scope variables and is read-only for device threads. Both maintain a concept of *registers/private*

memory that are accessible only by a single thread. Finally, due to the GPU origins of both CUDA and OpenCL, they support a region of *texture/image* memory that provides fast read and write access through special-purpose functions.

1.2 Related Works

Works most directly related to this thesis lie in the areas of source translation, CUDA, OpenCL, and general GPGPU and accelerated application development and performance, and more specifically, translation to or from CUDA/OpenCL. Historically, there have been a number of efforts to provide source translation and analysis of C and C-like languages. Over the past several years there has been an explosion in the popularity of GPGPU development, largely driven by potential performance gains and the availability of compute APIs like CUDA and OpenCL. As such there is a vast trove of literature documenting issues encountered during development of applications for GPU devices as well as addressing their performance and architecture-specific optimization methods. Further, this popularity has driven a number of attempts at providing automated translation to or from GPGPU development frameworks in order to provide access to the potential performance gains provided by the devices without requiring time-intensive manual porting of applications.

The C language, and its descendants, including C++, have undergone significant evolution since their creation, growing in functionality and expressivity based on the demands of their developer communities. Throughout the histories of these languages, various forces have created demands for automated analysis, transformation, and translation of their source code. As both CUDA and OpenCL inherit much from their C/C++ ancestors, many of the approaches used for source-level interaction could likely be applied to modern translation efforts. For example, the first implementation of a C++ compiler, CFront, was effectively

a C++ to C translator [39], which therefore might provide insights into mapping CUDA's device-side C++ onto OpenCL's device-side C99 requirement. Developers utilizing C, C++, and descendants such as CUDA and OpenCL also frequently make heavy use of the languages' separate preprocessing step in order to develop multiple variants of an application within a single body of source code, which complicates and limits source-level operations. Fortunately, there has been much work to understand [13] and address problems with source-level operations introduced by the preprocessing step. As an example, automated source code refactoring tools must account for preprocessing, particularly in the presence of conditional compilation directives [17, 37].

The primary goal in automating CUDA-to-OpenCL translation is focused on providing *functional portability* in order to ensure that codes translated from CUDA provide accurate results when running OpenCL on any vendor's device. However, realizing true *performance portability* remains the paramount goal to many OpenCL developers. Since CUDA's birth, a significant body of literature documenting various optimization strategies for NVIDIA GPUs has been developed, almost all of which apply to OpenCL codes when executed on NVIDIA devices [24, 35, 43]. Similarly, optimization techniques for various CPU architectures have also been well-studied [41, 42]. However, there remains a significant gap in the extensiveness of published studies of optimization techniques for AMD GPUs. While OpenCL is intended to provide functional portability across all these, as well as other devices, past work has shown that code that is optimized for a specific underlying architecture will not necessarily execute at similar levels of performance when executed on a dissimilar underlying architecture. For example, studies on linear algebra solvers [12], Monte Carlo simulations [40], and molecular modelling simulations [7], as well as many other problems, such as stencil computations [8] have demonstrated the need for device-specific optimizations to achieve high performance.

Given the large interest in CUDA since its development, a number of efforts have attempted to provide translation to the CUDA framework or from the CUDA framework. Additionally, the rise of OpenCL and interest in non-NVIDIA accelerator architectures has driven the development of similar translation tools providing accelerated codes portability to devices outside the scope of their original development language. In general, these translation tools take one of three forms, (1) source translation from one framework onto another, (2) translation of an intermediate representation (IR) to another, or (3) abstraction of multiple frameworks into a single high-level interface.

Source-level translation to or from CUDA has been largely driven by a desire to write data parallel applications through a framework with which one has more familiarity, while retaining the benefits of the target framework, be they performance, portability, or programmability. Attempts to translate to CUDA frequently consist of efforts to write applications in a previously existing parallel framework, such as OpenMP [23] and make use of an underlying CUDA accelerator in place of the original underlying hardware. However, as a number of applications have already been written in CUDA, and many consider it to be a developer-friendly accelerator framework, there is also interest in translation of CUDA codes to other platforms. For example, MCUDA implements a translator from CUDA to multi-threaded CPU applications [38]. There have also been efforts to automatically translate native CUDA applications to a form amenable for remote execution, such as CU2rCU [32]. Finally, there have been efforts to translate CUDA source directly to OpenCL in order to allow accelerated codes to run on devices other than NVIDIA without manual rewrites. Examples of usch efforts include CUDAtoOpenCL [27] and CU2CL [26, 25], upon which the work in this thesis is based.

Another popular approach to CUDA translation is to directly translate from CUDA's *Parallel Thread Execution* (PTX) IR onto another language or IR. The advantage of this approach

is that CUDA modules that have already been compiled to PTX need not be recompiled from source to target a differing underlying architecture. This is the approach taken by Ocelot [9, 10] and Caracal [11], which translate from PTX to SPU assembly for the Cell Broadband Engine, LLVM for CPU execution, and AMD's Compute Abstract Language (CAL) for execution on AMD GPUs, respectively. However, one downside to this approach is that a separate back-end must be implemented for each new intermediate representation one wishes to target. In contrast, translation to OpenCL allows users to instead rely on vendors to provide the necessary back-end.

A third approach for realizing portability between multiple device types or compute language specifications is to abstract multiple underlying frameworks into a single high-level interface. This is the approach taken by the Swan tool [19], which allows developers to write applications for both CUDA and OpenCL in a single high-level API, which is then executed using CUDA or OpenCL, depending on which version of `libswan` is linked. CUDACL takes a similar approach, which provides code generation for CUDA or OpenCL from an abstract C/Java API, implemented as an Eclipse plugin [20]. Other approaches make use of source annotation frameworks such as OpenACC [31], and there have been efforts to provide implementations supporting both CUDA and OpenCL [33]. While seeking a high-level representation of parallel software is a laudable goal for easing parallel development, as additional layers of abstraction are added, it becomes more difficult to access the fine-grained control of parallel architectures provided by lower-level alternatives like CUDA and OpenCL.

1.2.1 CU2CL Translator Prototype

The vast majority of the work presented in this thesis is either enabled by, or directly in support of, a prototype CUDA to OpenCL translator, known as *CU2CL*. The first imple-

mentation of this tool was created at Virginia Tech as part of a previous student's thesis [26]. As such, it is critical to address a number of its key features to provide context for the work presented here.

CU2CL was developed as a plugin to the Clang compiler framework [1], based on LLVM [22]. While a number of other frameworks were considered, Clang was settled upon due to its extensibility as well as its vibrant development community, rapid growth, and growing built-in support for compiling CUDA source code [26, 25]. Clang and LLVM provide lexing, parsing, semantic analysis, abstract syntax tree (AST) generation, rewriting, and many additional capabilities of use to a source-to-source translator.

The CU2CL plugin is implemented as an AST consumer, relying on the AST generated by Clang for identifying and recursively iterating over CUDA components of interest to be translated. However, in contrast to other efforts at source translation, modifications are not performed on the AST itself, to later be reconstituted into source code. Instead, by using Clang's Lex and Rewrite libraries, transformations are applied directly to CUDA source code, a technique known as *AST-driven, string-based rewriting* [26, 25]. This permits all non-CUDA specific structures in the original source code to be passed through unchanged, preserving original meta-content, such as commenting and formatting, therefore easing efforts to continue development on the translated source.

While the basic CU2CL prototype realizes translation of a significant portion of the CUDA specification, the vast majority of its work falls under translating either CUDA data types or CUDA API function calls. As such, a large portion of the translation makes use of common patterns that at a high level are largely agnostic to the specific API call or data type that is being converted. This high degree of abstraction allowed the first prototype to be assembled in under *2000* lines of code, while still providing decent coverage of commonly used CUDA structures and integrating a number of other notable features. Among these are support

for recursive expression rewriting as well as automatic translation of `#include` preprocessor directives [26, 25]. Several other features and mechanics of the initial prototype are discussed in greater detail in the thesis sections to which they are relevant.

1.3 Research Approach

A multi-step approach was taken to improve the viability of CU2CL as a production-ready translator suitable for the general translation of CUDA applications to OpenCL. The end-goal of the project is to promote their execution and continued development on non-CUDA-enabled platforms. As discussed in Section 1.2.1, an initial prototype providing the general framework and most frequently encountered CUDA structures was already developed as a launching point. To develop an understanding of the prototype's initial capability, an extensive study of its reliability was undertaken by attempting to translate nearly 100 CUDA applications. These applications include many from the CUDA SDK [30] and the Rodinia benchmark suite [5, 6], as well as three "large applications" in the domains of molecular modelling, molecular dynamics, and neural networks. By collecting and analyzing both the translator output as well as source code for these applications, an understanding was formed of the generalized patterns among CUDA source structures that the initial translator prototype was unable to fully handle. These patterns were then reconciled against both the CUDA and OpenCL documentation to further characterize them based on their theoretical translatability. This study also provided a more complete understanding of the relative frequency of inadequately-supported CUDA structures in order to help prioritize development efforts. This knowledge promoted expansion of the prototype's effective coverage and utility in the three following ways: (1) by improving its ability to provide at least a best-effort translation in the presence of difficult or potentially untranslatable structures, (2) by increasing the

quality and coverage of error reporting, and (3) by adding select features to the translator's functionality. After enhancement of the prototype, a repeat translation of the population of sample applications was performed to gauge the relative increase in effective coverage and reliability, in addition to collecting data on translator performance.

1.4 Contributions

The primary contributions of this thesis are:

- A characterization of CUDA source structures and programming practices that complicate manual or automatic translation of source code to OpenCL.
- Enhancement of the CU2CL prototype translator's reliability and verbosity by integration of knowledge gained by source code characterization and CU2CL-specific error analysis.
- A demonstration of the improved translator's reliability, performance, and CUDA source code coverage.
- A brief study demonstrating that performance of automatically-translated OpenCL applications as being comparable to the original CUDA when executed on the same NVIDIA hardware.

1.5 Thesis Organization

The organization of this thesis is as follows. First, this provides background on the area and problem of study and previous works related to CUDA and OpenCL translation. Fur-

ther, it details the research approach taken, as well as the primary contributions of the work. Chapter 2 presents and discusses a study of CUDA applications through the lens of the original CU2CL translator prototype, in order to identify commonly-occurring structures in CUDA source that provided practical limitations to automatic translation. It next presents a study of the effective increase in translator coverage and reliability, analysis of translator performance, and analysis of the performance of automatically translated applications. Chapter 3 then provides a detailed discussion of enhancements added to the prototype translator, which were designed to address several of the issues identified during profiling. Chapter 4 then transitions to a theoretical discussion on the viability of providing automatic translation for several of the structures identified during profiling, detailing numerous opportunities for future work. Finally, Chapter 5 presents a summary of the work alongside primary conclusions. An additional appendix is then provided, consisting of the data tables used to construct the graphs in Chapter 2.

Chapter 2

Characterization of Translator Capability

To work towards a more complete understanding of the practical difficulties in performing source-level translation of CUDA to OpenCL, an analysis of a large body of CUDA applications was performed. This analysis primarily consisted of a search for specific CUDA constructs and programming practices that present difficulties to the human or machine translator. The first phase of the analysis was to manually inspect CUDA source code for syntax and behaviors that could not be represented in OpenCL. The second phase utilized the CU2CL prototype translator to attempt translation of a subset of sample applications. This provided volumes of information on the diversity and relative frequencies of difficult CUDA source structures and programming practices. This information was then used to form generalizations of the structures, reason about their theoretical translatability (discussed in Chapter 4), and prioritize development to enhance the effective reach of the translator prototype (discussed in Chapter 3). After enhancement, a third phase of analysis using the upgraded prototype was conducted to provide more direct observation of the gains in the

tool's utility. This chapter presents an analysis of the primary metrics used to analyze the translator's capability: translator performance, translator reliability, translator coverage, and the performance of translated applications.

2.1 Test Applications

The population of sample CUDA applications used for analysis came from a number of sources, including both industrial/commercial affiliates and academics. The mix of application sources was chosen to ensure a high diversity of programming styles, coverage of CUDA language features, and range of application areas on complexities. Application areas present in the population include molecular modelling, bioinformatics, neural networks, finance, and several others.

2.1.1 CUDA SDK

By far the largest singular source of applications is a selection of 79 samples from version 3.2 of the CUDA SDK [30]. These samples are provided to demonstrate the use of a range of CUDA functionalities and programming styles. Combined, they demonstrate a multitude of difficult-to-translate features, including precompiled CUDA libraries, OpenGL interoperability, device-side C++, texture, shared, and constant memories, and special purpose on-device functions. Further, they range in complexity from single-file math demonstrations to significantly larger-scale particle simulations, demonstrating GPGPU approaches to a range of application areas, such as linear algebra, finance, N-Body simulation, image processing, and quasirandom number generation.

2.1.2 Rodinia

The next most significant source of samples is the 17 CUDA applications that made up version 2.0.1 of the Rodinia Benchmark suite [5, 6], which have been selected as representative of small-scale academic application development. They are developed to be largely standalone, and possess far fewer of the "SDK-isms" frequently present in the applications taken from the CUDA SDK, such as the use of `cutils` and `shrUtils`. These provide additional convenience wrappers and functions that are not part of the canonical CUDA specification. Additionally, they have been developed by a number of different authors of differing levels of CUDA/C experience, which increases the overall variance in programming style in the sample population.

2.1.3 Other Large Applications

In addition to the 96 "small" applications with source lines of code (SLOC) numbering in the hundreds to few thousands, the analysis includes three "large" CUDA applications to gauge the tool's effectiveness on more production-ready codes. All of these applications consist of several thousand lines of CUDA and C/C++ code, and together demonstrate use of several features of the C/C++/CUDA C languages that complicate the translation process.

GEM

The first large-scale application profiled for CUDA constructs was a molecular modelling application for computing the electrostatic surface potential of biomolecules, known as GEM [2, 14]. While analysis of its translation time as well as code coverage have been previously published [25], it is included here for completeness as well as its potential to offer insights during the code examination phases of profiling. Additionally, the methodology

used for determining code coverage differs slightly from that used in previous works in its improved analysis of device-side code coverage via NVIDIA's OpenCL kernel compiler.

Fen Zi

The second large-scale application profiled is a molecular dynamics simulation for modelling membrane-bound protein receptors, known as Fen Zi [3, 15, 16]. Fen Zi implements a version of the Particle Mesh Ewald method specifically designed for GPU execution, switching from a traditional charge-centric algorithm to a more GPU-amenable lattice-centric form. A particular facet of the method that is of direct interest to this work is its dependence on a Fast Fourier Transform (FFT), internally utilizing the closed-source CUFFT library of CUDA-accelerated FFT kernels. The use of this library represents a distinct challenge for a source-level translator, reducing effective translator coverage of the application's source code, an issue discussed further in Chapter 4. Fen Zi was developed by the Global Computing Lab at the University of Delaware.

IZ PS

The third production CUDA application profiled was a spiking neural network simulation implementing the Parker-Sochacki numerical integration method applied to the Izhikevich neuron model [44]. Of particular interest to translation efforts is their use of CUDA's shared and texture memory spaces for improved performance, as well as use of the CUDA Data Parallel Primitives (CUDPP) Library. Additionally, the source code makes heavy use of often-nested preprocessor macros, which complicate translation significantly (discussed further in Section 4.5.1).

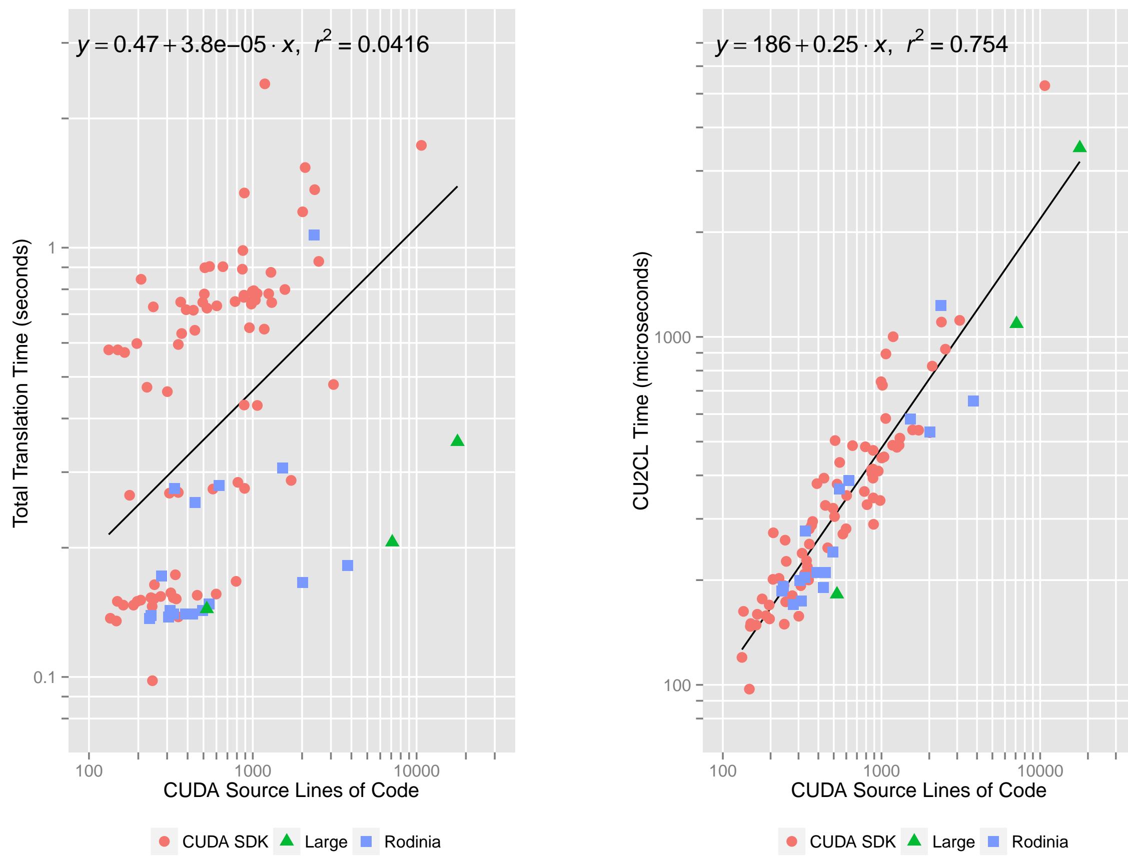
2.2 Test Environment

For performance analysis of both the translator prototype as well as automatically-translated applications, a consistent test platform was used. This platform consisted of a commodity desktop running 64-bit Ubuntu 12.04 with Linux kernel `3.2.0-35-generic`. The CPU used was an AMD Phenom II X6 (six-cores at 3.2 GHz) equipped with 16 GB RAM, and the GPU used was a NVIDIA GeForce GTX 480 running NVIDIA driver version 310.32 with CUDA Runtime version 5.0.

2.3 Translator Performance

The main emphasis of a translation effort is likely to translate an existing application from CUDA to OpenCL, and then continue development in OpenCL. However, it is likely that some developers may opt to continue their primary development in CUDA, either for reasons of personal familiarity with the language, or due to the generally-accepted increased ease of developing that CUDA provides. Therefore, a reasonable emphasis is placed on the translator's performance to support users who opt for frequent retranslations from their canonical CUDA source as part of building the application. For one-off translations a rapid translation time is likely unnecessary, as the speed of automatic translation will almost certainly surpass that of manual efforts. Fortunately, as previously demonstrated on a small number of codes [25, 26] the average time of translation is low, usually less than a second. However, as a number of features were added to the translator and a large and diverse population of sample applications was available, an expanded analysis of CU2CL's performance was conducted.

Tables A.1, A.2, A.3, and A.4 provide translator performance measurements on samples from



(a) Total translation time of sample applications with respect to application CUDA SLOCs.

(b) CU2CL portion of sample application translation time with respect to application CUDA SLOCs.

Figure 2.1: Translator Performance vs. Source Lines of Code

the Rodinia benchmark suite [5, 6], CUDA SDK [30], and large applications. Figures 2.1a and 2.1b show the full time of translation and the subset of time taken by the CU2CL portion of the translator, respectively, with respect to the number of CUDA SLOCs in the application. Reported times represent the average time of 10 translations using the most current development build of CU2CL, modified December 9, 2012, based on the Clang/LLVM 3.2 development tree, revision 159674. As can be seen in Figure 2.1a, although even the largest applications can be translated in a few seconds or less, there is no clear correlation between total time of translation and SLOCs. This can be explained by variations in the

time required by Clang to generate an AST, potentially caused by the presence of template functions and deeply-nested `include`d header files. In contrast, Figure 2.1b demonstrates a relatively strong correlation between SLOCs and the time taken by the CU2CL portion of translation. This is attributed to the one-time walk of the AST that CU2CL performs during translation; as the number of SLOCs grows, so does the AST.

2.4 Translator Reliability

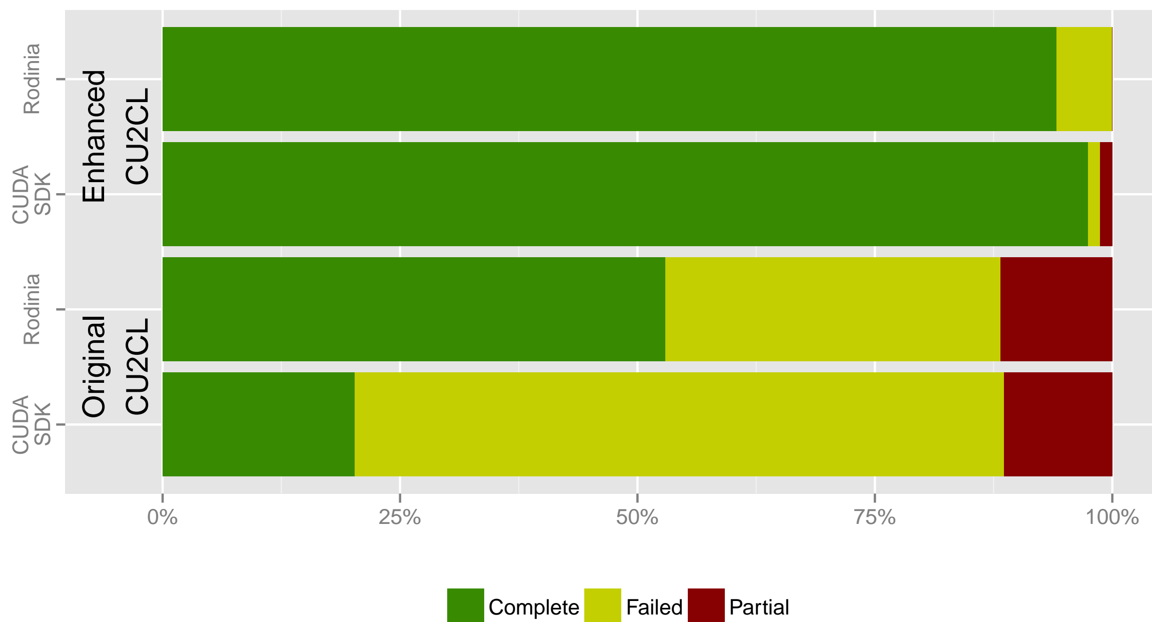


Figure 2.2: Reliability of the CU2CL Translator before and after enhancement when translating CUDA SDK and Rodinia sample applications.

Profiling a large selection of diverse applications provided a demonstration of the first prototype's effective limitations. Throughout testing it was discovered that a rather large portion of applications produced either only a partial translation or resulted in unidentified failures within the translator itself, producing no OpenCL output whatsoever. Therefore, it

was critical that the underlying causes of these failures were identified, in order to direct improvement of the translator's robustness. Figure 2.2 shows a visual comparison of the current prototype's reliability to the original's when translating samples from the CUDA SDK and Rodinia. The colors represent CU2CL's ability to provide at least a partial translation for every CUDA source file in an application. The green "Complete" bars signify that OpenCL is output for every CUDA source file in the application, the yellow "Partial" bars signify that OpenCL is output for at least one CUDA source file from the application, but not all, and the red "Failed" bars mean the translator was unable to produce OpenCL output for any CUDA source files in the application. As shown, the translator's reliability has dramatically improved, translating an additional 77% and 41% of CUDA SDK and Rodinia samples, respectively. Tables A.5, A.6, and A.7 provide more depth on the translation status of individual applications both before and after enhancement of the prototype. Further, for each partial or failed translation, the tables show additions to the translator, discussed further in Chapter 3, that removed the underlying failures and enabled the production of a more complete translation of the application. However, despite the numerous additions designed to improve robustness, two applications from the CUDA SDK samples — Interval and MonteCarloCURAND — still fail to provide complete translation, due to a limitation of Clang's ability to simultaneously parse C++0x and CUDA syntax elements. Additionally, one Rodinia sample, MummerGPU, contained an implicit cast that was invalid on 64-bit platforms, and prevented Clang from reaching the CU2CL plugin to perform translation.

2.5 Translator Coverage

Profiling tests on the sample applications identified many instances of unsupported, incomplete, or malformed translations. While this is not unexpected for an academic prototype

undergoing active development, it is critical to develop an understanding of the translator's scope, both to be honest in advertisement of its capability, as well as to drive future development efforts. The methodology used to analyze application coverage consisted of the following stages:

1. Perform manual preprocessing of the CUDA application to remove "SDK-isms" such as the usage of CUDA's `cutils` or `shrUtils`. These lines changed are not counted as part of characterization score as they apply to both the original CUDA as well as the OpenCL, and are only to remove non-canonical convenience wrappers, which are not intended to be translated. Otherwise they do not result in a net change of application semantics. The most frequent changes in this category are replacement of `shrLog` calls with standard `printf` calls, as well as removal of the `cutilSafeCall` wrapper function.

2. Translate all source files of the application that require either the `nvcc` compiler or inclusion of one of the standard CUDA header files (`cuda.h`, `cuda_runtime_api.h`, or `cuda_runtime.h`, for example) using default compiler definition arguments required for GPU execution.

3. Attempt compilation of both host and device OpenCL code using the same compiler definitions used during translation. This expands on earlier works that only characterized changes required in the host-side code, neglecting to address the potential for device-side code to also require post-translational repair.

4. Then either

    (a) Correct all errors identified during attempted compilation until compiled code runs to completion giving functionally-equivalent output to the original CUDA

when run on the same NVIDIA hardware[1]. This was the approach taken to analyze coverage on the five Rodinia samples, seven CUDA SDK samples, and GEM.

OR

(b) Selectively flag erroneous source code in a manner that preserves compilation semantics[2]. This ensures that lines that would need to be manually modified are accounted for without spending inordinate amounts of time performing one-off translations of simple applications. This approach was used to analyze coverage of the other two large applications, Fen Zi and IZ PS.

5. After code compiles without errors or translation-induced warnings[3] perform a minimal `diff` of the modified OpenCL with the original output of the translator, counting the *number of lines changed*, excluding whitespace.

6. Count the number of SLOCs in the original CUDA files before translation, subtract the number obtained in step 5 from this value, and then divide that by the CUDA SLOCs to count the percentage of lines *correctly translated*.

As the process of analyzing code coverage requires investment of significant developer time to actively perform extensive modifications on poorly-translated source files, only a subset of the applications have been profiled. While sample applications that would require substantive changes to execution patterns have been omitted from coverage analysis — such as those

[1] "Functionally equivalent output", means production of the exact data set produced by the original CUDA, within acceptable range of floating point round-off variance.

[2] In this case flagging implies the manual removal and replacement of source structures that cause failures during compliation with NVIDIA's OpenCL kernel compiler. In general this implies commenting of untranslated CUDA expressions, and the replacement of select function values and initializations with appropriate dummy values.

[3] For example, warnings about dangling attributes left behind in output OpenCL code due to incomplete removal during the translation process.

Source	Application	CUDA Lines	Lines Changed	Percent Automatically Translated
CUDA SDK	asyncAPI	135	5	96.3
	bandwidthTest	891	5	98.9
	BlackScholes	347	14	96.0
	fastWalshTransform	327	30	90.8
	matrixMul	351	9	97.4
	scalarProd	251	18	92.8
	vectorAdd	147	0	100.0
Rodinia	Back Propagation	313	24	92.3
	Breadth-First Search	306	35	88.6
	Gaussian	390	26	93.3
	Hotspot	328	2	99.4
	Needleman-Wunsch	430	3	99.3
[15, 16, 3]	Fen Zi	17768	1786	89.9
[2]	GEM	524	15	97.1
[44]	IZ PS	8402	166	98.0

Table 2.1: Coverage of CU2CL Translation

with device-side template usage, discussed further in Chapters 3 and 4 — a number include frequent use of CUDA structures that the translator currently has no support for, such as textures. Table 2.1 demonstrates that for even the largest applications, CU2CL provides a relatively high degree of coverage, significantly reducing the percentage of applications which remains to be translated by hand.

2.6 Translated Application Performance

Of high concern for those considering translating their code is ensuring they will not pay significant performance penalties for moving from CUDA to OpenCL. While earlier work showed that NVIDIA's earliest implementations of OpenCL occasionally provided performance well below that of CUDA [25, 26] their modern implementations have largely removed this con-

cern. The performance of applications that were repaired to give functionally equivalent as defined in Section 2.5 was compared to the performance of the original CUDA to show that performance is preserved when translated codes are executed on the same NVIDIA hardware. Tests were run on the platform detailed in Section 2.2. Whole application performance was measured using the Unix `time` command and the average of ten trials is reported. As can be seen in Table 2.2, in all applications there are only minor differences between the OpenCL and CUDA run times, with many examples of the automatically-translated OpenCL out-performing the original CUDA. Given that both source codes are near-identical other than API calls, it is unclear what underlies these performance differences. However, for fairness this analysis is restricted to solely executing automatically-translated OpenCL on NVIDIA devices. Due to different underlying devices requiring different optimization strategies, it is not necessarily true that functional portability via OpenCL will guarantee a similar level of performance on devices from other vendors.

Application	**CUDA Runtime (s)**	**OpenCL Runtime (s)**	**Percent Change**
asyncAPI	0.58	0.55	-6.6
bandwidthTest	0.94	0.86	-8.5
BlackScholes	1.98	1.75	-11.5
FastWalshTransform	2.00	2.03	+1.3
matrixMul	0.47	0.47	-1.6
scalarProd	0.51	0.51	-0.2
vectorAdd	0.47	0.46	-0.8
Backprop	0.87	0.87	+0.4
BFS	2.09	2.17	+4.1
Gaussian	0.48	0.46	-2.8
Hotspot	0.81	0.79	-1.9
Needleman-Wunsch	0.57	0.52	-9.2
GEM	0.51	0.49	-2.9

Table 2.2: Run Times of CUDA Applications and OpenCL Ports on an NVIDIA GTX 480

Chapter 3

Improved Robustness

One of the primary contributions of this thesis is an effort to move the CU2CL source to source translator from academic prototype towards being a robust production tool. Before fully-automated CUDA-to-OpenCL translation can be realized, there remains much work to be done, both to extend coverage of the translator and to expand the capabilities of OpenCL. A substantial effort has been given to improving the translator prototype's robustness in hopes that it might soon be released to the community for continued expansion. Several additional functionalities have been integrated into the enhanced prototype since the initial versions were published [26, 25], which are intended to address a number of issues identified during the first profiling runs discussed in Chapter 2. These additions fall loosely into three categories: Features Added, Mistranslations Repaired, and Bugs Quarantined. This chapter first discusses those features that are newly added, then repairs made to certain infrequently-used code paths for select translations, and finally the methods used to prevent the translator from failing when encountering irregularities in the incoming source code.

3.1 Features Added

As part of the effort to expand the translator, support was added for a number of new features, either to improve effective coverage of CUDA structures, or to improve the utility to the end user of the translator and translated source code. In some cases addition of functionality simply required filling in a scaffold already present in the early prototype, but the majority required addition of new code. Additionally, in a few cases implementation of a new feature is incomplete, but a significant portion of the scaffold has been constructed from which to finalize support at a later date.

3.1.1 Inline Error Reporting

A key update to the usefulness of CU2CL is a more robust error and warning reporting mechanism. In the original prototype, translator output messages were dumped directly to a console error stream, and were rather simplistic, containing only a brief message hinting at the underlying problem. While this has some utility to a CU2CL developer who has access to the source code, and can readily identify the CUDA source structures causing the error by examining the Clang structures used in that region of CU2CL, it has rather low utility for the end user. Additionally, the vast majority of encountered errors were accompanied by no explicit message whatsoever. Therefore, to more significantly assist the end user in resolving the issue, a generic error reporting facility was added. This facility allows highly-specific error notifications to be emitted to the standard error stream alongside pointers into the original source, similar to a traditional compiler. However, as the output OpenCL source code will also likely need modification, this idea is expanded on by utilizing Clang's rewriting functionality to emit similar error notifications directly into the output OpenCL source files as comments, including standardized tags that are easily searched by the developer.

To this end a standardized interface was created for all translation-time messages to be sent to the user. Each such message takes a "severity level" that determines the output tag prepended to the message in the output stream. Currently, the tool uses four such levels. The first, "CU2CL Error" is for exotic source code that the translator doesn't know how to handle. The second, "CU2CL Untranslated" is for CUDA structures that are identified but not actively translated and are thus emitted unmodified into the output OpenCL source. The third, "CU2CL Unsupported" is for similar cases in which CU2CL has identified a CUDA structure that **can** be translated but is not currently implemented. Finally, "CU2CL Notes" and "CU2CL Warnings" are akin to standard compiler warnings, in that they advise the user that the translator has had to make assumptions or perform some non-standard translation. These notes and warnings are generally emitted in areas where there is high confidence in translation accuracy, but it was achieved via atypical methods, as a courtesy to the end user to draw attention that there could be a flaw. Currently, this error interface is implemented as a hand-built feature of the CU2CL plugin to Clang; implementing similar handling through Clang's unified diagnostic subsystem remains for future work.

Note: due to an implicit ordering constraint in Clang's Rewrite library, all comments destined to be inserted directly into output source code are necessarily buffered until the end of translation, while their counterparts destined for the error stream are emitted at detection time.

3.1.2 Update to Clang 3.2

As an effort to simplify the installation and development process for CU2CL, as well as to ensure forward compatibility, the entire CU2CL plugin source was updated to make use of the LLVM/Clang 3.2 API, specifically revision 159674. This required a number of minor source

changes throughout the plugin to adapt to a slightly modified API and tweaked behaviors of a number of functions. Of particular relevance were changes to the API components allowing access to *SourceLocations* referring to macro pointers. Additionally, this update removed the need to manually apply patches to the LLVM and Clang source trees, as their support for CUDA parsing within the main development branches has significantly improved since version 2.9 upon which CU2CL was originally based. This update also contributed to the translation of a large number of the sample applications that previously encountered a known error with Clang 2.9's support for the 4.6 version of GNU Standard C++ Library [34]. In this revision, there remains an issue with support for simultaneous parsing of C++0x and CUDA.

3.1.3 Device-side Builtin Math Functions

As noted elsewhere, CUDA and OpenCL share a remarkable similarity in the capabilities of their device languages. One area in which there is a near perfect overlap is builtin support for many commonly used math functions. However, there is a slight discrepancy in the naming convention used by these functions, in particular their single-precision floating point versions. In CUDA these all take the form "`sin`*f*" with the "*f*" denoting that the single-precision version should be used. However, in OpenCL all variants of these math functions, including vector versions, make use of the same entry-point function, requiring only that all operands and return values share the same, possibly explicitly-cast, supported type. Therefore, several versions of the CUDA kernel math builtin functions must be converted in order to provide valid OpenCL kernel code. While the initial CU2CL prototype only provided a sparse scaffold demonstrating the translation of a select few of these functions — `fabsf`, `sqrtf`, `__expf`, `__logf`, `__log2f`, and `__powf`— support has been expanded to include all such functions for which a direct OpenCL equivalent exists. However, translation of the

versions of these functions that provide explicit control of internal rounding modes are not yet supported and remain for future work.

3.1.4 Partial Support for `cudaSetDevice`

One of the most frequently observed "optional" CUDA calls witnessed throughout profiling was `cudaSetDevice`, which takes a simple integer argument specifying a CUDA-capable device in a system, and requests that the CUDA context for the program be switched to refer to that device. It was observed that in many cases, the call simply requests a default device, which is redundant as CUDA already initializes a default at the occurrence of the first CUDA call in a program. However, in a few select cases the call is used to iterate over all or a subset of the devices present in a system. Therefore for compatibility, an equivalent method for explicitly setting the OpenCL context to refer to a specific device must be implemented, without interfering with the more common automatic initialization of a device.

To avoid interfering with the pre-existing code that implements the automatic initialization behavior required of CUDA codes which do not make use of the `cudaSetDevice` call, a separate utility function was developed. This handler, `__cu2clSetDevice`, assumes an OpenCL context has already been created for some device. It begins by executing the necessary OpenCL calls to release the context associated with the default device[1]. However, one key difference between how CUDA and OpenCL refer to devices is that CUDA uses simple integers, whereas OpenCL uses the `cl_device` opaque type. Therefore the handler also makes use of a helper method that, **if and only if** `cudaSetDevice` is detected, performs a one-time scan of all OpenCL devices across all OpenCL platforms present in the system, and provides an array of the devices that can be dereferenced by a simple integer. This array is then

[1]This behavior is inconsistent with CUDA, but is implemented as a simple precursor to later be replaced by a proper mechanism for preserving and switching between multiple active contexts.

used by the `__cu2clSetDevice` method to initialize a context for the device specified by the integer argument supplied[2].

However, currently the handler lacks support for recompiling the OpenCL kernels for the new context, which is known to result in silent errors when attempting to execute said kernels after a translated `cudaSetDevice`. Addition of this functionality should not require significant development effort as code already present in CU2CL for iterating over all kernels for compilation could likely be repurposed with slight modification. In addition, it was observed that `cudaSetDevice` calls are usually paired with a matching `cudaThreadExit` call, which, when translated, results in attempts to release the OpenCL context multiple times, causing segmentation faults in the translated OpenCL application. A mechanism will need to be devised for detecting such paired calls, in order to either inhibit the addition of a redundant `clReleaseContext` call associated with the default automatically-initialized context or provide another adequate handler in the form of a `__cu2clThreadExit` method.

3.1.5 Partial Support for Literal Parameter Values to Kernels

The semantics of invoking a parallel device kernel differ significantly between CUDA and OpenCL. While CUDA provides three such mechanisms, shown in Figure 3.1, the first (3.1a), in the form of an annotated function call is by far the most common. OpenCL only provides one such mechanism that is similar to both lower-level CUDA invocations in specifying parameters using pass-by-reference semantics (Figures 3.1b and 3.1c). However, this conflicts with CUDA C's high level invocation, which uses pass-by-value semantics for parameters. The initial version of CU2CL easily handles any case in which the passed parameter is simply a referenceable variable, by prepending the '&' reference operator to the variable name.

[2]This is not guaranteed to provide the same device as the same integer in a CUDA program, but relying on OpenCL's functional portability, will provide a compatible device without otherwise breaking application code.

```
dim3 block(BDIM_X, BDIM_Y);
dim3 grid(GDIM_X / block.x, GDIM_Y / block.y);
kernel<<<grid,block>>>(in1, in2, out);
```

(a) CUDA C

```
dim3 block(BDIM_X, BDIM_Y);
dim3 grid(GDIM_X / block.x, GDIM_Y / block.y);
cudaConfigureCall(grid, block, 0, 0);
cudaSetupArgument(in1, 0);
cudaSetupArgument(in2, 4);
cudaSetupArgument(out, 8);
cudaLaunch("kernel");
```

(b) CUDA Runtime API

```
cuFuncSetBlockShape(kernel, BDIM_X, BDIM_Y, 1);
cuParamSeti(kernel, 0, in1);
cuParamSeti(kernel, 4, in2);
cuParamSeti(kernel, 8, out);
cuParamSetSize(kernel, 12);
cuLaunchGrid(kernel,GDIM_X/BDIM_X, GDIM_Y/BDIM_Y);
```

(c) CUDA Driver API

Figure 3.1: Comparison of CUDA Kernel Launch Syntaxes

However, it did not provide any special handling for those cases in which the parameter being passed by value could not be reduced to a single memory address. For example any numerical or `#defined` constant would result in a malformed reference expression, as would the result of other complex expressions, such as simple math and the substitution of function-like macros. Figure 3.2a demonstrates a kernel call with several literal parameters, and Figure 3.2b demonstrates how they were handled by the original prototype.

Fortunately, a simple solution to these cases is rather easily achieved, by detecting such expressions, storing the result in a locally-scoped temporary variable, and then providing a reference to said temporary variable to the necessary `clSetKernelArg` call, in place of the malformed expression reference. This functionality is successfully implemented for all the above-mentioned cases, save that of function-like macros. As CU2CL examines the source code at the AST level, it is limited in its ability to correctly rewrite macros. It currently successfully uses macro names, but does not yet achieve reliable identification of

```
#define MIN(a,b) ((a < b) ? a : b)
#define PI_F 3.14f
. . .
float foo1 = 0.0f, foo2 =1.0f;
dim3 block(BDIM_X, BDIM_Y);
dim3 grid(GDIM_X / block.x, GDIM_Y / block.y);
kernel<<<grid,block>>>(in1, in2, 256, foo1 * 2, PI_F, MIN(foo1, foo2));
```

(a) CUDA C

```
#define MIN(a,b) ((a < b) ? a : b)
#define PI_F 3.14f
. . .
size_t block[3] = {BDIM_X, BDIM_Y, 1};
size_t grid[3] ={GDIM_X / block.x, GDIM_Y / block.y, 1};
clSetKernelArg(__cu2cl_Kernel_kernel, 0, sizeof(cl_mem), &in1);
clSetKernelArg(__cu2cl_Kernel_kernel, 1, sizeof(cl_mem), &in2);
clSetKernelArg(__cu2cl_Kernel_kernel, 2, sizeof(int), &256);
clSetKernelArg(__cu2cl_Kernel_kernel, 3, sizeof(float), &foo1 * 2);
clSetKernelArg(__cu2cl_Kernel_kernel, 4, sizeof(float), &PI_F);
clSetKernelArg(__cu2cl_Kernel_kernel, 5, sizeof(float), &MIN);
localWorkSize[0] = block[0];
localWorkSize[1] = block[1];
localWorkSize[2] = block[2];
globalWorkSize[0] = grid[0]*localWorkSize[0];
globalWorkSize[1] = grid[1]*localWorkSize[1];
globalWorkSize[2] = grid[2]*localWorkSize[2];
clEnqueueNDRangeKernel(__cu2cl_CommandQueue, __cu2cl_Kernel_kernel, 3, NULL,
        globalWorkSize, localWorkSize, 0, NULL, NULL);
```

(b) Original CU2CL Prototype

```
#define MIN(a,b) ((a < b) ? a : b)
#define PI_F 3.14f
. . .
size_t block[3] = {BDIM_X, BDIM_Y, 1};
size_t grid[3] ={GDIM_X / block.x, GDIM_Y / block.y, 1};
clSetKernelArg(__cu2cl_Kernel_kernel, 0, sizeof(cl_mem), &in1);
clSetKernelArg(__cu2cl_Kernel_kernel, 1, sizeof(cl_mem), &in2);
int __cu2cl_Kernel_kernel_arg_2 = 256;
clSetKernelArg(__cu2cl_Kernel_kernel, 2, sizeof(int), &__cu2cl_Kernel_kernel_arg_2);
float __cu2cl_Kernel_kernel_arg_3 = foo1 * 2;
clSetKernelArg(__cu2cl_Kernel_kernel, 2, sizeof(float), &__cu2cl_Kernel_kernel_arg_3);
float __cu2cl_Kernel_kernel_arg_4 = PI_F;
clSetKernelArg(__cu2cl_Kernel_kernel, 2, sizeof(float), &__cu2cl_Kernel_kernel_arg_4);
float __cu2cl_Kernel_kernel_arg_5 = MIN;
clSetKernelArg(__cu2cl_Kernel_kernel, 2, sizeof(float), &__cu2cl_Kernel_kernel_arg_5);
localWorkSize[0] = block[0];
. . .
globalWorkSize[2] = grid[2]*localWorkSize[2];
clEnqueueNDRangeKernel(__cu2cl_CommandQueue, __cu2cl_Kernel_kernel, 3, NULL,
        globalWorkSize, localWorkSize, 0, NULL, NULL);
```

(c) Current CU2CL Prototype

Figure 3.2: Comparison of Literal Parameter Handling

the parenthetical portion of a function-like macro instantiation, which remains for future work. This is demonstrated in Figure 3.2c by the lack of the parenthetical portion of the `MIN` macro — `(foo1, foo2)` — during the assignment of the temporary variable for argument 5.

3.1.6 Scaffolding for Struct Alignment Attribute Handling

```
struct __align__(8) foo
{
        int f1;
        float f2;
}
```

(a) CUDA

```
struct __attribute__((aligned(8))) foo
{
        int f1;
        float f2;
}
```

(b) OpenCL

Figure 3.3: Comparison of CUDA/OpenCL Alignment Attributes

In either CUDA or OpenCL, when transferring memory that uses structural organization more complicated than simple contiguous arrays between host and device, it often becomes necessary to explicitly set the alignment of these structs, in order to ensure memory offsets are interpreted identically on both host and device. Both languages provide an attribute that can be applied either to entire struct declarations or the members themselves to request the memory be aligned to a certain byte-width. However, there exists a slight difference between the user-facing version of the attribute in both languages. OpenCL uses the standard `__attribute__((aligned(`*n*`)))` form whereas CUDA uses a macro wrapper for this statement to allow the simpler `__align__(`*n*`)` form. While a translator could simply copy the simplifying macro from the CUDA headers, this would result in a departure from "canonical" OpenCL. Therefore it would be preferable to properly translate the simplified CUDA form into the expanded OpenCL form.

This appears to be a relatively simple task, simply extracting the n alignment expression from the attribute and rewrapping it. However, as CU2CL operates on the AST generated

by Clang, access to such preprocessor directives is limited, as discussed elsewhere. Fortunately, as explicit alignments are integral to compilation, information from them is retained. However, for alignments that are not dependant on a class or function template, only the precise alignment in bits is preserved, not the full expression. As an interim, one could simply extract this bit value, convert it to a byte-width, and use it in the translated output, but this ignores situations in which this value is intended to be non-constant, such as when the n expression is conditionally `#define`d. Therefore fully replicating the original expression, on the chance that it might contain such a macro is critical. The scaffold necessary to locate such alignment attributes was developed, but implementation of the parsing behavior required for extracting the relevant unpreprocessed expression for non-dependent alignments remains as future work.

3.1.7 Re-enabling Timing

The original CU2CL prototypes did not contain explicit timing code, as performance was measured for the entire time of translation. However, in efforts to more completely profile the translation time taken by both the Clang driver and the CU2CL plugin, the original authors created a branch that integrated timers for the CU2CL portion of the translation time. These timers measure the time required to walk the AST, perform all rewrites, and write output to disk. Thus, by also timing the duration of the entire translation process, they were able to infer the relative portion taken by Clang to perform preprocessing, parsing, and AST generation, based on the time CU2CL took to perform the actual translation. As a minor addition, this fork has been integrated into the CU2CL trunk, to ensure it is readily available to future users and developers. As this modification is largely a remnant of prior development, it is included here only for completeness and documentation purposes.

3.2 Mistranslations Repaired

Atypical use of source components is a potential pitfall that is almost certain to appear during efforts to automate source-to-source translation. These atypical variations manage to either fall through the cracks of translation entirely due to being unrecognised, or result in malformed structures after translation due to being improperly handled. CUDA-to-OpenCL translation is no different, in part because of the sheer number of ways of accomplishing certain behaviors provided by CUDA. Consequently, as the initial CU2CL prototype made an effort to achieve translation of only the most commonly-encountered structures, along with providing a scaffold for implementation of less frequent variants, a number of corner cases were identified in which CU2CL's basic assumptions resulted in malformed or incomplete translations. While there undoubtedly remain many more that have yet to be encountered, handling has been repaired for a number of those encountered during profiling, particularly in cases where a mistranslation resulted in a loss of information, potentially exacerbating the difficulty of a manual repair.

3.2.1 Support for Device Buffers as Members of Host-side Structures

While OpenCL does not support C++ constructs on the device side, there are no formal constraints on using OpenCL seamlessly with host-side C++. As previously mentioned, CUDA supports C++ on both host and device. As such several cases were observed where application developers found it most prudent to create a host-side C++ class that contains one or more device-side buffers as members. Additionally, the related technique in which host-side C structs are used to wrap one or more device side buffers appeared in several applications as well. The initial CU2CL prototype correctly supported conversion of `cudaMalloc` calls

```
typedef struct buff
{
    float * d_buff;
    int buff_size;
} buff;

buff foo;
...
cudaMalloc(
    (void **)&foo.d_buff,
    foo.buff_size
        * sizeof(float));
```

(a) CUDA

```
typedef struct buff
{
    float * d_buff;
    int buff_size;
} buff;

cl_mem foo;
...
*(void **)&foo.d_buff =
    clCreateBuffer(
    __cu2cl_Context,
    CL_MEM_READ_WRITE,
    foo.buff_size
        * sizeof(float),
    NULL, NULL);
```

(b) Initial CU2CL Prototype

```
typedef struct buff
{
    cl_mem d_buff;
    int buff_size;
} buff;

buff foo;
...
*(void **)&foo.d_buff =
    clCreateBuffer(
    __cu2cl_Context,
    CL_MEM_READ_WRITE,
    foo.buff_size
        * sizeof(float),
    NULL, NULL);
```

(c) Current CU2CL Prototype

Figure 3.4: Device Buffers as Struct Members

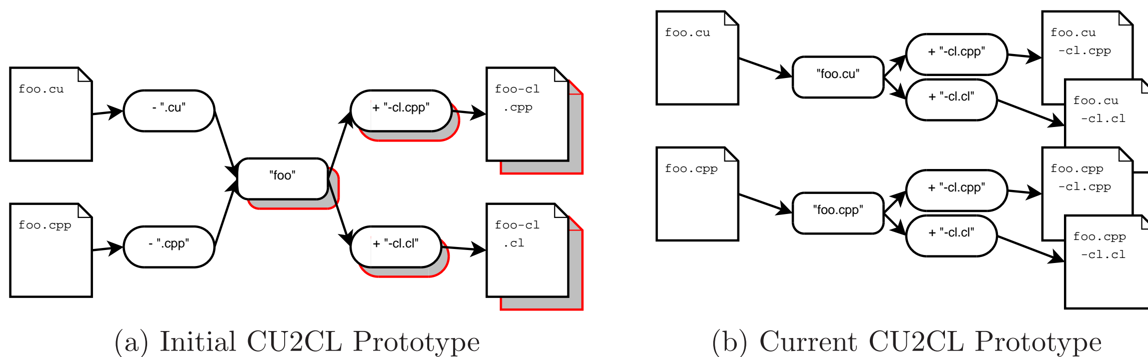
that referred to a device buffer declared as a singleton (either as a program global variable, a local scope variable, or a function parameter). However, when the buffer was a structure member, it would incorrectly convert the struct variable's declaration to a `cl_mem` rather than the internal struct member declaration. This behavior, as well as the corrected translation, are shown in Figure 3.4. CU2CL's handling of the *cudaMalloc* call has been slightly modified to appropriately handle these instances. However, no special protections have been implemented to ensure that the wrapping structure is not itself a device buffer, which is not currently supported by OpenCL. Implementing such checks remains for future work.

3.2.2 Handling of `cudaMemcpyToSymbol`

A minor mistranslation that is adjusted in the current version of CU2CL is the handling of the `cudaMemcpyToSymbol` call, designed to copy a buffer from the host into *constant* device memory. Originally, the call was simply removed, with no translation performed or error emitted, breaking semantics of the output program and destroying evidence of the untranslated call. Understandably, it would be preferable to fully support the translation of *constant* memory from CUDA to OpenCL, but this would require substantial changes

in other regions of the translator, not just this singular method call. These difficulties are address further in Section 4.2.1. For the time being, the silent removal behavior has been replaced with a "CU2CL Untranslated" error identifying that the original CUDA call is emitted untranslated into the OpenCL code. This provides a translation-time warning including the location of the call, an inline comment tagging the call to make it easily searchable for manual repair, and finally the less-elegant fallback of an OpenCL compilation error since the call will remain undefined without inclusion of the CUDA header files.

3.2.3 File Overwrites Due to Naming Convention



(a) Initial CU2CL Prototype (b) Current CU2CL Prototype

Figure 3.5: OpenCL Output Naming Conventions

Another small, but potent issue that has been repaired is the naming convention CU2CL uses for its translated output files. The original prototype would take the original CUDA filename, "foo.cu" for example, strip off the extension, and append "-cl.cpp", "-cl.h", or "-cl.cl" depending on whether the output was a primary host, included host, or device code source file, resulting in the form "foo-cl.cpp". However, a number of applications were noticed that make use of multiple source files needing translation which share a filename, other than the extension. (In particular, identically-named ".cu" and ".cpp" files were frequently seen, which contain the device and host code, respectively. In these cases, the ".cpp" file must be translated as it contains all CUDA API calls, such as device memory declarations

and initializations.) Therefore, as CU2CL would simply trim the file extension during construction of the output files, these cases would result in a silent overwrite of the output files from whichever source file was translated first, as shown by the overlapped red and grey files in Figure 3.5a. To reliably ensure that no such overwrites could occur, the behavior of the output writing routines was modified to no longer remove the original file extension, but still confer the same new extension, as shown in Figure 3.5b. (For example “foo.cu” and “foo.cpp” would result in output of the form “foo.cu-cl.cpp”, “foo.cu-cl.cl”, and “foo.cpp-cl.cpp”.) The behavior of CU2CL’s automatic `#include` rewriting was also modified to reflect this change.

3.2.4 Handling of Device-Side Kernel Parameter Rewrites

```
__global__ void Kernel(float* foo1,
                       float3* foo2, float3* foo3,
                       float3* fooKernVar4);
```

(a) CUDA

```
__kernel void Kernel(__global float* foo1,
                       __global float4__global float4
__global float4Var4);
```

(b) Initial CU2CL Prototype[3]

```
__kernel void Kernel(__global float* foo1,
                       __global float4* foo2, __global float4* foo3,
                       __global float4* fooKernVar4);
```

(c) Current CU2CL Prototype

Figure 3.6: Demonstration of Kernel Parameter Rewrites

When translating CUDA kernel declarations to OpenCL, a key facet of the process is the translation of kernel parameter declarations. While in most cases this simply requires prepending the `__global` attribute to pointers to device buffers transferred from the host, a select set of buffer types require an additional translation phase to convert the actual variable

[3]The loss of formatting and critical syntax here is intentional and matches the original prototype’s output from the CUDA code in Figure 3.6a.

type to one supported by OpenCL. One such example is the `float3` vector type, which must be converted to the `float4` type in order to support OpenCL 1.0, as 3-member vector types were not supported until OpenCL 1.1. The original translator prototype provided functional code for performing both translation steps, which performed correctly in isolation. However, when both stages were performed together, there was a slight unforeseen consequence to the type translation that occurred after the insertion of the `__global` attribute, due to the internal (intentional) behavior of the Clang Rewrite mechanism which does not modify *SourceLocation*s to account for inserted text. This resulted in malformed translation of the variable's type, as the source range modified was expanded by 9 characters, accounting for the 8 characters of the global attribute as well as the addition of a space. In many cases this caused errant overwrites of portions of the variable's identifier, as well as instances of deleting a portion of the next parameter declaration or the closing parenthesis of the function parameter listing. Figures 3.6a and 3.6b show an example of a CUDA kernel declaration with vector parameters, and demonstrate the original malformed OpenCL output. To avoid this behavior, CU2CL's generic wrapper method for type rewrites has been modified to include an adjustable offset, in order to manually adjust the size of source ranges to account for expansions or contractions resulting from previous rewrites. This results in the correct translation, demonstrated in Figure 3.6c.

3.2.5 Removal of Functions Lacking Explicitly-Declared Return Type

The original CU2CL prototype largely assumed C-like CUDA source with very little use of CUDA's C++ support. As such, its handling of classes had notable deficiencies in some areas. While OpenCL provides no real barriers to usage of C++ on the host-side, it explicitly disallows any C++ code present in source files compiled for the device. As CU2CL creates

host- and device-specific files by selective editing and removal of code from duplicates of the original source file, this presents the case of having to remove valid host-side C++ structures from an output device source file. Originally, the prototype was built with the assumption that all methods that might be removed would either have an explicit return type declaration or function attributes, such as CUDA's `__global__` attribute for kernels. However, this ignored special C++ functions such as constructors and deconstructors, whose return type is implicit. Thus, the translator was incapable of correctly removing these functions from device code. CU2CL's function removal method has been expanded to now correctly support the removal of all functions without an explicit type declaration or attributes.

3.2.6 Inhibited Translation of Implicitly-Defined Functions

Another side effect of CUDA's support for C++ is the potential presence of implicitly-defined functions such as default copy and assignment operators for classes. As these are implicitly defined, there exists no actual source code to be removed from the device output source file for these functions. However, such implicitly defined functions are nonetheless present in the AST generated by Clang. In the first prototype this created a situation in which the translator encountered such an implicitly-defined host-side function on the AST, and thus concluded it needed to be removed from device code, despite the fact that no explicit source was present. Additionally, it would attempt to recurse into the function to examine it for necessary rewrites before being emitted into the host-side source. The translator's function declaration handling code has been upgraded to explicitly detect such implicit functions in the AST and skip over them, emitting a courtesy notification in the process. This prevents errors caused by the translator's attempts to remove or modify non-existent code.

3.2.7 Host-side Arrays of Device Buffers

A number of sample applications made use of the common programming pratice of storing device buffers that are associated with the same task in an array on the host side. This practice is of particular utility to applications that perform multiple device iterations that need to update input or output values between iterations in an asynchronous fashion, a practice commonly known as *multiple buffering*. However, the possibility of a host-side array of pointers to device memory was not adequately supported in the original CU2CL prototype. It appropriately translated array references when translating a `cudaMalloc` call, but when translating the original type of a statically-sized array to a `cl_mem` pointer it lost the array size declaration, resulting in only a single `cl_mem` variable declaration. This results in a number of possible post-translation compilation errors with little to no indication as to the cause, without direct comparison to the original CUDA declarations and device memory allocations. Thus, the mechanism that provides `cl_mem` translation has been adapted to to appropriately handle such statically-allocated host-side arrays of device pointers.

3.3 Bugs Quarantined

A major detriment to the utility of any tool to accomplish its stated purpose is the prevalence of high-profile errors, such as segmentation faults, and other large-scale failures. The original CU2CL prototype's "survivability" in the presence of unanticipated structures in input CUDA source was somewhat deficient. As shown in Figure 2.2 and Tables A.5, A.6, and A.7, a large number of the sample applications failed to produce even a partial translation due to a range of different failures, which often compounded on one another, as demonstrated by the numerous applications with multiple related repairs. Although many of these cases represent source structures that are unlikely to be automatically translated in the immediate future,

or possibly ever if no mapping becomes available, each of these applications still contains a significant quantity of translatable CUDA structures. Therefore, to eliminate complete failure cases, the tool had to be upgraded to provide automated detection and avoidance of the source structures causing failure. By detecting these structures, CU2CL can emit intelligent notifications to the user in order to facilitate manual translation efforts. Further, by avoiding affected regions of code, it is able to continue producing a best-effort translation of the remaining source. Thus, by effectively quarantining these failure cases, the effective utility of the tool has dramatically increased.

3.3.1 Template Handling

As the OpenCL standard does not support any form of device-side C++ code, supporting translation of C++ templates when they are present in CUDA code — as allowed by the CUDA standard — is difficult, particularly when they occur within device code. Many instances were observed of template-dependent kernels being used to execute similar code on varying data structures via template specializations. While it is theoretically plausible that individual OpenCL kernels could be generated for each individual template specialization used by a program, such support would require large-scale additions to the translator. These would take the form of specialized code for handling detection of all possible kernel template specializations, development of a standardized naming convention for generated specialized kernels, as well as addition of a significant amount of host-side handling code for converting templated kernel entry points into wrappers which correctly select the device side kernel invocation required by the specialization. The OpenCL standard has advanced rapidly and there have been demands for simplified C++ support more similar to CUDA, which has resulted in improvements like OpenCL's host-side C++ bindings [21]. Thus, it is also plausible that a new OpenCL standard supporting device-side C++ might be released before

adequate translation support can be added to CU2CL. As such, attempting translation of templates has been deferred in favor of targeting more readily-reachable gains in effective coverage.

However, the original translator often encountered inelegant translation failures in CUDA codes that made use of templates, due in part to attempted multiple removals of individual specializations of CUDA kernels when rewriting host code. Specialized code has been added to detect instances of template usage, both in host and device code, and to inhibit any attempts at translation of individual specializations. The code eliminates these failures and instead produces various error notifications through our standard error reporting mechanism. The exact notification returned is dependent on which CU2CL module encountered the template-dependent structure. This allows the translator to continue to work on the remainder of the CUDA application that is not dependent on template specialization, including portions of kernel functions unrelated to the template type, producing a best-effort partial translation. It also helps to direct programmer attention to the affected regions to support their manual translation.

3.3.2 Launching a Kernel Function Pointer

As a fortunate side effect of how CUDA kernels and their invocations are handled by the `nvcc` CUDA compiler, it is rather easy to make use of function pointers to these kernels directly in invocations, with or without a dereference, using the standard execution configuration syntax shown in Figures 3.7a and 3.7b. This practice was observed in only two of the sample applications — *transpose* from the CUDA SDK and *Fen Zi*— and intended to make use of varying accelerated kernels depending on runtime configuration parameters, much like the use of traditional host-side function pointers. However, CU2CL performs kernel

```
void (*kernelPtr)()= &kernel;
dim3 block(BDIM_X, BDIM_Y);
dim3 grid(GDIM_X / block.x, GDIM_Y / block.y);
(*kernelPtr)<<<grid,block>>>();
```

(a) Dereferenced CUDA Kernel Pointer Launch

```
void (*kernelPtr)()= &kernel;
dim3 block(BDIM_X, BDIM_Y);
dim3 grid(GDIM_X / block.x, GDIM_Y / block.y);
kernelPtr<<<grid,block>>>);
```

(b) Direct CUDA Kernel Pointer Launch

```
cl_kernel __cu2cl_Kernel_kernel = clCreateKernel(...);
. . .
void (*kernelPtr)()= &kernel;

size_t block[3] = {BDIM_X, BDIM_Y, 1};
size_t grid[3] ={GDIM_X / block.x, GDIM_Y / block.y, 1};
localWorkSize[0] = block[0];
localWorkSize[1] = block[1];
localWorkSize[2] = block[2];
globalWorkSize[0] = grid[0]*localWorkSize[0];
globalWorkSize[1] = grid[1]*localWorkSize[1];
globalWorkSize[2] = grid[2]*localWorkSize[2];

clEnqueueNDRangeKernel(__cu2cl_CommandQueue, __cu2cl_Kernel_kernelPtr, 3, NULL,
        globalWorkSize, localWorkSize, 0, NULL, NULL);
```

(c) Incorrect CU2CL Kernel Pointer Translation

```
cl_kernel __cu2cl_Kernel_kernel = clCreateKernel(...);
. . .
cl_kernel kernelPtr= __cu2cl_Kernel_kernel;

size_t block[3] = {BDIM_X, BDIM_Y, 1};
size_t grid[3] ={GDIM_X / block.x, GDIM_Y / block.y, 1};
localWorkSize[0] = block[0];
localWorkSize[1] = block[1];
localWorkSize[2] = block[2];
globalWorkSize[0] = grid[0]*localWorkSize[0];
globalWorkSize[1] = grid[1]*localWorkSize[1];
globalWorkSize[2] = grid[2]*localWorkSize[2];

clEnqueueNDRangeKernel(__cu2cl_CommandQueue, kernelPtr, 3, NULL,
        globalWorkSize, localWorkSize, 0, NULL, NULL);
```

(d) Ideal CU2CL Kernel Pointer Translation

Figure 3.7: Kernel Function Pointer Launches

translation in two separate stages. The first translates the device-side kernel declaration and definition, and creates a `cl_kernel` object based on prefixing `__cu2cl_Kernel_` to the function's name. The second translates the host-side kernel invocation by adding an identical

prefix to the name of the invoked kernel. An issue arose when this prefix was applied to a direct invocation of a kernel function pointer, resulting in a reference to a non-existent `cl_kernel` object, as demonstrated by Figure 3.7c. This results in inelegant post-translation compilation failure without readily observable cause, as the translated kernel call is otherwise correctly formatted. OpenCL would theoretically support the emulation of these function pointers, via conversion to the `cl_kernel` opaque type, in a form similar to Figure 3.7d. Upgrading the translator to handle detection or conversion of kernel pointer declarations remains as future work. However, as a precursor to adding this functionality, CU2CL's handling of kernel invocations has been upgraded to include explicit detection of direct invocation of kernel function pointers. As CU2CL does not yet provide a robust translation of these pointers, this explicit detection code skips translation of these invocations, and emits a standardized error to direct manual post-translation code repair efforts.

3.3.3 Failure When Main Method is not Present

It is well known that standard OpenCL requires extensive explicit initialization code for setting up portions of an execution environment like creating a device context and command queue and compiling device kernels. Fortunately, assuming one wishes to emulate CUDA's automatic device initialization code, much of this initialization code, referred to as *boilerplate*, is almost completely identical between programs. Therefore the original CU2CL prototype provided automatic insertion of this code, as well as associated cleanup OpenCL calls into the main method of the translated application. However, this was based on the assumption of rather simple single-source-file CUDA programs, when in practice applications frequently make use of many source files, often mixing between CUDA, C++, C, and occasionally other languages such as FORTRAN within a single application. While the original work noted the need for explicit support of separate compilation [26, 25], the assumptions

inherent in the original prototype prevented even partial translation of source files that did not bear a main method. Due to a lack of verification that a main method was present when attempting to insert boilerplate, the prototype would always encounter a segmentation fault when attempting to translate a file that did not contain a main method.

While the newest version of the translator still does not provide the hooks necessary for full support of separate compilation, the assumption underlying these segmentation faults was addressed by adding an explicit check for the existence of a main method. This check then inserts the required boilerplate as usual if such a method is found, or emits a notification to the user via the standardized error reporting mechanism if no such method is found. Regardless of whether a method is found, translation of the remainder of the application is preserved. Thus, even if the necessary boilerplate for a non-main-bearing source file must be developed and added by hand, the newest version of CU2CL survives to provide a partial translation. This still significantly reduces the amount of hand translation required compared to the first prototype. However, the avoidance mechanism could be further expanded to automatically dump suggested boilerplate into a comment region at the beginning of the source file, to further aid manual efforts as an interim effort until full support is added.

3.3.4 Handling of Separately Declared and Defined Kernels

A rather significant issue that is removed in the current CU2CL release was an inherent assumption that a CUDA kernel would solely have a definition, not a forward declaration as well as a definition. CU2CL detects CUDA kernel declarations/definitions based on presence of the `__global__` or `__device__` attribute in order to replace them with the OpenCL equivalents. This resulted in a number of cases in which the translator would attempt to remove all code between declaration and definition when rewriting the host-side OpenCL

by basing a removal off the *SourceLocation* of the wrong attribute. Were this to occur with source structures early in the source file, this behavior cascaded into additional further obfuscated errors due to the resulting extreme malformation of the partially-rewritten host code. However, a method for checking if both a declaration and definition were present, based on the presence of a duplicate of either of these attributes was sufficient to eliminate this errant behavior and prevent the translation failure. While the kernel definition is now appropriately removed from host-side OpenCL output, removal of the corresponding forward declaration is yet to be implemented.

3.3.5 Handling for `#defined` Functions

During investigation of several applications that make use of OpenGL interoperability, an additional failure was discovered when the translator attempted to recurse into certain OpenGL function calls. This was due to the translator being unable to identify a direct callee for the expression. An explicit catch was inserted to eliminate segmentation faults associated with this situation by inhibiting translation and emitting a notification, much like the other failures in this section. However, beyond identifying a chain of `typedef` and `#define` statements that collectively assemble these functions' declarations, a more formal understanding of the underlying cause and need/potential for translation has yet to be constructed.

Chapter 4

Characterization of Translator Limitations

Information gained from profiling the sample applications both before and after enhancement of the translator was used to develop a characterization of structures that pose current and future limitations to translation. This characterization is provided to help prioritize development and reason about the theoretical translatability of difficult structures. Table 4.1 demonstrates the percentage of applications from both the CUDA SDK samples and Rodinia samples that make use of a number of these structures. By aggregating information on untranslated structures across these applications a more adequate estimate of their relative popularity in the wider population of CUDA applications was developed. Further, by studying multiple disjoint instances of these CUDA structures, more complete characterizations of potential difficulties in realizing their translation were developed. Based on this characterization, each of the identified issues that remain to be translated have been further classified based on their potential for translation. Alongside the theoretical discussion of identified issues, possible avenues to realizing their partial or complete translation are pre-

Challenge	CUDA SDK Frequency (%)	Rodinia Frequency (%)
Separate Compilation	54.4	29.4
CUDA Libraries	10.1	0.0
Kernel Templates	21.5	0.0
`cudaSetDevice`	54.4	29.4
Textures	27.8	23.5
Graphics Interoperability	24.1	11.8
CUDA Driver API	8.9	5.9
Literal Parameters	19.0	17.6
Aligned Types	6.3	5.9
Constant Memory	17.7	29.4
Shared Memory	46.8	70.6

Table 4.1: Frequency of Translation Challenges in Sample Applications

sented. A portion of this work is presented in [36]. Exploring the issues presented in this chapter further and implementing automatic translations for those that are possible provide ample opportunities for future work.

4.1 Partially Supported 1-to-1 Mappings

4.1.1 Device Buffer `cl_mem` Propagation

As mentioned in previous works and earlier sections in this thesis, when translating buffers of on-device memory from CUDA to OpenCL, one must translate the buffer's standard pointer type from CUDA into OpenCL's opaque `cl_mem` type. This type must be used in translated memory allocations, memory transfers, and kernel invocations, which often reside in separate scopes. Currently, CU2CL supports translation of the variable's nearest declaration that shares the scope of the translated *cudaMalloc* call, either as a global variable, function parameter, or local variable. Further, during conversion of kernel functions, it can infer that

any pointer argument to a kernel function must be translated to an OpenCL `cl_mem` type within the host-side invocation. However, when these two translated structures do not occur within the same scope, and the device variable is instead passed as a parameter to one or more additional function calls, it is likely that the `cl_mem` type rewrite will require additional propagation throughout each of these additional function calls, as noted in [27].

As CU2CL currently provides rather localized translation for device buffers — often requiring post-translation manual intervention to perform the necessary modifications to accessory functions in the call stack — it is critical to begin devising ways of reducing or removing the necessity of manual intervention in favor of more complete automation. One such method might be to simply wrap all outbound and inbound `cl_mem` references with an explicit cast to the original type before they are propagated up and down the stack. This method would be a relatively "quick fix". However it could cause potentially grievous issues if accessory methods attempt to naively access the cast `cl_mem` pointers as if they were pointers to host-side buffers, as the change in actual type would not be apparent to those methods. Fortunately, dereferencing of a device buffer that is not allocated on the host and page-locked — i.e. allocated with a traditional `cudaMalloc` and not one of the host-side variants — is also invalid in CUDA, so the prevalence of such attempts to access device memory from the host should be reduced.

Another approach would be to integrate within CU2CL a form of control flow analysis to deduce functions requiring rewrites of parameters to the `cl_mem` type. This would obviate the noted problems with the explicit cast technique, ensuring translation propagation within explicitly-included source files. However, a potential issue could still be reached within the case of separate compilation as necessary changes for an already-translated portion of the source may not be immediately visible to the translator while work is being performed on other source files. As separate compilation necessitates the use of header files that are shared

between separately compiled source files, a potential solution might be to add functionality that searches for partially-translated header files from previous translation passes.By examining these partially-translated header files CU2CL could infer a list of translations from other sources that must be propagated into the current source file undergoing translation. However, this does not address the possibility of cyclical dependencies between source files that share a header, requiring additional propagation of rewrites from a later-translated file into an earlier-translated one. This potential may require that the translator perform analysis on all program source files simultaneously, or perform a possibly-variable number of translation passes.

4.2 Unsupported 1-to-1 Mappings

4.2.1 Constant Memory

As noted in Section 3.2.2, the CU2CL translator is yet to support the translation of *constant* memory regions, despite there being an effective 1-to-1 mapping of functional usage within the device kernels. The current lack of support is largely due to variations on the scope for declarations and initializations of such memory regions. In CUDA, `__constant__` variables declared at program scope can be initialized via `cudaMemcpyToSymbol` as long as declaration and initialization reside in the same source file [28]. However, in OpenCL `__constant` memory can either have device program scope, or device function scope. The program scope variant cannot easily be set at runtime via an analog of `cudaMemcpyToSymbol` — it would require detailed modification of the string containing the OpenCL kernel source code at runtime before invoking `clCreateProgramWithSource` — but the function scope variant can, as to the host it appears as a `cl_mem` type, just like any other device side buffer. However, this

comes with the restriction of needing to be explicitly defined as a parameter to kernels that reference the constant memory region. Thus, in order to effect a full translation of constant memory, the translator must both convert the host-side reference to a `cl_mem`, as well as dynamically insert the reference into both the host- and device-side parameter list for kernels that reference the region. In order to avoid over-zealously appending all declared constant memory to parameter lists for all kernels, the translator will need to be extended to actively identify access to variables by a kernel that utilize the constant address space.

4.2.2 Shared Memory

```
//host code
dim3 block(BDIM_X, BDIM_Y);
dim3 grid(GDIM_X / block.x, GDIM_Y / block.y);
size_t s_mem = GDIM_X * GDIM_Y * sizeof(float);
kernel<<<grid,block,s_mem>>>(...);

//device code
__global__ void kernel(...) {
    extern __shared__ float foo[];
... }
```

(a) CUDA C

```
//host code
size_t[3] group_size = {BDIM_X, BDIM_Y, 1}
size_t[3] range_size = {GDIM_X, GDIM_Y, 1}
size_t s_mem = GDIM_X * GDIM_Y * sizeof(float);
clSetKernelArg(__cu2cl_Kernel_kernel, 0, s_mem, NULL);
clEnqueueNDRangeKernel(...);

//device code
__kernel void kernel(__local float* __cu2cl_SharedMem, ...) {
    __local float * foo = __cu2cl_SharedMem;
... }
```

(b) OpenCL

Figure 4.1: Dynamically allocated shared memory

There is also an effective mapping of CUDA's `__shared__` memory space onto OpenCL's `__local` memory space. The initial CU2CL prototype already handled conversion of kernel `__shared__` memory declarations to `__local` memory declarations, which provides support

for statically allocated regions of this type. However, there is a distinct difference between the semantics of dynamically allocating external shared memory for a kernel at runtime. In CUDA, such memory is declared to be external with the addition of the `extern` keyword, and the total memory allocated for *all* such external shared memory declarations within a given kernel is specified as an additional argument to the kernel launch execution configuration. Figure 4.1a provides an example of this behavior, with "s_mem" being used to declared the size of the shared region on the host, and "foo" being a pointer into this region. Therefore if multiple dynamically allocated `__shared__` variables are used, they all point to the same buffer and index offsets must be manually managed by the programmer.

In contrast, OpenCL's dynamically allocated `__local` variables are specified as additional formal parameters to the kernel, and the amount of space allocated for *each* such variable is specified by a separate call to `clSetKernelArg`. Thus, not only must the size of the allocation be extracted from the CUDA execution configuration statement, but additional `__local` parameters must be added to the OpenCL kernel at translation time. Further, a mechanism must be added to ensure that offset indices referring to the memory region remain valid after translation. The direct approach is to emulate CUDA's behavior by allocating a single shared `__local` memory region, and converting all external declarations to initializations that point to this buffer. Figure 4.1b demonstrates the OpenCL this translation mechanism would produce.

4.2.3 Texture to Image Translation

Translation of CUDA `texture`s and `surface`s to OpenCL `cl_image`s is another high-frequency translation that is yet to be supported. Both standards provide for read and write access to the device texture memory, organized in various multi-dimensional configurations. How-

ever, while the two types are conceptually similar, the semantics for utilizing them have a number of notable differences that complicate the translation process. For example, when mapping onto OpenCL 1.0, there is no direct equivalent for CUDA's 1-dimensional `texture` type, so a translator would have to provide a mechanism for translating the one-dimensional indices used by CUDA into a two-dimensional form for use with an OpenCL `cl_image`. Further, CUDA provides both scalar and vector access to texture memory, whereas OpenCL exclusively provides four-member vector access. Additionally, texture access with OpenCL requires use of both a `cl_image` and a `cl_sampler` specifying the addressing, filter, and normalized coordinate settings used to access the image, whereas CUDA includes these settings directly in the texture reference type. Due largely to the number of variations on texture access, much work remains to fully conceptualize the additions necessary to provide automatic translation to OpenCL.

4.3 Functionally Emulatable Mappings

As mentioned in other sections, there exist a number of situations in which CUDA and OpenCL both provide equivalent functionality, but utilize different semantics for achieving it. In most cases this is simply a matter of converting a single CUDA function call or implicit behavior into a collection of various OpenCL structures and API calls. However, two situations were identified that require a more nuanced translation effort to ensure functional correctness once converted to OpenCL . The first of these is an emulation of CUDA's high level mechanism for invoking pointers to kernel functions directly via the execution configuration syntax. This requires intelligent identification of such invoked pointers and conversion to the more verbose OpenCL kernel invocation semantics. The second such case of potential emulation addresses the potential for codes to rely on the implicit synchronization guaran-

teed within warps by virtue of current NVIDIA hardware platforms. Guaranteeing functional portability of codes that make use of this behavior is particularly difficult, as the underlying OpenCL devices may or may not provide similar implicit synchronization behavior.

4.3.1 Kernel Function Pointer Invocation

As addressed in Section 3.3.2, CUDA provides a mechanism by which a kernel function pointer may be directly invoked from the host without a dereference, as if the pointer were itself an actual device kernel. Fortunately, use of this feature even within CUDA's own samples is not widespread, only observed within the SDK sample *transpose*, and the *Fen Zi* application. While OpenCL provides no similar method of kernel invocation to that of CUDA's high-level execution configuration syntax, this does not inhibit its potential ability to emulate function pointers. This is due to the fact that by default, when translating a CUDA kernel invocation using the execution configuration syntax, the translator must necessarily convert the kernel function to OpenCL's `cl_kernel` type, in conjunction with other additions in the boilerplate region to force compilation of the kernel and initialization of the corresponding `cl_kernel` object. Fortunately, this `cl_kernel` is itself an opaque pointer type. Therefore, it is possible that a translator could actively convert CUDA function pointer variables into additional `cl_kernel` variables, that would then point to the appropriate OpenCL kernels, and be seamlessly used in kernel invocations, just like their non-pointer-based analogues. Figure 3.7d gave an example of how this translation would appear in OpenCL. Addition of this functionality would require a modification of CU2CL's host-side translation code to actively search for kernel pointer declaration and assignment expressions, in order to drive conversion of the pointer types to `cl_kernel` variables.

4.3.2 Warp-level Synchronization

```
...
//work-group size 32
__local r_vals[32];
r_vals[tid & 31] = some_var[tid];
barrier(CLK_LOCAL_MEM_FENCE
    | CLK_GLOBAL_MEM_FENCE);
if ((tid & 31)< 16)
    r_vals[tid & 31] +=
        r_vals[(tid + 16) & 31];
//implicit warp sync
if ((tid & 31)< 8)
    r_vals[tid & 31] +=
        r_vals[(tid + 8) & 31];
//implicit warp sync
if ((tid & 31)< 4)
    r_vals[tid & 31] +=
        r_vals[(tid + 4) & 31];
//implicit warp sync
if ((tid & 31)< 2)
    r_vals[tid & 31] +=
        r_vals[(tid + 2) & 31];
//implicit warp sync
if ((tid & 31) == 0)
    r_vals[0] += r_vals[1];
```

(a) OpenCL kernel reduction with warp-level synchronization

```
...
//work-group size 32
__local r_vals[32];
r_vals[tid & 31] = some_var[tid];
barrier(CLK_LOCAL_MEM_FENCE
    | CLK_GLOBAL_MEM_FENCE);
if ((tid & 31)< 16)
    r_vals[tid & 31] +=
        r_vals[(tid + 16) & 31];
barrier(CLK_LOCAL_MEM_FENCE
    | CLK_GLOBAL_MEM_FENCE);
if ((tid & 31)< 8)
    r_vals[tid & 31] +=
        r_vals[(tid + 8) & 31];
barrier(CLK_LOCAL_MEM_FENCE
    | CLK_GLOBAL_MEM_FENCE);
if ((tid & 31)< 4)
    r_vals[tid & 31] +=
        r_vals[(tid + 4) & 31];
barrier(CLK_LOCAL_MEM_FENCE
    | CLK_GLOBAL_MEM_FENCE);
if ((tid & 31)< 2)
    r_vals[tid & 31] +=
        r_vals[(tid + 2) & 31];
barrier(CLK_LOCAL_MEM_FENCE
    | CLK_GLOBAL_MEM_FENCE);
if ((tid & 31) == 0)
    r_vals[0] += r_vals[1];
barrier(CLK_LOCAL_MEM_FENCE
    | CLK_GLOBAL_MEM_FENCE);
```

(b) OpenCL kernel reduction with explicit synchronization

Figure 4.2: Warp-level synchronization in reduce

Implicit synchronization is one area in which porting code targeted specifically towards NVIDIA GPUs to a myriad of OpenCL devices can potentially result in functionally incorrect results. By virtue of current NVIDIA device architectures, CUDA programs can rely on warps of 32 sequentially-indexed threads being executed in complete lock-step, obviating the need for explicit synchronizations between expressions in certain algorithms. In contrast, while these NVIDIA devices retain a warp size of 32 when executing OpenCL kernels, AMD GPUs have an effective warp size, or *wavefront* of 64, CPUs may have an effective warp size of either 1 or their SIMD width, and other devices may similarly provide all or none of this implicit synchronization behavior. Therefore, when translated to OpenCL, CUDA codes that

rely on this behavior may show correct execution profiles on NVIDIA devices, and in lucky cases on AMD GPUs, but will almost certainly result in nondeterministic loss of correctness when executed on CPUs due to the relaxation of inherent underlying synchronization. For example, a typical OpenCL reduce operation in a work-group of 32 threads is demonstrated in Figure 4.2. The code in Figure 4.2a lacks explicit synchronization in favor of warp-level synchronization, and will only guarantee a correct result on hardware platforms with a warp size of at least 32. However, the code in Figure 4.2b includes the necessary explicit synchronization to reach a correct answer on any OpenCL device. This directly inhibits the goal of functional correctness and is exacerbated by a translator's relative inability to detect usage of such synchronization at the source level due to the lack of an explicit synchronization syntax. However, there have been efforts to automatically analyze code for the possible use of warp-level synchronization intrinsics in the context of GPU to CPU translation using dependence analysis [18]. Thus, it is possible that a similar functionality could be integrated into the tool to either automatically augment OpenCL device code with additional explicit synchronization, or at the very least emit a warning to the user that implicit synchronization has likely been detected. Thus, manual efforts could be more readily directed to handle the loss of correctness likely observed when such code was executed on a non-NVIDIA platform.

4.3.3 OpenGL Interoperability

Both CUDA and OpenCL provide conceptually-similar mechanisms for interoperating between the compute context and OpenGL graphical context. This allows the sharing of buffers between contexts such that *in situ* visualization may occur between computation steps, without the need for additional GPUs or transfer of buffer data back to the host. While there exist several 1-to-1 mappings between API calls, such as `cudaGraphicsMap/UnmapResources` to `clEnqueueAcquire/ReleaseGLObjects`, other portions of interoperability modules have

more complex mappings, such as the previously mentioned texture to image translation. Work remains to conceptualize a complete functionally-equivalent mapping of this portion of the specifications and integrate it into the translator prototype, but has been deferred in favor of addressing other more critical translation issues.

4.4 Device Language Expressivity Limitations

A key restriction of source translation efforts is the difficulty in attempting to map a more expressive language onto another with reduced expressivity. In the case of CUDA-to-OpenCL translation, this reduction of expressivity takes a number of forms, some of which may be more readily resolved than others. The first of these concerns the core device language specifications of the two APIs, as CUDA's nearly complete support for on-device C++ results in a number of potential source structures that cannot be concisely represented in OpenCL's more limited variant of C99. Additionally, CUDA provides functions that give access to special purpose features of their hardware, which are not present in the OpenCL standard due to its intention to support devices from a myriad of vendors.

4.4.1 Mapping C++ to C

One of the main areas in which OpenCL's device language provides less expressivity than CUDA's is the disparity between CUDA supporting essentially the full C++ standard on device, whereas OpenCL only supports a variant of C99. Thus, there are a number of C++ programming practices and constructs that have no immediately-realizable representation in OpenCL's device language. One such feature that has already been given substantial attention in Section 3.3.1 is the use of templates. These could potentially be emulated via

extensive additions designed to generate all necessary kernel specializations in C. Another such issue is support for classes. While these see significantly less usage on-device, they remain a potential roadblock. In the majority of cases they could be largely replaced by C structs, however a method for automatically realizing this translation has not yet been devised, and will likely require a CFront-like approach of "compiling" C++ down to C.

4.4.2 Threadfence and Other Intrinsic Device Functions

Additionally, CUDA provides access to several on-device operations for which OpenCL provides no direct or emulatable equivalence. One such operation is `__threadfence()`, which waits until all outstanding global or shared memory accesses by the calling thread are visible to threads in the same block for shared memory accesses or threads in the same device for global accesses [28]. Similarly, the `_system()` variant of threadfence that supports coherency with page-locked host memory also has no equivalent mapping. However, `__threadfence_block()` does accurately map onto OpenCL's `mem_fence()` device function with both `CLK_LOCAL_MEM_FENCE` and `CLK_GLOBAL_MEM_FENCE` flags set. The `__syncthreads()` command also maps to OpenCL's `barrier()` operation with both `CLK_LOCAL_MEM_FENCE` and `CLK_GLOBAL_MEM_FENCE` flags. However, there is no mapping for CUDA's `_count()`, `_and()`, and `_or()` variants of `__synchthreads`. Additionally, CUDA provides support for the "Warp Vote Functions" `__all()`, `__any()`, and `__ballot()` that are not represented in OpenCL. There remain several other functions in this category, added as additional functionality became available on newer devices, a trend that is likely to continue. In these cases, until the function can be formally translated in an OpenCL specification, the most viable option is to emit a notification to the user, to advise them of the limitation so that they might manually refactor their algorithm to remove reliance on the unsupported function.

4.5 Inherited Limitations

During the course of profiling, it became apparent that complete source-level CUDA-to-OpenCL translation was significantly inhibited by CUDA's basis as a C++ variant. By its construction, C and its descendant languages, such as CUDA, allow for a number of programmatic techniques and code structures that lie largely outside the reach of source translation. Three such structures have been identified that reduce CU2CL's effective code coverage in the sample applications. They are the use of preprocessor macros, multiple source file applications that are separately compiled and linked, and the integration of non-source precompiled modules.

4.5.1 Preprocessor Macros

Source translation of heavily-preprocessed languages is known to be a difficult problem [37]. As CUDA is effectively an extension to C++, any preprocessor construct that is valid in C++ could be encountered in a CUDA program. Should the translator implement its own preprocessor and parser, it is possible that it might be able to handle these directives in stride, without any significant limitations on its effective reach. More frequently, source translators make use of an extant parser, likely from a modular compiler, as has been done with CU2CL [26, 25]. Many translators perform their transformations by walking the AST and directly modifying the AST, from which output source code is then regenerated by walking the modified tree. However, CU2CL instead uses the original AST assembled by Clang merely as a guide for identifying constructs demanding translation that are then rewritten by modification of the original source pointed to by the various AST nodes. In both cases, this raises a problem when dealing with preprocessor macros of all varieties, as they are inadequately represented on the AST and translation occurs long after preprocessing.

CU2CL is able to infer from data fields on the AST nodes whether a given SourceLocation represents a macro, but there are limited tools for performing accurate string manipulations on located macros without manually implementing portions of parsing and preprocessing behavior. Extending Clang/LLVM to provide more information in these nodes would simplify rewriting of function-like macros and `#defined` constants. However, preprocessing causes an additional issue with translation in the presence of conditional compilation macros. As these are evaluated at preprocessing time, the compiler/translator only ever even "sees" one version of the program, with possibly widely varying function signatures, global state, and execution behavior between multiple versions. Currently, in order to achieve full translation of such a source file, one would have to manually perform multiple translation passes with varying combinations of conditional compilation arguments, in order to reach all possible code paths, and then manually merge the translated output files from all passes. A more automatic solution to this problem has not been fully constructed, but might take the form of generating multiple ASTs with varying conditional compilation definitions. It is possible that the Clang preprocessor could also be modified to allow the simultaneous parsing of incompatible conditional branches, similar to the method used in [17].

4.5.2 Separate Compilation

The separate compilation and linking stages of building applications in C and its descendants introduce an additional layer of complexity when translating multiple-file applications. In the case of CUDA-to-OpenCL translation, this is only exacerbated by the presence of a largely-implicit global state in CUDA applications, which must be made completely explicit within OpenCL applications. This implicit global state consists of the device context, compiled kernels, device queues, and other associated CUDA mechanisms that function behind the scenes to provide a unified runtime state across application components, which may reside

in separate source files. In contrast, as mentioned elsewhere, OpenCL requires each of these components to be explicitly managed, and thus such management must be moved into application source. However, during translation this creates the complication of requiring access and modification of the explicit global state by code components that quite likely reside in separate source files, and are translated as distinct entities. Thus, complete source translation must provide a mechanism for ensuring cross-file global state modifications are accurately integrated in the output application, while preserving the separate compilation and linking semantics of the original CUDA. As mentioned in Section 3.3.3, a potentially viable solution would be to make use of the header files that are already shared between linked binary modules. This would require a mechanism for supporting the inspection of any partially-translated OpenCL equivalents created from a previous translation of another file, and potentially multiple translation passes or simultaneous analysis of all program source files.

4.5.3 Precompiled Modules

One area in which a source-level translation effort is necessarily inhibited is in the translation of applications that make use of non-source components. This commonly takes the form of precompiled libraries, such as the many developed to provide high-performance CUDA implementations of commonly used operations. These modules see common use due to the convenience provided to programmers by removing the need to hand-develop and tune application components, the desire to allow programmers to make use of such libraries without the need to release proprietary source code, and the libraries' high performance. Therefore, a source translator must be mindful of the possibility of encountering API calls for such libraries. However, without access to library source, only library headers will be translated, and the output OpenCL will not be able to utilize the CUDA library. Further, portions of

CUDA applications can be implemented in NVIDIA's PTX IR, which cannot currently be converted to high-level OpenCL, but must be accounted for nonetheless.

Fortunately, there remain possible approaches to providing functionally portable translation in many cases despite the prevalence of these non-source components. For example, in the case of closed-source libraries for accelerated device code, there is a growing drive to provide similar or equivalent OpenCL libraries. Thus, for select cases, it is plausible that a source translator could recognize closed-source CUDA library calls for which there exist OpenCL equivalents, and translate the application to instead make use of the OpenCL libraries. Further, for the case of translating codes that make use of PTX, translation might take a hybrid approach. Translation of high-level CUDA source could be performed by a tool like CU2CL, with PTX translation to other architectures performed separately, via tools like Ocelot [9, 10] and Caracal [11].

4.6 Engineering Limitations

Many limitations of the current CU2CL translator prototype do not represent limitations of the CUDA and OpenCL languages themselves, nor do they represent limitations of automatic translation approaches. Rather, they represent situations where there is a well-defined mapping of CUDA to OpenCL and the translation has simply not yet been implemented. Each of these has already received some discussion within Chapter 3, and they are briefly presented here so that this chapter can serve as a summary of opportunities for future work on the CU2CL translator. Notable features of the translator that need implementation or modification include:

- Support for explicit rounding modes of all builtin kernel math functions (Section 3.1.3)

- Emulation of the CUDA Runtime's background context saving in the `__cu2cl_setDevice` handler for `cudaSetDevice` (Section 3.1.4)
- Compilation of OpenCL kernels for the new device and context in the `__cu2cl_setDevice` handler for `cudaSetDevice` (Section 3.1.4)
- Support for checking if an OpenCL context has been previously released due to translation of a `cudaThreadExit` (Section 3.1.4)
- Support for parsing and rewriting the parenthetical portion of function-like preprocessor macros (Section 3.1.5)
- Complete the scaffold for handling of CUDA's `__align__` attribute (Section 3.1.6)
- Support for removal of forward declarations of CUDA kernels from OpenCL host code (Section 3.3.4)

Chapter 5

Conclusion

Accelerator-based heterogeneous computing has seen significant growth and interest in recent years, and there exist a number of approaches to providing programmatic access to accelerator devices. Some approaches take the form of a proprietary framework, like CUDA, providing ease of development on a single family of devices, whereas others take a more open approach, attempting to provide a single interface for programming a multitude of underlying device families, like OpenCL. When developing an accelerated application, performance and programmability are key considerations that often sway the decision of which accelerator framework is chosen. However, portability is also a significant consideration, to provide a wider user-base for accelerated software and to reduce the development cost required to access alternative or future devices that may afford increased performance. An expansive range of applications have been developed in CUDA that are currently vendor-locked to NVIDIA platforms barring time-intensive manual translation to another framework, such as OpenCL. Therefore, there is a demand for automated translation from CUDA to OpenCL. This thesis consists primarily of work to support automatic translation efforts, in particular the CU2CL prototype CUDA-to-OpenCL source-to-source translator.

Chapter 2 of this thesis discussed efforts to provide quantitative measurement of several facets of the CU2CL prototype translator. This work was performed in the context of a large population of sample applications, consisting of 79 samples from the CUDA SDK [30], 17 samples from the Rodinia benchmark suite [5, 6], and three large applications, GEM [2, 14], Fen Zi [3, 15, 16], and IZ PS [44]. Two notable conclusions were reached through this analysis of the translator prototype. First, automatic translation dramatically reduces the cost of porting an application from CUDA to OpenCL. Translations that used to take days, weeks, or months when performed by hand can now be achieved in mere seconds or less, even for relatively large applications. The second major conclusion coming from analysis of the translator prototype is that when executing on the same NVIDIA platform with a modern driver and CUDA runtime, developers pay no major runtime performance penalty for porting CUDA applications to OpenCL.

Chapter 3 provided a detailed discussion of specific changes made to the translator that underlie the dramatic improvement in its reliability and utility. By actively identifying syntax elements from the sample population that provide challenges to the translator, a number of bugs observed in the original prototype were effectively removed. Additionally, by examining the translations of the sample applications produced by the original prototype, handling for special variants of a number of CUDA structures was significantly improved. Finally, several new features were added to the translator, including a robust error reporting mechanism, designed to improve the tool's ability to actively guide manual translation of unsupported CUDA structures.

Finally, Chapter 4 identifies a number of current limitations to providing automatic CUDA-to-OpenCL translation. Many limitations addressed there are somewhat specific to the CU2CL prototype, and describe portions of the CUDA specification for which there exist valid OpenCL equivalences, but for which automatic translation has not yet been imple-

mented. This chapter also discussed a number of limitations that are not necessarily specific to the CU2CL translator, but rather are specific to deficiencies in mapping CUDA onto OpenCL. By identifying and addressing these limitations, a more complete understanding of the difficulty in providing functional portability for CUDA codes via translation to OpenCL was constructed. In spite of these limitations, this thesis has demonstrated the automatic translation from CUDA to OpenCL is largely achievable, dramatically reducing the cost of providing CUDA programs access to new accelerator devices and users.

Appendix

Application	CUDA Lines	Total Translation Time (s)	CU2CL Time (μs)
Back Propagation	313	0.14	174
Breadth-First Search	306	0.14	200
CFD	2371	1.07	1230
Gaussian	390	0.14	210
Heartwall	2018	0.17	532
Hotspot	328	0.14	204
Kmeans	494	0.14	241
LavaMD	240	0.14	192
Leukocyte	624	0.28	386
LU Decomposition	332	0.28	277
MummerGPU	3786	0.18	655
Nearest Neighbor	278	0.17	170
Needleman-Wunsch	430	0.14	191
Particle Filter	1517	0.31	582
Path Finder	235	0.14	186
SRADv1	541	0.15	366
Stream Cluster	443	0.26	211

Table A.1: Rodinia Sample Translation Times

Application	CUDA Lines	Total Translation Time (s)	CU2CL Time (μs)
alignedTypes	316	0.16	239
asyncAPI	135	0.14	163
bandwidthTest	891	0.28	289
bicubicTexture	1251	0.78	482
bilateralFilter	864	0.89	415
binomialOptions	443	0.64	328
BlackScholes	347	0.27	200
boxFilter	980	0.74	339
clock	162	0.15	149
concurrentKernels	177	0.27	177
conjugateGradient	196	0.06	170
convolutionFFT2D	1175	0.65	488
convolutionSeparable	363	0.75	288
convolutionTexture	368	0.63	295
cppIntegration	247	0.73	261
dct8x8	1715	0.29	539
deviceQuery	165	0.57	160
deviceQueryDrv	150	0.58	150
dwtHaar1D	598	0.16	281
dxtc	886	0.43	472
eigenvalues	3109	0.48	1116
fastWalshTransform	327	0.15	208
FDTD3d	870	0.99	405
fluidsGL	811	0.28	330
FunctionPointers	1004	0.76	449
histogram	545	0.90	436
imageDenoising	1305	0.75	512
lineOfSight	337	0.17	228
Mandelbrot	2528	0.93	922
marchingCubes	1571	0.80	540
matrixMul	351	0.14	211
matrixMulDrv	525	0.72	378
matrixMulDynlinkJIT	301	0.46	158
mergeSort	954	0.65	412
MersenneTwister	310	0.27	193
MonteCarlo	1014	0.79	726
MonteCarlo Multi GPU	994	0.79	743
nbody	2088	1.54	824
oceanFFT	1037	0.76	452

Table A.2: CUDA SDK Sample Translation Times

Application	CUDA Lines	Total Translation Time (s)	CU2CL Time (μs)
particles	1184	2.41	1001
postProcessGL	1291	0.88	489
ptxjit	132	0.58	120
quasirandomGenerator	510	0.90	504
radixSort	2387	1.37	1103
randomFog	888	1.34	345
recursiveGaussian	883	0.77	417
reduction	1063	0.78	583
scalarProd	251	0.16	226
scan	495	0.75	322
simpleAtomicIntrinsics	197	0.15	155
simpleCUBLAS	244	0.10	149
simpleCUFFT	249	0.15	173
simpleGL	603	0.73	350
simpleMPI	208	0.84	274
simpleMultiCopy	351	0.27	254
simpleMultiGPU	226	0.47	202
simplePitchLinearTexture	274	0.15	180
simplePrintf	1066	0.43	893
simpleStreams	243	0.15	193
simpleSurfaceWrite	207	0.15	201
simpleTemplates	458	0.16	248
simpleTexture	239	0.15	186
simpleTexture3D	506	0.78	305
simpleTextureDrv	392	0.72	379
simpleVoteIntrinsics	341	0.15	218
simpleZeroCopy	149	0.15	147
smokeParticles	2016	1.21	531
SobelFilter	780	0.75	360
SobolQRNG	10698	1.73	5275
sortingNetworks	657	0.90	487
template	187	0.15	158
threadFenceReduction	791	0.17	483
threadMigration	434	0.72	393
transpose	571	0.27	271
vectorAdd	147	0.14	97
vectorAddDrv	351	0.60	281
volumeRender	884	0.78	393

Table A.3: CUDA SDK Sample Translation Times (cont.)

Application	**CUDA Lines**	**Total Translation Time (s)**	**CU2CL Time (μs)**
GEM	524	0.14	182
Fen Zi	17768	0.35	3491
IZ PS	7103	0.21	1091

Table A.4: Large Application Translation Times

Application	Original CU2CL Prototype	Current CU2CL Prototype	Related Repairs
Back Propagation	Complete	Complete	
Breadth-First Search	Complete	Complete	
CFD	Failure	Complete	3.3.1
Gaussian	Failure	Complete	3.3.4
Heartwall	Complete	Complete	
Hotspot	Complete	Complete	
Kmeans	Complete	Complete	
LavaMD	Partial	Complete	3.3.3
Leukocyte	Failure	Complete	3.3.3
LU Decomposition	Partial	Complete	3.3.3
MummerGPU	Failure	Failure	3.3.3
Nearest Neighbor	Failure	Complete	3.1.2
Needleman-Wunsch	Complete	Complete	
Particle Filter	Complete	Complete	
Path Finder	Complete	Complete	
SRADv1	Complete	Complete	
Stream Cluster	Failure	Complete	

Table A.5: Rodinia Sample Translation Robustness

Application	Original CU2CL Prototype	Current CU2CL Prototype	Related Repairs
alignedTypes	Failure	Complete	3.3.1
asyncAPI	Complete	Complete	
bandwidthTest	Failure	Complete	3.1.2
bicubicTexture	Failure	Complete	3.3.1, 3.3.3, 3.3.5
bilateralFilter	Failure	Complete	3.3.3, 3.3.5
binomialOptions	Partial	Complete	3.3.3
BlackScholes	Failure	Complete	3.1.2
boxFilter	Failure	Complete	3.3.3, 3.3.5
clock	Complete	Complete	
concurrentKernels	Failure	Complete	3.1.2
conjugateGradient	Failure	Complete	3.1.2
convolutionFFT2D	Partial	Complete	3.3.3
convolutionSeparable	Failure	Complete	3.1.2, 3.3.3
convolutionTexture	Partial	Complete	3.3.1, 3.3.3
cppIntegration	Failure	Complete	3.1.2, 3.3.3
dct8x8	Failure	Complete	3.1.2
deviceQuery	Failure	Complete	3.1.2
deviceQueryDrv	Failure	Complete	3.1.2
dwtHaar1D	Complete	Complete	
dxtc	Failure	Complete	3.1.2, 3.3.1, 3.3.3
eigenvalues	Partial	Complete	3.3.1, 3.3.3
fastWalshTransform	Complete	Complete	
FDTD3d	Failure	Complete	3.1.2, 3.3.3
fluidsGL	Failure	Complete	3.1.2, 3.3.5
FunctionPointers	Failure	Complete	3.1.2, 3.3.3, 3.3.5
histogram	Failure	Complete	3.1.2, 3.3.3
imageDenoising	Failure	Complete	3.1.2, 3.3.3, 3.3.5
Interval	Failure	Failure	3.1.2
lineOfSight	Failure	Complete	3.3.4
Mandelbrot	Failure	Complete	3.1.2, 3.3.1, 3.3.3, 3.3.5
marchingCubes	Failure	Complete	3.1.2, 3.3.1, 3.3.3, 3.3.5
matrixMul	Complete	Complete	
matrixMulDrv	Failure	Complete	3.1.2, 3.3.3
matrixMulDynlinkJIT	Failure	Complete	3.3.5
mergeSort	Partial	Complete	3.3.1, 3.3.3
MersenneTwister	Failure	Complete	3.1.2
MonteCarlo	Partial	Complete	3.3.1, 3.3.3
MonteCarloCURAND	Partial	Partial	3.1.2
MonteCarloMultiGPU	Partial	Complete	3.3.1, 3.3.3
nbody	Failure	Complete	3.1.2, 3.3.1, 3.3.3, 3.3.5
oceanFFT	Failure	Complete	3.1.2, 3.3.3, 3.3.5

Table A.6: CUDA SDK Sample Translation Robustness

Application	Original CU2CL Prototype	Current CU2CL Prototype	Related Repairs
particles	Failure	Complete	3.1.2, 3.3.3, 3.3.5
postProcessGL	Failure	Complete	3.1.2, 3.3.3, 3.3.5
ptxjit	Failure	Complete	3.1.2
quasirandomGenerator	Failure	Complete	3.1.2, 3.3.3
radixSort	Failure	Complete	3.1.2, 3.3.1, 3.3.3
randomFog	Failure	Complete	3.1.2, 3.3.3
recursiveGaussian	Failure	Complete	3.1.2, 3.3.3, 3.3.5
reduction	Failure	Complete	3.1.2, 3.3.1, 3.3.3
scalarProd	Complete	Complete	
scan	Failure	Complete	3.1.2, 3.3.3
simpleAtomicIntrinsics	Complete	Complete	
simpleCUBLAS	Complete	Complete	
simpleCUFFT	Failure	Complete	3.3.4
simpleGL	Failure	Complete	3.1.2, 3.3.3, 3.3.5
simpleMPI	Failure	Complete	3.1.2, 3.3.3
simpleMultiCopy	Failure	Complete	3.1.2
simpleMultiGPU	Partial	Complete	3.3.3
simplePitchLinearTexture	Complete	Complete	
simplePrintf	Failure	Complete	3.3.1, 3.3.3
simpleStreams	Complete	Complete	
simpleSurfaceWrite	Complete	Complete	
simpleTemplates	Failure	Complete	3.3.1
simpleTexture	Complete	Complete	
simpleTexture3D	Failure	Complete	3.1.2, 3.3.3, 3.3.5
simpleTextureDrv	Failure	Complete	3.1.2, 3.3.3
simpleVoteIntrinsics	Complete	Complete	
simpleZeroCopy	Complete	Complete	
smokeParticles	Failure	Complete	3.1.2, 3.3.3, 3.3.5
SobelFilter	Failure	Complete	3.1.2, 3.3.3, 3.3.5
SobolQRNG	Failure	Complete	3.1.2, 3.3.3
sortingNetworks	Failure	Complete	3.1.2, 3.3.3
template	Complete	Complete	
threadFenceReduction	Failure	Complete	3.3.1
threadMigration	Failure	Complete	3.1.2, 3.3.3
transpose	Failure	Complete	3.1.2, 3.3.2
vectorAdd	Complete	Complete	
vectorAddDrv	Failure	Complete	3.1.2, 3.3.3
volumeRender	Failure	Complete	3.1.2, 3.3.3, 3.3.5

Table A.7: CUDA SDK Sample Translation Robustness (cont.)

Bibliography

[1] "clang" C Language Family Frontend for LLVM, http://clang.llvm.org/. [Online; Accessed March 11, 2013].

[2] R. Anandakrishnan, T. R. Scogland, A. T. Fenley, J. C. Gordon, W. chun Feng, and A. V. Onufriev, "Accelerating Electrostatic Surface Potential Calculation with Multi-Scale Approximation on Graphics Processing Units," *Journal of Molecular Graphics and Modelling*, vol. 28, no. 8, pp. 904 – 910, 2010.

[3] B. A. Bauer, J. E. Davis, M. Taufer, and S. Patel, "Molecular Dynamics Simulations of Aqueous Ions at the LiquidVapor Interface Accelerated using Graphics Processors," *Journal of Computational Chemistry*, vol. 32, no. 3, pp. 375–385, 2011.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," in *ACM SIGGRAPH 2004 Papers*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004, pp. 777–786, http://doi.acm.org/10.1145/1186562.1015800. [Online].

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct., pp. 44–54.

[6] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[7] M. Daga, T. Scogland, and W. chun Feng, "Architecture-Aware Mapping and Optimization on a 1600-Core GPU," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, Dec., pp. 316–323.

[8] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures," in *Proceedings of the 2008 ACM/IEEE*

conference on Supercomputing, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 4:1–4:12, http://dl.acm.org/citation.cfm?id=1413370.1413375. [Online].

[9] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU binaries to Tiered SIMD Architectures with Ocelot," Tech. Rep., 2009.

[10] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 353–364.

[11] R. Domínguez, D. Schaa, and D. Kaeli, "Caracal: Dynamic Translation of Runtime Environments for GPUs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 5:1–5:7.

[12] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming," *Parallel Computing*, vol. 38, no. 8, pp. 391 – 407, 2012.

[13] M. Ernst, G. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," *Software Engineering, IEEE Transactions on*, vol. 28, no. 12, pp. 1146 – 1170, dec 2002.

[14] A. T. Fenley, J. C. Gordon, and A. Onufriev, "An Analytical Approach to Computing Biomolecular Electrostatic Potential. I. Derivation and Analysis," *The Journal of Chemical Physics*, vol. 129, no. 7, p. 075101, 2008.

[15] N. Ganesan, M. Taufer, B. Bauer, and S. Patel, "FENZI: GPU-Enabled Molecular Dynamics Simulations of Large Membrane Regions Based on the CHARMM Force Field and PME," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May, pp. 472–480.

[16] N. Ganesan, B. A. Bauer, T. R. Lucas, S. Patel, and M. Taufer, "Structural, dynamic, and electrostatic properties of fully hydrated DMPC bilayers from molecular dynamics simulations accelerated with graphical processing units (GPUs)," *Journal of Computational Chemistry*, vol. 32, no. 14, pp. 2958–2973, 2011.

[17] A. Garrido and R. Johnson, "Refactoring C with Conditional Compilation," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, Oct., pp. 323–326.

[18] Z. Guo, E. Zhang, and X. Shen, "Correctly Treating Synchronizations in Compiling Fine-Grained SPMD-Threaded Programs for CPU," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, Oct., pp. 310–319.

[19] M. Harvey and G. D. Fabritiis, "Swan: A tool for porting CUDA programs to OpenCL," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093 – 1099, 2011.

[20] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray, "CUDACL: A tool for CUDA and OpenCL programmers," in *High Performance Computing (HiPC), 2010 International Conference on*, Dec., pp. 1–11.

[21] Khronos Group. The OpenCL C++ Wrapper API, http://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf. [Online; Accessed March 11, 2013].

[22] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, March, pp. 75–86.

[23] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 101–110.

[24] Y. Liu, D. Maskell, and B. Schmidt, "CUDASW++: Optimizing Smith-Waterman Sequence Database Searches for CUDA-Enabled Graphics Processing Units," *BMC Research Notes*, vol. 2, no. 1, pp. 1–10, 2009, http://dx.doi.org/10.1186/1756-0500-2-73. [Online].

[25] G. Martinez, M. Gardner, and W. chun Feng, "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, Dec., pp. 300–307.

[26] G. Martinez, "CU2CL: A CUDA-to-OpenCL Translator for Multi-and Many-Core Architectures," Master's thesis, Virginia Polytechnic Institute and State University, 2011.

[27] D. Nandakumar, "Automatic Translation of CUDA to OpenCL and Comparison of Performance Optimizations on GPUs," Master's thesis, University of Illinois at Urbana-Champaign, 2011.

[28] NVIDIA. CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Online; Accessed March 11, 2013].

[29] NVIDIA. CUDA Parallel Computing Platform, http://www.nvidia.com/object/cuda_home_new.html. [Online; Accessed February 15, 2013].

[30] NVIDIA. CUDA Toolkit Archive, https://developer.nvidia.com/cuda-toolkit-archive. [Online; Accessed March 11, 2013].

[31] OpenACC. OpenACC Home, http://www.openacc-standard.org/. [Online; Accessed March 11, 2013].

[32] C. Reano, A. Pena, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Ortı, "CU2rCU: a CUDA-to-rCUDA Converter," *Proceedings of the 13th Jornadas de Paralelismo*, vol. 1, 2012.

[33] R. Reyes, I. López-Rodríguez, J. Fumero, and F. de Sande, "accULL: An OpenACC Implementation with CUDA and OpenCL Support," *Euro-Par 2012 - 18th International European Conference on Parallel and Distributed Computing*, pp. 871–882, 2012.

[34] B. Rosenkraenzer. Bug 9526 - clang++ (2.9 branch) can't deal with GNU libstdc++ 4.6.0 headers, http://llvm.org/bugs/show_bug.cgi?id=9526. [Online; Accessed March 11, 2013].

[35] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 73–82, http://doi.acm.org/10.1145/1345206.1345220. [Online].

[36] P. Sathre, M. Gardner, and W.-C. Feng, "Lost in Translation: Challenges in Automating CUDA-to-OpenCL Translation," *2012 41st International Conference on Parallel Processing Workshops*, vol. 0, pp. 89–96, 2012.

[37] D. Spinellis, "Global Analysis and Transformations in Preprocessed Languages," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 1019–1030, Nov. 2003.

[38] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "Languages and Compilers for Parallel Computing," J. N. Amaral, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, ch. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pp. 16–30.

[39] B. Stroustrup, "History of Programming Languages—II," T. J. Bergin, Jr. and R. G. Gibson, Jr., Eds. New York, NY, USA: ACM, 1996, ch. A History of C++: 1979–1991, pp. 699–769.

[40] R. Weber, A. Gothandaraman, R. Hinde, and G. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 58 –68, jan. 2011.

[41] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–14.

[42] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse MatrixVector Multiplication on Emerging Multicore Platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178 – 194, 2009, revolutionary Technologies

for Acceleration of Emerging Petascale Applications, http://www.sciencedirect.com/science/article/pii/S0167819108001403. [Online].

[43] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 86–97, http://doi.acm.org/10.1145/1806596.1806606. [Online].

[44] D. Yudanov, M. Shaaban, R. Melton, and L. Reznik, "GPU-Based Simulation of Spiking Neural Networks with Real-Time Performance & High Accuracy," in *Neural Networks (IJCNN), The 2010 International Joint Conference on*, July, pp. 1–8.