CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures

Gabriel E. Martinez Arroyo

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
Adrian Sandu

July 14th, 2011
Blacksburg, Virginia

Keywords: Source Translation, Clang, CUDA, OpenCL,
GPU, Compilers

© Copyright 2011, Gabriel E. Martinez Arroyo

CU2CL: A CUDA-to-OpenCL Translator for
Multi- and Many-Core Architectures

Gabriel E. Martinez Arroyo

ABSTRACT

The use of graphics processing units (GPUs) in high-performance parallel computing continues to steadily become more prevalent, often as part of a heterogeneous system. For years, CUDA has been the de facto programming environment for nearly all general-purpose GPU (GPGPU) applications. In spite of this, the framework is available only on NVIDIA GPUs, traditionally requiring reimplementation in other frameworks in order to utilize additional multi- or many-core devices. On the other hand, OpenCL provides an open and vendor-neutral programming environment and run-time system. With implementations available for CPUs, GPUs, and other types of accelerators, OpenCL therefore holds the promise of a "write once, run anywhere" ecosystem for heterogeneous computing.

Given the many similarities between CUDA and OpenCL, manually porting a CUDA application to OpenCL is almost straightforward, albeit tedious and error-prone. In response to this issue, we created CU2CL, an automated CUDA-to-OpenCL source-to-source translator that possesses a novel design and clever reuse of the Clang compiler framework. Currently, the CU2CL translator covers the primary constructs found in the CUDA Runtime API, and we have successfully translated several applications from the CUDA SDK and Rodinia benchmark suite. CU2CL's translation times are reasonable, allowing for many applications to be translated at once. The number of manual changes required after executing our translator on CUDA source is minimal, with some compiling and working with no changes at all. The performance of our automatically translated applications via CU2CL is on par with their manually ported counterparts.

This work was supported in part by NSF grant IIP-0804155 and an AMD Research Faculty Fellowship Award.

Acknowledgments

Getting to this stage in my life has been a long and difficult journey, all of which would not have been possible without the aid of several people. First, I would like to thank my parents for not only providing for my physical needs, but also for encouraging my interests and curiosity.

I would also like to thank Wu and Mark for providing so much direction and feedback through the years. There were always plenty of opportunities available, which both of them encouraged and (usually) gently pushed me to take advantage of.

And last, but certainly not least, I would like to thank all of the friends I have made throughout my years at Virginia Tech. Without them, I would not be who I am today.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Related Work 5

1.3 Contributions 7

1.4 Document Organization 9

2 Overview of GPGPU Frameworks **11**

2.1 CUDA 12

2.2 OpenCL 15

3 Source-to-Source Translation **19**

3.1 CUDA Data Structures 21

3.2 CUDA Device Memory 22

3.3 Runtime API Procedures 23

3.3.1 Thread Management 24

3.3.2 Device Management 24

3.3.3 Stream Management 26

3.3.4 Event Management 27

3.3.5 Memory Management 28

3.4 Device Procedures 30

3.5 Kernel Calls 33

3.6 OpenCL Limitations 34

4 The Design and Implementation of CU2CL **36**

4.1 Approach 36

4.2 Architecture 38

4.3 AST-Based, String-Based Rewriting 40

4.3.1 Common Patterns 41

4.3.2 Recursively Rewriting Expression 43

4.3.3 Rewriting Includes 44

4.4 Challenges 47

5 Evaluation **49**

5.1 Performance of the CU2CL Translator 49

5.2 Performance of CU2CL-Translated Applications 51

5.3 Coverage of the CU2CL Translator 55

6 Conclusion **58**

6.1 Summary 58

6.2 Future Work 59

Bibliography **62**

List of Figures

1.1 Overview of CU2CL's translation process 8

2.1 Overview of the CUDA and OpenCL programming models. Items with multiple labels present the CUDA terminology, followed by the OpenCL terminology. 13

2.2 Overview of the CUDA and OpenCL memory models. Items with multiple labels present the CUDA terminology, followed by the OpenCL terminology. 14

2.3 OpenCL code to initialize a GPU device and compile the kernel code that is to be executed, neither of which is required when using the CUDA runtime API 18

3.1 Example of a translating a CUDA kernel call. 34

3.2 Example of a `struct` with nested device pointers. 35

4.1 Diagram of the Clang libraries and their dependencies. 38

4.2 High-level view of CU2CL's architecture as a Clang plug-in. 39

4.3 Example of rewriting a CUDA API call which expects a pointer to an OpenCL call which does not. 43

4.4 Example of rewriting an expression and its subexpressions. 44

4.5 Algorithm detailing how expressions are recursively rewritten. 45

4.6 Example of rewriting an `#include` directive. 46

5.1 Total time for CU2CL to translate an application, including the time for Clang to parse the source and generate the AST, versus the number of preprocessed lines in the original CUDA source. 51

5.2 Time spent in CU2CL translating an application versus the number of lines in the original CUDA source. 52

List of Tables

3.1 Common CUDA data structures and their OpenCL equivalents 21

3.2 CUDA API modules and their OpenCL equivalents. 23

3.3 CUDA attribute qualifiers and their OpenCL equivalents 32

3.4 Common CUDA kernel built-in functions and variables and their equivalents in OpenCL 32

5.1 CU2CL's translation times for applications from the CUDA SDK and Rodinia and their original lines of code. 50

5.2 Run times of the four CUDA applications and both OpenCL ports (including percent differences with respect to the CUDA times) on an NVIDIA GTX 280. 54

5.3 Top 15 CUDA calls in the CUDA 3.2 SDK. 56

5.4 Top 15 CUDA calls in the Rodinia benchmark suite. 56

5.5 CU2CL's automatic translation coverage of a range of applications. 57

Chapter 1

Introduction

Until recently, graphics processing units (GPUs) were used exclusively for graphics workloads, such as those found in the computer gaming and visualization markets. In the past decade, though, the scientific community discovered that some of their large computations could take advantage of the powerful and highly-parallel capabilities of commodity GPUs. The first applications that used GPUs in this manner were forced to map their data and algorithms to the models in graphics APIs (e.g. OpenGL, DirectX), as they provided the only method of executing code on GPUs [24]. NVIDIA released the CUDA (Compute Unified Device Architecture) toolkit [21], a proprietary framework for executing general GPU applications written in a C-like programming language. Inevitably, this opened the floodgates of general-purpose GPU (GPGPU) computing; now even those inexperienced with GPUs and graphics programming can harness the power of GPUs.

1.1 Motivation

Since its release, CUDA has only been available on NVIDIA GPUs, requiring the use of other platform-specific frameworks when programming AMD GPUs and other multi-core compute devices. This led Apple, who wished to write GPU-accelerated applications once and yet execute them on several kinds of GPUs, to create OpenCL. OpenCL is a platform-agnostic framework for executing single instruction, multiple data (SIMD) tasks along with more general parallel workloads on accelerators, including GPUs, CPUs, or any other device with an OpenCL implementation. The OpenCL specification [12], now maintained by the Khronos Group, provides a C API for querying, allocating memory, and executing kernels on OpenCL-capable devices. Furthermore, it defines a variant of C99 for writing device code. As an open standard, OpenCL fosters a vendor-neutral environment wherein multiple heterogeneous devices may be used at once from the same application using the same language and framework.

Though still a relatively new language, OpenCL already has several of implementations available from Intel (x86 CPUs), AMD (x86 CPUs and AMD GPUs), NVIDIA (NVIDIA GPUs), and even IBM (IBM POWER line, including Cell processor). As a result, developers can write platform-independent applications that can to take advantage of any of these compute devices. This is of particular importance to scientists who often want to simply write an application once and not have to port it when moving to another compute platform. OpenCL enables such a scenario.

Both CUDA and OpenCL are seeing use in several computing applications, even outside of scientific and high-performance ones. Many programming languages have bindings to both languages. Native support for CUDA and OpenCL can be found in some mathematical software, like Mathematica. Video games have been using these compute frameworks for performing real-time physics through engines such as PhysX and Bullet. Even several benchmark suites now provide CUDA implementations of their applications—the Rodinia benchmark suite [5], the Parboil benchmark suite [3], and the Scalable HeterOgeneous Computing suite (SHOC) [8], to name a few. Unlike the Rodinia and Parboil benchmarks, SHOC, from Oak Ridge National Lab, also provides benchmarks in OpenCL. It is one of the first works among a few others [11, 14] that have started using OpenCL for evaluating GPUs. Even so, the adoption of OpenCL has been slow so far.

There are issues that have hindered the wide use of OpenCL in applications. First, OpenCL provides a lower-level API than commonly used CUDA API, requiring more time and effort in order to learn how to set up a device and execute kernels on it. Of greater impact, however, has been CUDA's established presence in the GPGPU market, which has made it the de facto GPGPU programming environment. Not surprisingly, there are a much greater number of CUDA applications available than those implemented in OpenCL. In order to drive adoption of OpenCL, applications can be ported from CUDA, a task that is mostly straightforward given OpenCL's GPU origins. Nevertheless, performing the process by hand can be tedious and is often error-prone. Although there has been some work to alleviate

the issue [13], only recently was there an effort [19] to automate the process in order to free developers from the burden of the more mechanical portions of translation.

We believe that an effective translation can be automatically achieved by means of a source-to-source translator from CUDA to OpenCL. Such a translator would insert OpenCL code to initialize an appropriate CUDA-like environment. In addition, a translator based on a powerful source-to-source translation framework, could handle the splitting of CUDA C's host and device code, which do not normally reside in the same source file in OpenCL. Finally, given that the OpenCL standard is much newer and meant to support many types of compute devices, there are several of CUDA's GPU-centric features that are not yet supported. A source-to-source translator should be able to notify the user when it cannot handle these aspects of a CUDA program, much like how a traditional compiler emits warnings, allowing the user to focus manual translation efforts on the most interesting parts of the code.

Of particular importance to this work is seeing the results of this project actively adopted and used by the CUDA and OpenCL communities. Therefore, after exploring several possible options, we have decided to base our source-to-source translator's implementation on the Clang framework [1]. The reasoning behind our decision will be discussed in greater detail in Chapter 4.

1.2 Related Work

The most relevant related work in this area has been a recent source-to-source translator, introduced in the last two months. CUDAtoOpenCL [19] is a translator based on the Cetus source-to-source framework [18]—a Java-based framework that provides a class hierarchy that represents a program's abstract syntax tree (AST), allowing for easy traversal and modification of the program. The tool was developed as part of a Master's thesis. CUDA-toOpenCL's code has not been released yet, so we have not been able to perform an analysis of the translator. The authors state that CUDAtoOpenCL supports the basic features found in CUDA: allocating and copying device memory, making use of constant device memory, and executing kernels. They also point out advanced features in their translator, such as the ability to propagate the change of a device pointer type to an OpenCL device memory structure throughout a program via control flow analysis. However, it appears that support for other, more advanced, parts of CUDA, like the Stream and Event modules, is lacking.

Two other previous source-to-source translators which translate to or from CUDA have also been based on the Cetus framework. The first is OpenMP to GPGPU [17], that allows programmers to use the simpler and well-known OpenMP annotations for CPU framework to program GPUPUs with CUDA. The focus of the work was to determine the portions of an OpenMP application that could be offloaded to the GPU, generating CUDA code, and then performing back end optimizations to allow for better performance. The other work MCUDA [23], which instead goes from OpenMP to GPGPU, is another source-to-source translator that instead translates CUDA to multi-threaded CPU code. The authors

sought to make use of the CUDA programming model to make it easier to write SIMD CPU programs. In doing so, they created a CUDA parser for Cetus, which the CUDAtoOpenCL translator also leveraged.

Closer to our goal than the previous two works, Swan [13] is a tool made to ease the transition between OpenCL and CUDA. Note, however, that Swan is not actually a source-to-source translator like the above tools; instead it provides a higher-level library that abstracts both the CUDA and OpenCL APIs. In this way an application makes calls to Swan and allows the library to take care of the details in mapping them to CUDA or OpenCL. Unfortunately, the Swan API is currently quite limited, as it only abstracts a few features in common between CUDA and OpenCL. In addition, it requires applications to be ported to use its API, leaving developers to do most of the porting work. On the other hand, it does provide a simple Perl script that can automatically translate some kernel code by essentially performing a regular expression-based "search and replace" operation on the source.

Another project of interest is Ocelot [9]. It provides a PTX-to-LLVM translator along with a run-time system that can decide whether to execute the PTX [4] on a GPU or on a CPU after just-in-time (JIT) compilation to LLVM [16]. In this regard it is similar to MCUDA as it allows for CUDA kernels to be run on CPUs, but it takes the approach of performing translations on lower-level bytecodes. This is a typical approach for modern compilers, converting a higher-level language to some intermediate representation and then compiling that to the target architecture. In the case of GPGPUs and other OpenCL-capable devices, there are multiple intermediate languages that can be targeted, such as NVIDIA PTX,

AMD IL, and LLVM, to name a few. This works against Ocelot as separate back ends must be implemented in order to execute CUDA applications on other devices. This is seen in more recent work titled Caracal [10], which implements the PTX to IL back end. A CUDA to OpenCL translator can instead rely on the vendor-specific OpenCL implementations to simply handle compiling the code to the proper bytecode. As a result, CUDA applications, once successfully ported, will be automatically available on several architectures.

A recent source-to-source translator [28], based on the Cetus framework, as the others have been, focuses on optimizing CUDA source codes. The optimizing compiler attempts to generate new code based on the original that has better memory access patterns as well as the best-performing thread and block configuration for kernels. It accomplishes these via thread and thread-block merging, alongside generating and testing multiple variations of the same program. This GPGPU optimizing compiler is of particular interest for our future work, wherein we wish to generate OpenCL code optimized for specific device architectures.

1.3 Contributions

In this thesis, we propose CU2CL, an automated CUDA-to-OpenCL source-to-source translator. As shown in the high-level diagram in Figure 1.1, CU2CL takes as input an application's CUDA source files and rewrites them into semantically-equivalent OpenCL host and kernel files. In this process, it handles adding all OpenCL "boilerplate" code necessary to set up the compute environment—platforms, devices, contexts, command queues, etc. Currently

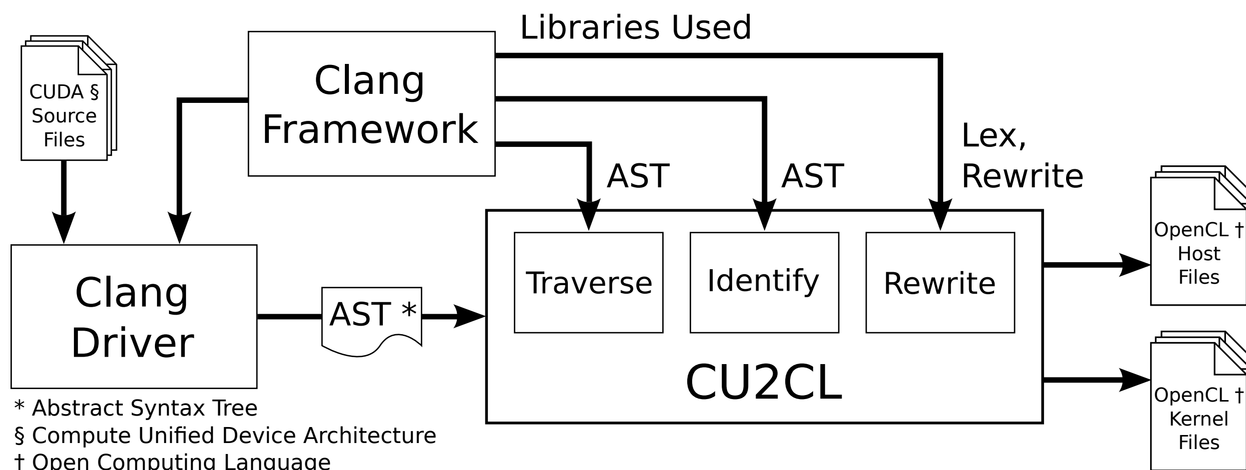

Figure 1.1: Overview of CU2CL's translation process

the most-used features in CUDA are automatically translated, with a framework in place to handle larger subsets of CUDA in the future. The eventual goal of this work is to provide a robust translator that will automatically handle a large majority of CUDA applications and generate maintainable OpenCL code with little to no manual porting effort. Additionally, we hope to leverage the results to create a framework that will allow for (possibly user-supplied) optimization passes that can transform CUDA kernels into efficient OpenCL for different compute devices. As previously shown [2, 7], OpenCL kernels must be optimized for the different device architectures. CU2CL is simply the first step in this process.

We have implemented CU2CL as a Clang [1] plug-in, allowing us to take advantage of a powerful, production-quality compiler framework. Although a recent project with only a few years of work behind it, Clang can already compile C, C++, Objective-C, and Objective-C++ into the LLVM [16] intermediate language; as a result, the compiler is already seeing production use at Apple. More importantly, Clang provides several libraries for performing

source-level transformations. By leveraging these tools, we have created a robust CUDA-to-OpenCL translator in less than 2000 source lines of code (SLOC).

This paper makes the following contributions:

- We present a framework for the automatic translation from CUDA to OpenCL. We also include an in-depth procedure of how to transform CUDA to OpenCL.
- In creating a source-to-source translator based on the Clang framework, we demonstrate general insights that may be used when designing source-level tools within the framework (which likely also apply outside of Clang). Included are (1) *common patterns* that arise when performing the translations, (2) a technique for *recursively rewriting expressions*, and (3) a process for *rewriting #includes*.
- We have evaluated a prototype of CU2CL with respect to translation speed, performance of translated applications compared to the original CUDA and manually ported codes, and coverage of sample CUDA applications from the CUDA SDK and Rodinia benchmark suite.

1.4 Document Organization

The remainder of this thesis is organized as follows: Chapter 2 gives an overview of the CUDA and OpenCL frameworks, focusing on how their similarities and differences influence the translation effort. Chapter 3 presents an in-depth look into the details of the CUDA to

OpenCL translation process. In Chapter 4 we discuss CU2CL's general design and implementation as a Clang-based source-to-source translator. There we also present general strategies we have identified that may be used by other source-to-source translators, In Chapter 5, an evaluation of CU2CL's performance during translation is given, along with the performance of a few automatically translated applications. We wrap the chapter up by discussing the CUDA code coverage of our translator. Lastly, we summarize our work and present possible future work in Chapter 6.

Chapter 2

Overview of GPGPU Frameworks

CUDA and OpenCL are both frameworks designed for general-purpose GPU computation. Furthermore, NVIDIA initially contributed a lot of CUDA's GPU features to OpenCL. As a result, the two are very similar—though OpenCL is now more general. Both have the notion of kernels that run on compute devices, threads that run in parallel within them, specifying a kernel's execution configuration at launch time, and managing device memory. However, as CUDA's compute devices are strictly GPUs, it includes many GPU-centric features in its APIs that are not found in OpenCL. OpenCL, on the other hand, provides a platform-agnostic framework. While, it supports a wider variety of compute devices, it requires extra work to initialize and utilize them. Given these circumstances, OpenCL applications will generally require more explicit device management.

In this chapter, we first examine version 3.2 of the CUDA APIs, the latest stable release. Next, we discuss the OpenCL 1.0 standard.[1] Our focus is on the differences and similarities between the two and how an automatic translator may map CUDA to OpenCL.

2.1 CUDA

CUDA (Compute Unified Device Architecture) is a programming model and environment that enables data-parallel, single instruction, multiple data (SIMD) computations to be offloaded onto a GPU. Users write device code in a C-like language which is run on the streaming multiprocessors of NVIDIA GPUs. Kernels functions, SIMD procedures launched from the host, are executed across a possibly multi-dimensional *grid* of *blocks*. In turn, each *block* contains numerous *threads* in another possibly multi-dimensional configuration. These configurations are specified by the host during a kernel invocation. Figure 2.1 illustrates the CUDA programming model.

CUDA's memory model has three separate general-use memory spaces: *global* memory, which resides off-chip and can be accessed by all threads in all blocks; *shared* memory, which is on-chip and available to threads in a block; and *registers*, which can only be accessed by the owning thread. In addition to these, there are two special-use memory spaces that provide faster memory operations: *constant* memory and *texture* memory. *Constant* memory is meant to be cached for fast reads, but is limited in size and does not support writes, as

[1] NVIDIA's OpenCL 1.1 implementation is not yet released so we restrict our attention to OpenCL 1.0 in order to execute on NVIDIA and AMD GPUs.

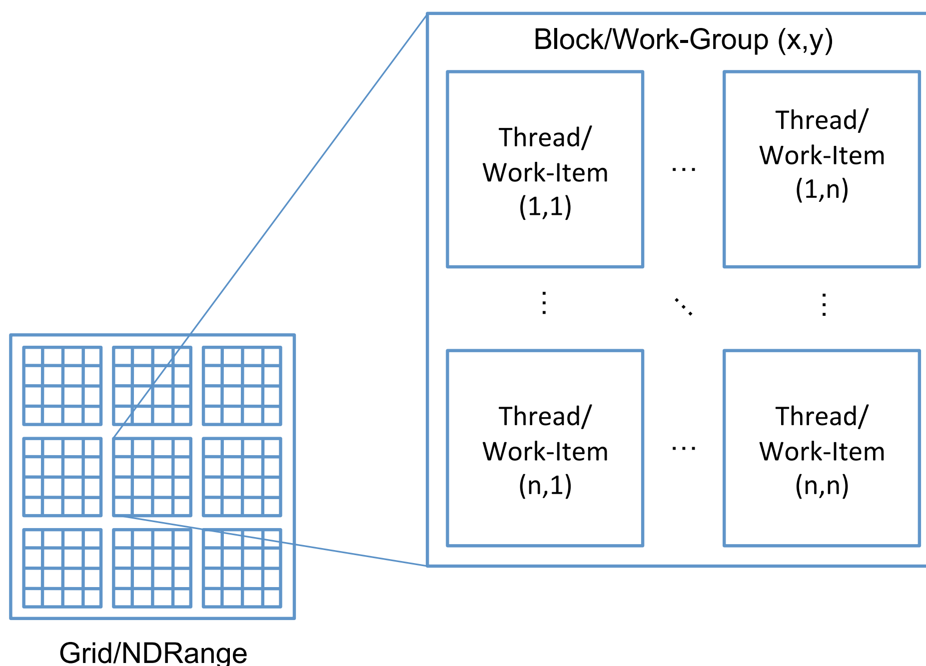

Figure 2.1: Overview of the CUDA and OpenCL programming models. Items with multiple labels present the CUDA terminology, followed by the OpenCL terminology.

its name implies. *Texture* memory allows for fast reads as well as fast writes, but is also rather limited in size. Furthermore, kernels must use special built-in functions in order to access data residing in the region. In general, device memory must be explicitly allocated through CUDA API calls on the host and is typically initialized by copying data over from host memory. Figure 2.2 shows how the general memory spaces are laid out in CUDA.

CUDA provides two different APIs that may be used, a low-level driver API and a high-level runtime API. The driver API is CUDA's lower-level API for programming GPGPUs. Similar to OpenCL (as detailed in Section 2.2), it requires that a GPU context be created for each GPU that is to be used, as nothing is implicitly done behind the scenes for the developer. Setting up and executing kernels requires several API calls to set arguments and to set the

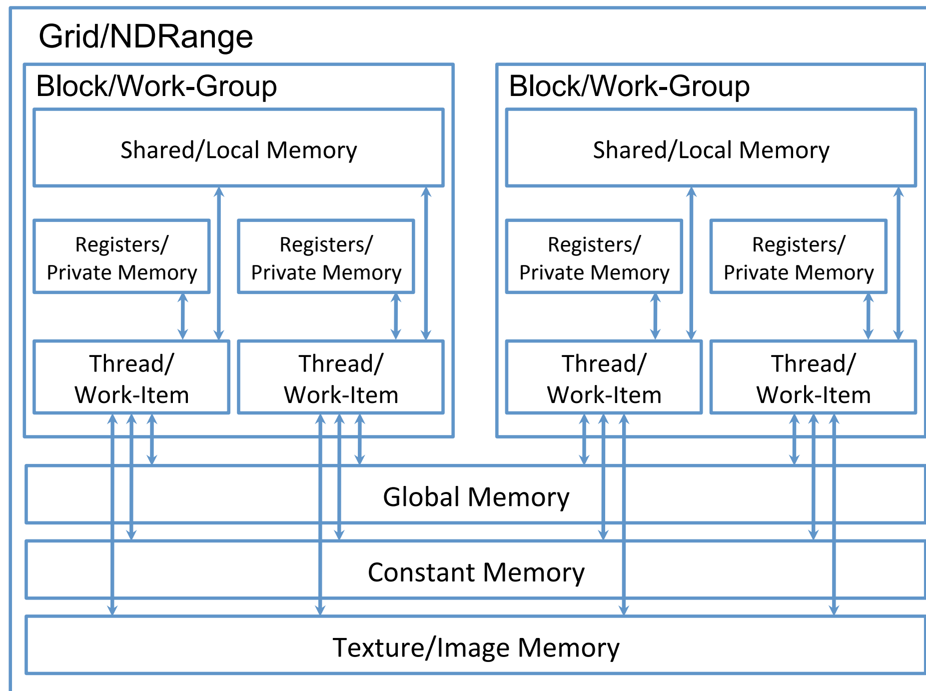

Figure 2.2: Overview of the CUDA and OpenCL memory models. Items with multiple labels present the CUDA terminology, followed by the OpenCL terminology.

dimensions and execution configuration of blocks and threads. On the other hand, the driver API allows for much finer control over GPU devices, and thus it is a requirement when using multiple devices. The API includes methods to explicitly load pre-compiled kernel modules at run time, which are the typical way that kernels are launched when using the driver API. These modules are pre-compiled, again using the NVIDIA compiler.

The CUDA runtime API is a higher-level API which abstracts many of the lower-level details found in the driver API, while simultaneously making reasonable assumptions behind the scenes. For example, the runtime will perform initialization of a GPU device on the first call to any CUDA runtime API method. If no particular GPU device is specified by the programmer, the runtime will simply choose the first GPU it finds on the system. Like the

driver API, the runtime API has methods for setting up and launching kernels, though they are not utilized often.

Both APIs may be used in applications written in standard C or in CUDA C, an extension to C that allows for the mixing of kernel code and host code in the same files. In CUDA C, device code and memory are distinguished by adding CUDA-specific function or variable qualifiers, respectively, to their declarations. In addition to these qualifiers, CUDA C extends the C language by introducing a special notation for launching kernels, effectively replacing multiple driver API calls to configure a kernel's arguments and launch configuration with one method invocation. In order to take advantage of these extensions, applications written using CUDA C must be compiled with NVIDIA's compiler, `nvcc`, which handles the C extensions and properly sorts host and kernel declarations.

Looking at code samples [5, 20], it is evident that many applications opt to use CUDA C along with the runtime API. Therefore, we have initially focused CU2CL's source-to-source translation on that combination.

2.2 OpenCL

The OpenCL standard is an open, vendor-neutral programming model and environment for executing general-purpose computations. Originally very much like CUDA given NVIDIA's contributions, it is now more general than CUDA, allowing for the use of arbitrary compute devices. Rather than targeting specific hardware, OpenCL provides abstract compute and

memory models that are mapped to real hardware through vendor-provided implementations. Like CUDA, OpenCL allows users to execute SIMD computations on GPUs, along with other compute devices supporting them. In addition, it provides the ability to run more general parallel workloads on traditional multi-core CPU devices. OpenCL expects kernel code to be written in a variant of C99. This code is typically dynamically compiled by a vendor-provided compiler through an OpenCL API call. Kernels are broken down into *work-groups*—similar to CUDA's *blocks*—each of which consist of *work-items*—akin to *threads* in CUDA. Figure 2.1 also shows OpenCL's programming model which is nearly identical to CUDA's although with different terminology.

Another similarity to CUDA can be seen in OpenCL's memory model in Figure 2.2. There are also three memory spaces that correspond to a memory space from CUDA: *global* memory, equivalent to CUDA's *global* memory, *local* memory, which resembles *shared* memory, and *private* memory, analogous to *registers* in CUDA. Support for *constant* and *image memory*—like CUDA's texture memory—also exists. These have similar limitations to CUDA's constant and texture memories, which we will cover in more depth in Chapter 3.

The OpenCL standard defines only one API, which is very similar to CUDA's driver API. Therefore, users have be aware of the low-level concerns and write some of the code that the CUDA runtime API handles automatically[2]. Furthermore, OpenCL adds the concept of OpenCL platforms—an abstraction of the set of installed vendor implementations—and device command queues (similar to CUDA streams) for sending commands to a particular

[2]OpenCL 1.1 defines C++ classes that abstract away several of the low-level issues.

device, on top of everything that the driver API has. On the other hand, kernels in OpenCL are very similar to kernels in CUDA, containing constructs that map almost one-to-one to the CUDA equivalents. Noteworthy exceptions are found in how image memory is accessed and the lack of some synchronization functions.

Figure 2.3 provides a simple example of what must be done in OpenCL to set up a device before executing code on it. It demonstrates OpenCL's low-level and verbose nature, as the CUDA runtime API implicitly performs this set up and thus gets by without any such code.

In the example, the first two statements get the first compute device from the first available OpenCL platform. Then, a context and command queue are created for the retrieved device. Next, the kernel source code stored in the file *kernel.cl* is read into a string. The kernel source string is used to build a new OpenCL program using the OpenCL platform's run time compiler. Finally, the kernels are created allowing for device code to be executed.

```
//Get the first GPU device and create a context
clGetPlatformIDs(1, &clPlatform, NULL);
clGetDeviceIDs(clPlatform, CL_DEVICE_TYPE_GPU, 1, &clDevice, NULL);
clContext = clCreateContext(NULL, 1, &clDevice, NULL, NULL, &errcode);
clCommands = clCreateCommandQueue(clContext, clDevice, 0, &errcode);

//Load OpenCL program source file
kernelFile = fopen("kernel.cl", "r");
fseek(kernelFile, 0, SEEK_END);
kernelLength = (size_t) ftell(kernelFile);
kernelSource = (char *) malloc(sizeof(char)*kernelLength);
rewind(kernelFile);
fread((void *) kernelSource, kernelLength, 1, kernelFile);
fclose(kernelFile);

//Build OpenCL program and create kernels
clProgram = clCreateProgramWithSource(clContext, 1,
                                      &kernelSource,
                                      &kernelLength,
                                      &errcode);
free(kernelSource);
clBuildProgram(clProgram, 1, &clDevice, NULL, NULL, NULL);
clKernel1 = clCreateKernel(clProgram, "kernel1", &errcode);
clKernel2 = clCreateKernel(clProgram, "kernel2", &errcode);
```

Figure 2.3: OpenCL code to initialize a GPU device and compile the kernel code that is to be executed, neither of which is required when using the CUDA runtime API

Chapter 3

Source-to-Source Translation

In this chapter, we outline the general strategy used to translate from CUDA to OpenCL.

As mentioned in the previous chapter, the CUDA runtime API is the most common API found in CUDA applications. Therefore we have chosen to focus our work to target it as the initial source of CUDA applications. Additionally, we decided to support CUDA C by default, as it is also very common and adds only a few extensions to standard C. For the time being CU2CL does not support the CUDA driver API. However, due to its low-level nature and similarity to the OpenCL API—many of its constructs have direct equivalents.

The CUDA Runtime API contains four primary constructs:

- **Data structures** – are the types and structs that are defined in CUDA and used by several of the runtime API procedures. They range from `dim3`s, which specify kernel configurations in three dimensions, to `textureReference`s, which are responsible for storing information about texture memory.

- **Device memory** – pointers may either be annotated with variable qualifiers or simply acquired through multiple API procedures. In the CUDA runtime API they are treated as normal pointers (e.g., `float *`).

- **Runtime API calls** – are the host-side CUDA runtime API procedure calls. Their main use is to set up and manage memory and code execution on GPU devices. They are broken up into several modules, of which we focus on the ones supporting general, non-graphical operations (e.g., `cudaMalloc`).

- **Device procedures** – also known as kernels, these are the procedures to be run on GPU devices. They are distinguished by the use of the CUDA function qualifiers `__global__` or `__device__`.

- **Kernel calls** – in CUDA C are similar to host method invocations, but differ in that they also provide the kernel execution configuration through a CUDA C extension (e.g., `kernel<<<blocks, threads>>>(args, ...)`).

In order to properly translate CUDA to OpenCL, each of these must in turn be translated into a semantically-equivalent construct—or set of constructs—in OpenCL. For the rest of this chapter, we will discuss how the automatic translations are performed for the currently supported subset of the CUDA runtime API.

3.1 CUDA Data Structures

The CUDA runtime API introduces a number of data structures. We list some commonly used ones in Table 3.1 along with their equivalent structures in OpenCL, if any.

Currently we have chosen to support `dim3`s and `cudaDeviceProp`s, given they are some of the most often used data structures in CUDA applications. In addition, we support `cudaStream_t` and `cudaEvent_t`. Of these, only `cudaDeviceProp`s lack an equivalent structure in OpenCL. This structure's fields are a GPU device's properties. For the sake of simplicity in translation, we opt to create an equivalent structure with the same fields. In OpenCL, then, multiple calls to `clGetDeviceInfo` may be used to fill the structure.

CUDA	**OpenCL**
Device pointers (e.g. `float *` created through `cudaMalloc`)	`cl_mem` created through `clCreateBuffer`
`dim3`	`size_t[3]`
`cudaDeviceProp`	*No direct equivalent*
`cudaStream_t`	`cl_command_queue`
`cudaEvent_t`	`cl_event`
`textureReference`	`cl_mem` created through `clCreateImage`
`cudaChannelFormatDesc`	`cl_image_format`
`surfaceReference`	*No direct equivalent*

Table 3.1: Common CUDA data structures and their OpenCL equivalents

Notably missing from the discussion so far are the vector types defined in the CUDA runtime API. The list is rather large, and direct equivalents in OpenCL exist for nearly all of them in both host and device code—the types have the same names save for the "cl_" prefix on the host vector types. The biggest exception in OpenCL 1.0 is the lack of vectors with three fields (e.g. float3).

3.2 CUDA Device Memory

Device memory in CUDA is separated into different regions, corresponding to the type of GPU memory. As our focus is on the non-graphical portions, we look at *global*, *shared*, and *constant* memory. In general, all of these are stored in traditional pointers in CUDA, therefore they will be translated into `cl_mem`s in OpenCL. Each type of memory has certain traits and limitations, which must be taken into account during the translation process.

Global memory is allocated through standard allocation procedures in CUDA. Unlike the other types of memory, pointers to it do not have to be qualified with the `__global__` or `__device__` attributes, making them a bit more difficult to identify and transform. The general strategy is to look for calls to global memory allocation procedures, such as `cudaMalloc`, and identify the pointer variable passed in as a pointer to *global* memory.

Constant memory arrays may only be declared at the top level in CUDA. Furthermore, they must either be given an initial size or an initial value; no dynamic allocation of *constant* memory is allowed. In OpenCL, *constant* memory is instead allocated as normal device memory is. It is marked as *constant* when passed in as a kernel parameter that has the `__constant` attribute. Finally, as they reside at the file level, CUDA device code assumes global access to them. Thus in OpenCL all pointers to *constant* memory must be passed in as arguments to kernel calls so that device code has access to them.

Shared memory in CUDA can be declared at the top level or within device code. In the first case, pointers to it can be treated in similar fashion to *constant* memory by passing them in

as arguments. In the latter case, the variable qualifier marking them as `__shared__` in CUDA can simply be rewritten to `__local` in OpenCL. One final case exists where *shared* memory is declared to be `extern`. In CUDA, this means that the memory is allocated dynamically when a kernel is invoked. In OpenCL, this can be emulated by allocating the device memory just before a kernel is executed and freeing it just after, making sure to pass in the device memory as an argument.

3.3 Runtime API Procedures

The CUDA runtime API is broken down into several modules, which provide the specific functionality of the CUDA framework. Table 3.2 shows the modules we are interested in, along with replacements found in the OpenCL API. The majority of the porting effort from CUDA to OpenCL occurs in rewriting these API calls to their OpenCL equivalents. As a result, proper automatic translation of the majority of the CUDA runtime API calls will greatly reduce any manual translation that must be done.

CUDA Module	**Sample Call**	**OpenCL Structure**
Thread	`cudaThreadSynchronize`	Contexts & Command Queues
Device	`cudaSetDevice`	Platforms & Devices
Stream	`cudaStreamSynchronize`	Command Queues
Event	`cudaEventRecord`	Events
Memory	`cudaMalloc`	Memory Objects

Table 3.2: CUDA API modules and their OpenCL equivalents.

3.3.1 Thread Management

The thread management API provides high-level methods to the underlying CUDA context. The most commonly used procedures are `cudaThreadExit` and `cudaThreadSynchronize`. The first asks the runtime to clean up the CUDA context, which translates to releasing the OpenCL context through `clReleaseContext`. As for the second function, it tells the runtime to block until all previously queued work on the current GPU device has been completed. In OpenCL, this is equivalent to invoking the `clFinish` method on global command queue that was created.

The CUDA runtime API also defines a few other procedures that allow a user to set and get resource limits and the host thread's cache configuration. The OpenCL API, though, does not provide any methods with similar functionality to these. Hence, CU2CL does not currently support the rest of this API, emitting a warning when an unsupported call is encountered.

3.3.2 Device Management

In the CUDA runtime API, set up and tear down of CUDA devices is done automatically. The user has little need to explicitly manage devices, but a direct consequence is that they also do not have as much control. By default, the first GPU device found on a system is used by the CUDA runtime API. Keeping with this assumption, OpenCL code to initialize a GPU device can be inserted at the start of an application's main method. This initialization

includes querying the system for an OpenCL platform, finding a GPU device in the platform, and then creating a context and command queue associated with the device. At the end of the same procedure, code is added to clean up the OpenCL constructs—specifically, the device's command queue and context are released. Several OpenCL API procedures require information about the system, such as a platform ID, device ID, context, or command queue, to be passed as parameters. In the CUDA runtime API, these are not made explicit, but are available globally to the underlying implementation. Since CUDA API calls may reside in many different locations in an application, we must declare the OpenCL constructs at the global level so as to be visible throughout the source file. The names of these global variables may clash with user-defined names, so we prepend "`__cu2cl_`" to each one.

There are a few CUDA runtime API calls that allow an application to query the available devices and to set the GPU device the current thread will use. Of these, the most commonly used are `cudaGetDevice`, `cudaGetDeviceCount`, `cudaGetDeviceProperties`, and `cudaSetDevice`. `cudaGetDevice` returns the ID of the device currently in use, therefore the call will be translated into a reference to CU2CL's global device ID. As the OpenCL device ID, context, and command queue are available globally, a call to `cudaGetDevice` will return the device ID currently in use. `cudaSetDevice` is replaced by OpenCL calls that release the current context, set the new device ID, and create a new context and command queue for the given device. `cudaGetDeviceCount` becomes a call to `clQueryPlatformInfo`, to get the number of devices from the OpenCL platform.

`cudaGetDeviceProperties` requires more work to translate. As mentioned earlier, there is no OpenCL equivalent of a `cudaDeviceProp`, hence we add in code that defines a similar `struct`. This newly defined data structure is populated by replacing calls to `cudaGetDeviceProperties` with calls to a new procedure we define that makes the necessary calls to `clGetDeviceInfo`. Not all the properties found in a `cudaDeviceProp` can be queried in OpenCL. If the CUDA source uses one of these properties, CU2CL must notify the user that the code will have to be translated by hand; currently this is done by emitting a warning during translation.

3.3.3 Stream Management

CUDA *streams* provide a mechanism for performing GPU operations asynchronous to other operations that may be executing. As a result, they function in a fashion very similar to *command queues* in OpenCL. In general, CUDA streams are turned into command queues, and the associated CUDA API calls are translated into equivalent OpenCL calls operating on command queues.

`cudaStreamCreate` creates a new CUDA stream, therefore it can simply be replaced by `clCreateCommandQueue`, while `cudaStreamDestroy` will become `clReleaseCommandQueue`. `cudaStreamSynchronize` forces the calling thread to block until the stream passed in has completed all previous operations. In OpenCL, this is equivalent to a `clFinish`. Likewise, `cudaStreamWaitEvent` becomes a call to `clEnqueueWaitForEvents`. The tougher one in this API is `cudaStreamQuery`. This function call queries the current status of the stream,

i.e. whether or not the stream has completed its work. In order to preserve the semantics of the call, several OpenCL API calls must be used, which is done by creating a procedure that does the following steps: First, a `clEnqueueMarker` is done, so as to get the event associated with the call. Next, the event created is queried for completion using a call to `clGetEventInfo`. We are only interested in the event's status, thus the event is released and the status returned.

3.3.4 Event Management

CUDA *events* provide a platform-independent way to perform time measurements. The API has methods to create and destroy events, in addition to methods that record an event—store the current time in the event—or that synchronizes on an event. While the API's primary facilities are to record times and compare them, it may also be used to synchronize by blocking until the work in a recorded stream has been finished.

OpenCL also has the concept of an event, which has similar goals. The way the two event mechanisms function are dissimilar enough that at first glance an automatic translation from CUDA events to OpenCL events appears difficult. This stems from the fact that CUDA events are explicitly created by the user through API calls, whereas OpenCL events are created anytime that an enqueue operation is performed on a command queue. However, OpenCL provides mechanisms that can bridge the disparity between the two, allowing OpenCL events to function as CUDA events do. By using the OpenCL API call `clEnqueueMarker` on a command queue, a new event will be made such that the call emulates a call to

`cudaEventRecord`. As a result, OpenCL events can be used as CUDA events are, to perform platform-independent timing. The event record call simply records the time after all the previous operations in the given CUDA stream have finished. Assuming time profiling has been enabled on the given command queue, the OpenCL event will also store the time.

Since OpenCL events are not explicitly created, calls to `cudaEventCreate` and `cudaEvent-CreateWithFlags` are simply removed during their translation to OpenCL. However, calls to `cudaEventDestroy` are not removed, but instead replaced with `clReleaseEvent`.

The final three procedures in the CUDA event API are `cudaEventQuery`, `cudaEvent-Synchronize`, and `cudaEventElapsedTime`. The first call, `cudaEventQuery`, requests the current status of the given event. This is equivalent to a call to `clGetEventInfo` for requesting the current status of the event. `cudaEventSynchronize`, which makes the caller wait for the given event to complete, can be replaced by a call to `clWaitForEvents`. Finally, when two events are compared using a `cudaEventElapsedTime`, one can compare the time between the end of the two OpenCL events to acquire an equivalent measurement. In CU2CL we have implemented this as a method that mimics `cudaEventElapsedTime`'s function signature in order to simplify the translation process.

3.3.5 Memory Management

There are dozens of procedures in the CUDA runtime API used for managing device and pinned host memory. These include various methods of allocating device memory and also copying memory to and from the different memory spaces on the device. We choose to

support the most prevalent and general of these API calls: `cudaMalloc`, `cudaFree`, `cudaMemcpy`, and `cudaMemset`. In the case of `cudaMalloc`, `cudaFree`, and `cudaMemcpy`, there are direct equivalents in OpenCL, namely `clCreateBuffer`, `clReleaseMemObject`, and `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` (for two different cases of `cudaMemcpy`). OpenCL lacks an equivalent of `cudaMemset`, requiring one of two possible workarounds. The first approach is to simply allocate host memory of the desired size and memory layout and then to copy it over to the given portion of device memory. This is far from ideal as it allocates unnecessary memory on the host and requires an extra copy to device memory, which becomes very expensive when transferring a large amount of data. Instead, we create a "memset" kernel that takes a `cl_mem`, a size, and an integer value for parameters—as expected in `cudaMemset`—and explicitly sets the memory region. There is a trade off, of course, as this method requires several OpenCL API calls to set up and execute a kernel. On the other hand, it does not use extra memory like the first and the cost of one kernel launch is small compared to a large copy operation.

Beyond translating CUDA API procedures, the main challenge in translating the device memory management API arises in identifying which declared variables correspond to device memory. In the CUDA runtime API, device memory addresses acquired from `cudaMalloc` are stored in traditional pointers. In OpenCL, however, device pointers are abstracted into `cl_mem` structs. Therefore, device pointer variables must be recognized when passed in as arguments to `cudaMalloc` and then their references tracked throughout an application's source code. The original declaration's type must be changed from a pointer of a specific type

(e.g. `float *`) to a `cl_mem`. In addition, procedures expecting a device pointer, identified through their use of one of these device pointer variables, must also have their parameter list modified so as to expect `cl_mems` instead. The exception are kernels, since the OpenCL runtime ensures that the device pointers are unwrapped and made directly available in device code.

3.4 Device Procedures

Device procedures, or kernels, in CUDA are annotated with either the `__global__` or `__device__` function qualifiers. This makes identifying kernel functions a relatively easy task. Along the same line, converting CUDA kernels to OpenCL is a simpler task than translating host code; most built-in functions and variables in the kernels have direct mappings in OpenCL and, consequently, could be mainly handled through a simple search and replace operation. However, CUDA kernels often present a few quirks that complicate the procedure.

In particular, as already mentioned, CUDA C allows for both host and kernel code to be mixed together in the same source file. From this emerge several complications when translating, since OpenCL kernels must be compiled separately from host code. Compilation of device code is done at run-time from a string, therefore the kernel sources can either be strings in host code files or read into strings from separate kernel files. We choose to do the latter in CU2CL, primarily because some C compilers can only handle strings of a certain length.

Another common problem, also supported by CU2CL, occurs when variables declared outside of a kernel function in CUDA are referenced from within the kernels. This is perfectly fine in CUDA, but in OpenCL kernels may only access their local variables and parameters. The typical approach CU2CL takes is to pass in these external variables as arguments to the kernels.

The basic translation of kernels (i.e., device code) requires the rewriting of several CUDA constructs. First, the function and memory space qualifiers are rewritten as specified in Table 3.3. One thing to note during this is that parameters of pointer types which lack a memory address space qualifier are assumed to be in global memory in CUDA. In OpenCL, the pointers must be explicitly tagged as residing in global memory and are prepended with the `__global` qualifier. Next, CUDA kernel built-ins, such as `gridDim` and `__syncthreads()`, must be translated to their OpenCL equivalents. Table 3.4 details the transformations of the most common CUDA kernel built-ins. There are many more built-in functions, including arithmetic functions and functions for retrieving data from special memory spaces like texture memory. Most of the mathematical functions also have direct OpenCL equivalents, though the applications we tested did not use them and thus, for the sake of brevity, we exclude listing them here.

Because the CUDA C language allows for the intermingling of host and device code, we split the input source files into their host code and OpenCL device code during translation. In order to handle this, two output files are created for each main input file and included CUDA header files: one for the host code and one for the kernels. By filtering on the

	CUDA	OpenCL
Function Qualifiers	`__device__`	*Not required in device code*
	`__global__`	`__kernel`
	`__host__`	*Not required in host code*
Variable Qualifiers	`__device__`	*Not required in device code*
	`__constant__`	`__constant`
	`__shared__`	`__local`

Table 3.3: CUDA attribute qualifiers and their OpenCL equivalents

CUDA	OpenCL
`gridDim.{x,y,z}`	`get_num_groups({0,1,2})`
`blockIdx.{x,y,z}`	`get_group_id({0,1,2})`
`blockDim.{x,y,z}`	`get_local_size({0,1,2})`
`threadIdx.{x,y,z}`	`get_local_id({0,1,2})`
`warpSize`	*No direct equivalent*
`__threadfence_block()`	`mem_fence(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE)`
`__threadfence()`	*No direct equivalent*
`__syncthreads()`	`barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE)`

Table 3.4: Common CUDA kernel built-in functions and variables and their equivalents in OpenCL

function qualifiers, the declarations in the original file can be copied to the correct output file. An exception occurs when a kernel function is annotated with the `__host__` qualifier, meaning that the function is callable by both device code and host code. In this case, the function is included in both the host file and the kernel file so that it will be available in both environments.

Finally, if a CUDA kernel references a variable declared outside of its local environment, then the function must be changed to expect a new parameter for that declaration, effectively making the variable local. The host code must also be modified to set the new kernel argument when the kernel function is launched.

3.5 Kernel Calls

The CUDA runtime API provides a syntactic extension to C that allows for concise kernel invocations. This is done by using the `<<<...,...>>>` notation before the parenthesized list of arguments, wherein the kernel launch configuration can be set all in one call to the kernel function. Inside of the angle brackets, the first slot holds grid size while the second contains the block size, each of type `dim3`. OpenCL, on the other hand, simply provides a C API and thus requires several API calls in place of the one CUDA runtime API kernel launch—similar to what is done in the CUDA driver API. The kernel arguments must be set, one at a time, using the `clSetKernelArg` function and the launch bounds are specified when the kernel is launched via the `clEnqueueNDRangeKernel` call. The `dim3`s in CUDA must be translated to `size_t[3]` arrays and pointers to them are passed to the `clEnqueue-NDRangeKernel` procedure. Furthermore, OpenCL expects to be passed the global work size, equivalent to the number of blocks multiplied by the number of threads for each dimension, and the local work size, equivalent to the number of threads per dimension. In order to do this, we create two global `size_t[3]` arrays which correspond to the global work size and the local work size. These values are always passed to the `clEnqueueNDRangeKernel` call. We copy the grid and block size values specified in the CUDA kernel call into the `size_t[3]` arrays. The local work size is first set to the block size `dim3` and then the global work size is set to the grid size `dim3` multiplied by the local work size. This is done for all dimensions.

In case of a constant integer expression or variable—values that are not `dim3`s—is passed in as the grid or block size, we set the first dimension of the global or local work size to that

```
//CUDA
kernel<<<16, 16>>>(globalArrPtr, num);

//OpenCL
clSetKernelArg(clKernel_kernel, 0, sizeof(cl_mem), &globalArrPtr);
clSetKernelArg(clKernel_kernel, 1, sizeof(int), &num);
localWorkSize[0] = 16;
globalWorkSize[0] = 16*localWorkSize[0];
clEnqueueNDRangeKernel(clCommandQueue, clKernel_kernel, 1, NULL,
                       globalWorkSize, localWorkSize, 0, NULL, NULL);
```

Figure 3.1: Example of a translating a CUDA kernel call.

value and set the other dimensions to one. If both parameters meet this criteria, then we can make use of OpenCL's ability to launch kernels of lower dimensions and simply set the number of dimensions one, as seen in Figure 3.1.

3.6 OpenCL Limitations

The OpenCL framework currently has a number of limitations that prevent the translation of several of CUDA's features. The most difficult to resolve at this time is OpenCL's lack of support for C++ device code, as the standard defines device code to be based on C99 (with a few extensions). Another particularly difficult issue to resolve stems from OpenCL's use of `cl_mem`s on the host for abstracting pointers to device memory. In CUDA, device memory is handled in both host and device code through direct pointers. This allows applications to allocate space in device memory for `struct`s that have pointers to device memory nested within them. Figure 3.2 shows such a scenario, wherein a `nested struct` is to be allocated

```
//CUDA
typedef struct {
    float *d_mem1;
    float *d_mem2;
} nested;
//Legal

//OpenCL
typedef struct {
    cl_mem d_mem1;
    cl_mem d_mem2;
} nested;
//Illegal!
```

Figure 3.2: Example of a `struct` with nested device pointers.

on the GPU. As OpenCL's `cl_mem`s are translated to pointers only when passed in as kernel arguments, there is no way to nest device pointers as in CUDA. Currently, such nested pointers will be detected and warnings emitted.

Chapter 4

The Design and Implementation of CU2CL

In this chapter, we outline the general design and implementation of CU2CL's source-to-source translator.

4.1 Approach

Several mechanisms for source-to-source translation are in common use—from simple tools that utilize regular expressions to find and replace strings in a program's source to more complex ones that leverage a full framework and parse a language into an abstract syntax tree (AST) and perform transformations at that level. In the related work, we discussed two source-to-source translators that go to or from CUDA, both of which opt for the latter

option and base their tools on the Cetus framework, a research-oriented, non-production framework.

Our project seeks to produce a useful tool that can be rapidly adopted by the CUDA and OpenCL communities. As such, while numerous frameworks and tools for source-to-source translation exist [6, 22, 27], we chose to explore a number of production-quality and widely-used, open-source compilers (e.g. gcc, Clang, Open64) to base CU2CL on. Of those, gcc and Clang have the largest communities behind them. We chose Clang [1], primarily for the following three reasons. First, though relatively young, Clang has a large and active community, with many new features and better quality every day (Apple even uses Clang as a production compiler in OS X). Second is Clang's design: instead of being a monolithic compiler binary like gcc, the Clang driver has been created from a set of compiler libraries in the Clang framework. The libraries, which provide lexical analysis, parsing, semantic analysis, and much more (see Figure 4.1), may be used independently to create other source-level tools. Finally, initial support for parsing the CUDA C extensions has been recently added to clang, thus we did not have to implement the functionality. Therefore, basing CU2CL on Clang should help it be accepted quicker.

As implicitly noted in Figure 4.2, CU2CL is a Clang plug-in that ties into the main driver, allowing Clang to handle parsing and AST generation, as during normal compilation. Afterwards, CU2CL takes over and walks the generated AST to perform the rewrites. Of particular interest in designing CU2CL were the AST, Basic, Frontend, Lex, Parse, and Rewrite libraries. These facilitate file management (Basic), AST traversal and retrieval of

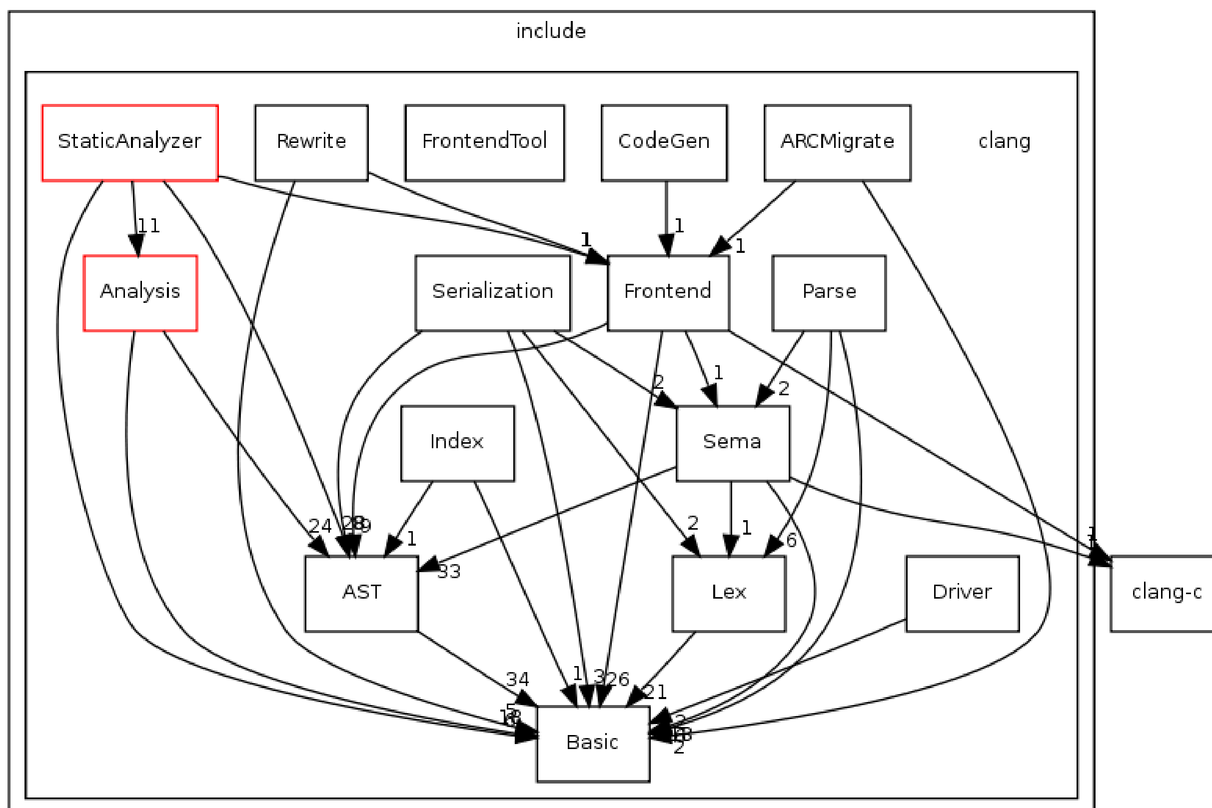

Figure 4.1: Diagram of the Clang libraries and their dependencies.

information from AST nodes (AST), plug-in interface and access to the compiler instance (Frontend), preprocessor access and token utilities (Lex), and the actual rewriting mechanism (Rewrite). By composing the libraries and classes included within each, we have created a robust prototype CUDA-to-OpenCL translator all in under 2000 source lines of code.

4.2 Architecture

In the Clang driver, once the AST has been created, an AST consumer is responsible for producing something from the AST. As a Clang plug-in, CU2CL provides an AST consumer

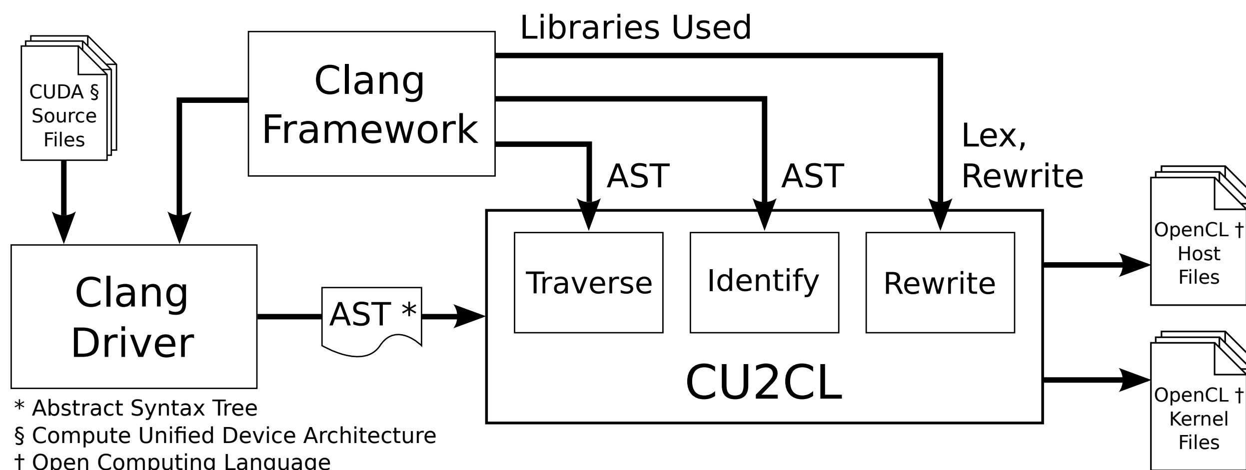

Figure 4.2: High-level view of CU2CL's architecture as a Clang plug-in.

that traverses the AST, searching for nodes of interest. While Clang's AST library provides several simple methods of traversing the tree, we have opted to manually traverse the AST in order to perform more complex translations. This traversal is done in a recursive descent fashion, using AST node iterators to recurse into each node's children. Figure 4.2 gives an overview of CU2CL's translation procedure: it traverses the AST for both host and device code, locates nodes of interest, and rewrites them.

The actual rewriting is done primarily through the use of Clang's Rewrite library. This library provides methods to insert, remove, and replace text in the original source files. It also has methods to retrieve the rewritten file by combining the original with the rewritten portions. While many traditional source-to-source translators build an AST, modify it, and then walk the new AST to produce the rewritten file, CU2CL uses the AST of the original source only to walk the program. Rewrites are done through strings; therefore, we say that our approach is *AST-driven and string-based*.

This approach is quite useful when translating CUDA to OpenCL as the two languages are based on C. This common ground between the two means that only the CUDA-specific constructs have to be translated to OpenCL. Given the size of typical applications that make use of CUDA, the scope [26] of our translations are very small. One of CU2CL's goals is to translate CUDA to OpenCL such that further development may continue in OpenCL. As a document's structure and comments are of vital importance to developers [25], leaving them intact is a requirement in CU2CL. By rewriting only the parts of interest and leaving everything else in the original source as it was we can retain most of the original structure and comments.

4.3 AST-Based, String-Based Rewriting

There are three areas of novelty in CU2CL's design as an AST-driven, string-based translator. First, we have identified *common patterns* that occur when performing source-to-source translation within the Clang framework. These are based around common structures in CUDA C and how to handle the task of identifying AST nodes of interest, as well as handling the rewriting of the original source that they represent. Second, we present a method for *recursively rewriting expressions* using Clang's Rewrite library. Being able to properly rewrite expressions and their subexpressions is not a trivial task with this string-based approach, thus we discuss our solution. Finally, we demonstrate how to *locate* and *rewrite* *#include*s by leveraging a powerful preprocessor, such as the one found in Clang's Lex li-

brary. All three of the above insights should aid future work in source-to-source translation based on the Clang framework, if not in more general uses.

4.3.1 Common Patterns

In translating CUDA constructs to OpenCL, some patterns are found to occur multiple times. CU2CL's design takes into account two primary patterns: rewriting CUDA types and processing CUDA API calls and their arguments. CUDA types may be found in many declarations and expressions, but the rules to identify and rewrite them are uniform with a few exceptions. The CUDA API functions share similar patterns in their arguments—what types are expected and how they are laid out—and also in their return types, as they all return an enumerated CUDA error value.

There are several places that CUDA-specific type declarations may occur. These include variable declarations, parameter declarations, type casts, and calls to `sizeof`, all of which may occur in both host and device code. Rewriting such types, as previously covered, can be generalized for both CUDA host code and device code. In the Clang framework, variable declarations carry with them information about what their full type is (including type qualifiers) as well as the source location of each part. The base type can be derived from the full type, which may then be inspected and rewritten accordingly. Types may be rewritten differently depending on where the type declaration occurred (e.g. host code, device code, kernel parameters, etc.). The generalizations to type rewriting can be applied in locations where there is overlap. For example, CUDA vector types (Appendix B.3 in

the CUDA C Programming Guide) may be found in any of those areas. OpenCL vector types have slightly different names depending on where they are found—i.e. `float4` versus `cl_float4`—but, for the most part, rewriting vector types can be combined. This pattern also extends to other CUDA types, like `dim3`s, which may declared anywhere in a CUDA C application.

For the purposes of CU2CL's source-to-source translation, it is preferable to generalize as much of the rewriting as possible. The most important pattern in CUDA API function calls occurs when a pointer to a data structure that is to be filled is passed in as an argument. The equivalent OpenCL API procedures instead return a new structure, as shown in Figure 4.3, therefore the dereferenced pointer must be retrieved from the argument expression. This can be done by traversing the expression and checking the types until the proper one is found. Then the subexpression with this evaluated type may be pulled out and used in the replacement OpenCL call. For the time being, CU2CL simply dereferences the pointer argument expression. The uniform enumerated CUDA error return type used by all the CUDA API calls can be used in rewriting the call's parent expressions. While CU2CL does not currently support rewriting the CUDA error type, knowledge of a CUDA call's possible returned error values in comparison to the equivalent OpenCL procedure will help in properly rewriting parent that use the returned error.

```
float *newDevPtr;
...
cudaMalloc((void **)&newDevPtr, size);
//Becomes
cl_mem newDevPtr;
...
newDevPtr = clCreateBuffer(clContext, CL_MEM_READ_WRITE, size, NULL, NULL);
```

Figure 4.3: Example of rewriting a CUDA API call which expects a pointer to an OpenCL call which does not.

4.3.2 Recursively Rewriting Expression

In performing string-based rewriting using Clang, several complications arise. Of these, being able to properly rewrite expressions and their subexpressions is of great importance. For example, when rewriting a kernel, one may encounter an expression such as the one in Figure 4.4. In order for the outer expression, `__powf`, to be rewritten, the argument expressions should be processed first. In the example, the arguments are rewritten by replacing references to the CUDA built-in variables with calls to OpenCL built-in functions. Then, the new strings are used in rewriting the top-level expression as a whole. Rewriting expressions in this recursive manner allows for nearly all expressions in CUDA to be rewritten without the need for special cases.

Using the Clang framework, we accomplish recursive expression rewriting through the Rewrite library. With each expression that is being rewritten, we associate a Rewriter object to facilitate easy string-based rewriting. Expressions are associated with the source range of text that it represents. Pseudocode for the algorithm is given in Figure 4.5. When an expression that is not interesting—one containing no CUDA constructs—is encountered, CU2CL

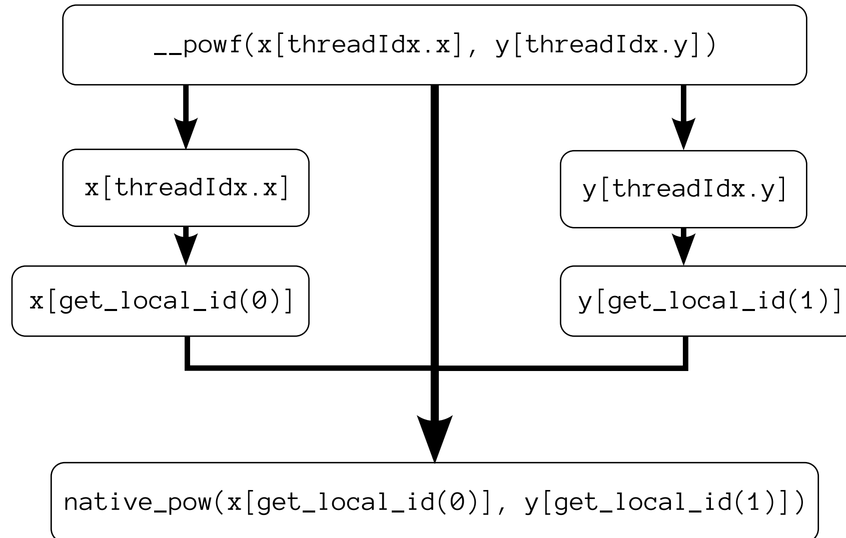

Figure 4.4: Example of rewriting an expression and its subexpressions.

recurses down into its child subexpressions in a depth-first manner, invoking the recursive expression rewriting mechanism. If a subexpression is rewritten, the function will return a new string which is used to replace the text in the child's original source range. After all children have been processed in this manner, the associated Rewriter is used to retrieve the range of text for the current expression, including all rewrites that took place when in the subexpressions.

4.3.3 Rewriting Includes

In order to provide a seamless translation experience, some `#include` preprocessor directives in the original CUDA source must be removed or rewritten. As `#include`s are not resident in the AST we transform, this rewriting has been implemented using the Clang driver's preprocessor, as shown in Figure 4.6. CU2CL registers a callback with the preprocessor that

```
 1: procedure RewriteExpression(expr)
 2:     type ← Type(expr)
 3:     if type is interesting then
 4:         return RewriteType(expr)
 5:     else
 6:         r ← SourceRange(expr)
 7:         for all subexpr in SubExpressions(expr) do
 8:             s ← RewriteExpression(subexpr)
 9:             if rewrite occurred then
10:                 subr ← SourceRange(subexpr)
11:                 ReplaceSource(subr, s)
12:             end if
13:         end for
14:         return GetSourceWithRewrites(r)
15:     end if
16: end procedure
```

Figure 4.5: Algorithm detailing how expressions are recursively rewritten.

is invoked upon a new `#include` being processed. As the preprocessor expands the include directive, it has all the information necessary to decide whether CU2CL should rewrite the directive. In particular, CU2CL needs the current file that is being parsed, the name of the file that is to be included, and whether or not it is a system header. Finally, if the directive is to be rewritten the source range associated with the `#include` is passed to Clang's rewriting mechanism along with any new text. By tying into Clang's preprocessor, CU2CL can avoid the task of locating these directives manually. This adds robustness and efficiency to CU2CL's `#include` rewriting.

The `#include` rewrites fall into two categories: removing `#include`s pointing to CUDA and system header files and rewriting includes to CUDA files that CU2CL has rewritten.

In the first case, CU2CL removes includes to *cuda.h* and *cuda_runtime_api.h* found in any rewritten files, both host and kernel files. It also removes system header files (e.g. *stdio.h*)

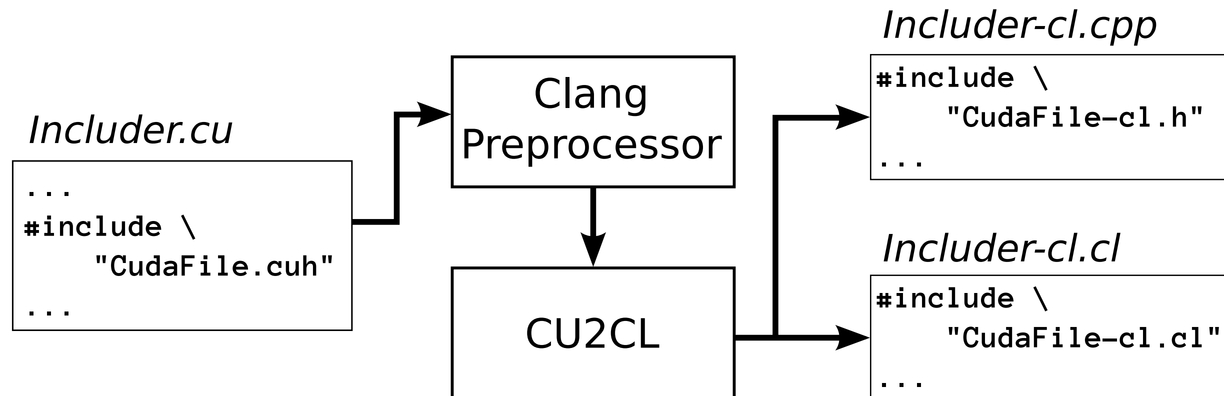

Figure 4.6: Example of rewriting an `#include` directive.

from the OpenCL kernel files, as they cannot be used in device code. In Clang, these header files are identified as those included using the angle bracket notation as opposed to double quotes.

In the second case, CU2CL rewrites includes to files that have been rewritten. The original included CUDA source files will be split into two new files, one for the host and one for device code (e.g. *cudaFile.cu* will become *cudaFile-cl.h* and *cudaFile-cl.cl*). Therefore, CU2CL rewrites the original `#include`s so that they point to the new OpenCL files. Figure 4.6 shows an example of how an `#include` pointing to a CUDA file may be rewritten in a new host code file. The kernel file will be used during run time compilation of device code, so it is not `#include`d.

4.4 Challenges

There are some cases in automatically translating CUDA to OpenCL that make generating maintainable code difficult. For instance, CUDA is based on C and can therefore make use of a preprocessor to generate code at compile time. Consequently, while an abstract syntax tree (AST) representation of the source may be fine for compilation, the resulting translated code may look very different from the original. In Clang, macros are represented as a series of tokens, thus, while its libraries provide access to the tokens, they are simply raw and unparsed. Therefore, the process of rewriting macros would require at least partial parsing of the tokens contained within. This is a complex task, beyond the current scope of CU2CL.

Another wrinkle in automatic translation occurs when CUDA applications make use of closed-source libraries built on top of CUDA, such the as CUBLAS or CUFFT libraries in the CUDA toolkit. A pure CUDA translator like CU2CL cannot fully translate these applications, since the libraries will continue to contain CUDA constructs. As a result, users will have to either reimplement the libraries from scratch in OpenCL or find other libraries written in OpenCL that provide the same functionalities. On the other hand, if a CUDA library's source code is available, it could be translated using CU2CL, resolving the problem. The same issue is seen with user functions expecting CUDA constructs or results from CUDA calls as arguments, which cannot be handled entirely without rewriting the functions first. This is typical of utility functions like those found in the CUDA SDK's `cutil` library, as well as common CUDA code shared between several applications like in the SHOC benchmark suite [8].

The largest unresolved issue in CU2CL's translation of CUDA lies in translating applications with several files that are separately compiled. CU2CL currently expects the main source file to be a full CUDA application, including all kernels functions and the main method. While most complications of separate translation could be resolved, generating code to initialize and build kernel functions without knowledge of where they come from is very difficult. This could be mitigated by passing CU2CL a path to the kernel files, but this has not yet been implemented.

Chapter 5

Evaluation

We evaluate CU2CL along three metrics: the translator's performance, the performance of translated applications, and the amount of CUDA covered.

5.1 Performance of the CU2CL Translator

While CU2CL will ideally be run once on a given CUDA application, we do not wish to simply ignore the time to translate the source to OpenCL. This is particularly important if the end user wishes to convert multiple CUDA applications from the well-established CUDA ecosystem. Thus, we have evaluated CU2CL's speed of translation on several applications from the CUDA SDK and Rodinia benchmark suite. For each application we averaged the time to translate the code from CUDA to OpenCL over ten runs. Two times are reported:

Source	Application	Total Translation Time (s)	CU2CL Time (ms)	CUDA Lines	Preprocessed Lines
CUDA SDK	asyncAPI	0.331	3.35	136	13743
	bandwidthTest	0.623	7.98	891	34766
	BlackScholes	0.606	5.24	347	34222
	matrixMul	0.607	5.47	351	34209
	scalarProd	0.327	3.75	171	13835
	vectorAdd	0.287	3.11	147	11605
Rodinia	Back Propagation	0.300	4.46	313	12765
	Breadth-First Search	0.301	4.51	306	12594
	Hotspot	0.297	4.90	328	11810
	Needleman-Wunsch	0.303	5.46	418	12815
	SRAD	0.303	6.56	541	12778

Table 5.1: CU2CL's translation times for applications from the CUDA SDK and Rodinia and their original lines of code.

the total time, including the time for the Clang driver to parse and perform semantic analysis of the program, and the time CU2CL takes to translate a CUDA application. Table 5.1 summarizes our results. The table shows that, for these applications, all translation times are under a second. Furthermore, note that CU2CL's translation time is two orders of magnitude smaller than the total translation time. Thus, Clang's parsing and AST generation dominates the total time and CU2CL's overhead is nearly negligible. This implies that larger applications should be translated in at most the time Clang takes to compile them.

The test applications vary in length from a couple hundred to nearly a thousand lines of code. We see in Figure 5.1, however, that the total translation time is dependent on the total number of lines of code, after running the preprocessor. Figure 5.2 shows that CU2CL's

translation time is instead dependent on the number of CUDA lines found in the original source code. This is expected, as it does not traverse parts of the AST that come from non-CUDA files.

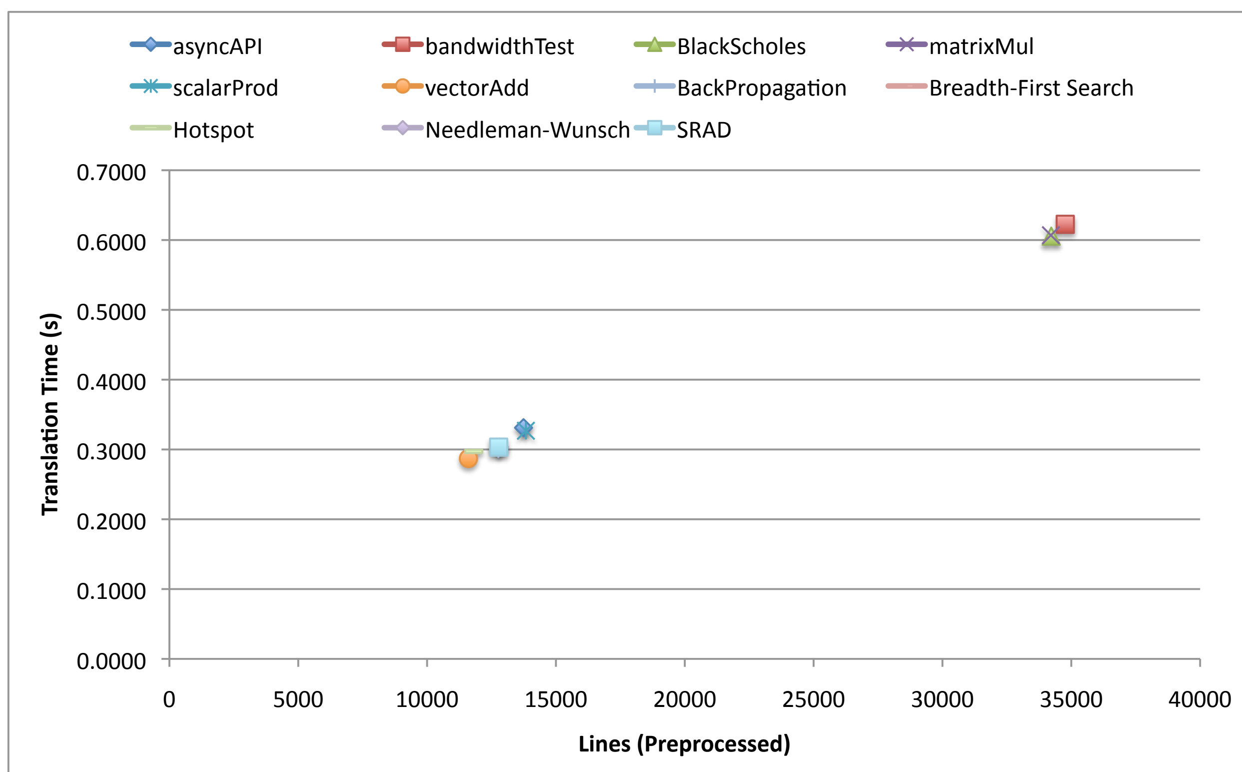

Figure 5.1: Total time for CU2CL to translate an application, including the time for Clang to parse the source and generate the AST, versus the number of preprocessed lines in the original CUDA source.

5.2 Performance of CU2CL-Translated Applications

We evaluated the performance of one application from the CUDA SDK, vectorAdd, and three from the Rodinia benchmark suite, Hotspot, Needleman-Wunsch, SRAD. Each application was only modified to add code to measure the run time of the program.

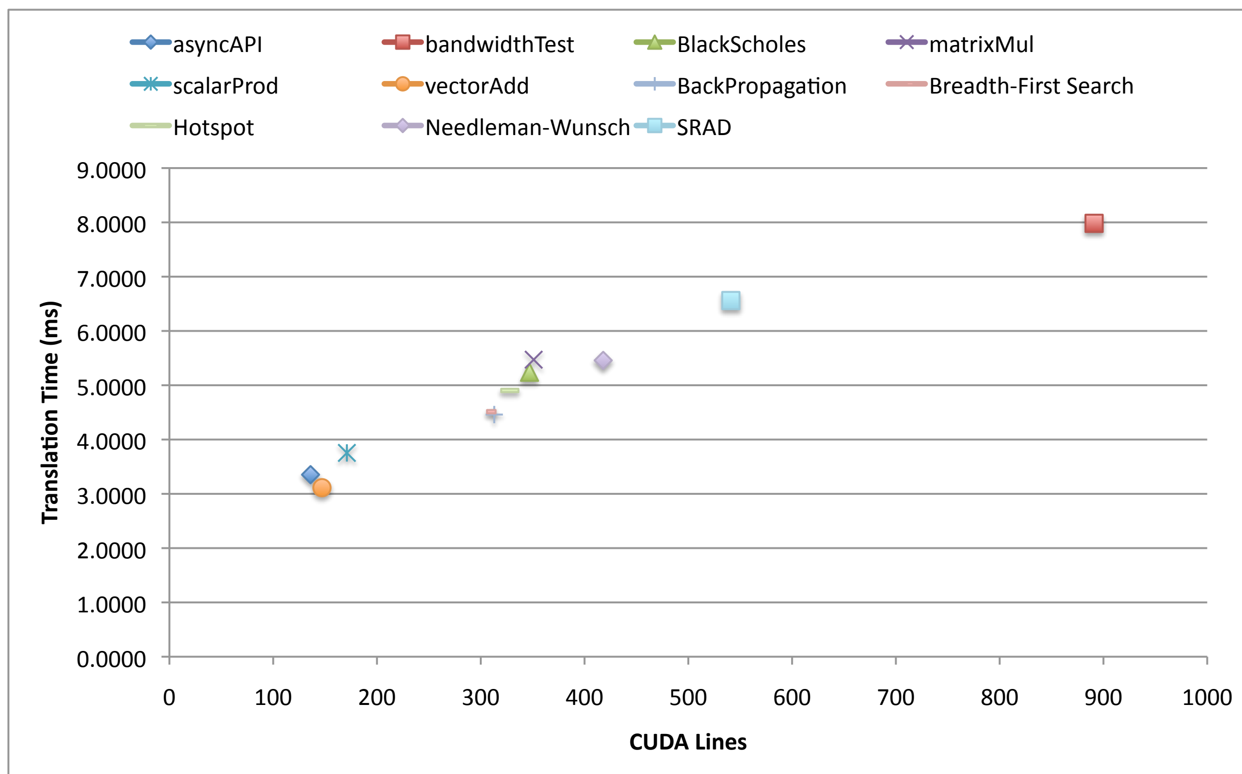

Figure 5.2: Time spent in CU2CL translating an application versus the number of lines in the original CUDA source.

vectorAdd is a very simple application that generates two random vectors in host memory and copies them over to the GPU's global memory. The kernel performs the addition and stores them in a third vector allocated in global memory. The resulting vector is then copied back to host memory.

Hotspot is a physics simulation that can estimate the temperature of a processor given its architectural floor plan and some power measurements. As the simulation runs, a series of differential equations is solved, outputting the average temperature in each part of the processor.

Needleman-Wunsch is a global sequence alignment application commonly used in the field of bioinformatics for the analysis of DNA sequences. Two character sequences are compared and a two dimensional matrix is filled with scores—calculated using a predetermined scoring chart—showing how good the match between the two is. The last step is to trace-back through the matrix and find the aligned sequence, including any insertions or deletions. Typical implementations would launch a kernel per anti-diagonal in the matrix, but this implementation breaks the matrix into blocks of which multiple can be computed at once. This reduces the number of kernel launches, resulting in better performance.

SRAD (Speckle Reducing Anisotropic Diffusion), is a computational method of removing noise from images produced by ultrasonic or radar imagery applications. The goal is to do this without losing any of the important features present in the pictures. In the SRAD implementation, two kernels are launched per iteration of the main loop and memory copies to and from the GPU are done.

For all of the experiments, we compiled and ran the applications on a commodity desktop computer with two 2.0-GHz Intel Xeon E5405 quad-core CPUs and 4 GB of RAM. The GPU device used is an NVIDIA GTX 280, which has 30 streaming multiprocessors (240 total cores) clocked at 1.3 GHz along with 1 GB of graphics memory. The compute capability, as defined by NVIDIA, is 1.2.

In running each application, the run time is taken to be the time starting from the first data copy to GPU device memory from the host and ending after the last copy back to host memory. For all of these applications, this includes all of the time spent launching and

Application	CUDA	Automatic OpenCL		Manual OpenCL	
		Time	% Change	Time	% Change
vectorAdd	0.0499s	0.0516s	+3.33%	0.0521s	+4.32%
Hotspot	0.0177s	0.0565s	+219.06%	0.0561s	+217.14%
Needleman-Wunsch	6.65s	8.77s	+31.87%	8.77s	+31.86%
SRAD	1.25s	1.55s	+24.30%	1.54s	+23.47%

Table 5.2: Run times of the four CUDA applications and both OpenCL ports (including percent differences with respect to the CUDA times) on an NVIDIA GTX 280.

executing kernels. The reason behind not timing the whole application was to get a more accurate comparison of the CUDA and OpenCL codes; we did not wish to add the time OpenCL spends compiling the kernels. Each program was executed a total of ten times and their run times were averaged over those to get the reported numbers.

Table 5.2 summarizes the performance comparisons between the original CUDA code, CU2CL's automatically-generated OpenCL, and our manually-ported OpenCL. As can be seen, the OpenCL vectorAdd performs nearly as well as CUDA for both cases. This is not the case for the real-world Rodinia applications. SRAD's OpenCL performance is roughly 25% worse than its CUDA. The OpenCL Needleman-Wunsch code performs about 30% worse than the CUDA version. The worst is Hotspot, however, which performs several times worse than the original CUDA. These results are typical as the NVIDIA OpenCL implementation is known to not perform as many optimizations as CUDA does [11, 15].

In all applications, the automatically-translated OpenCL performs just as well as the manually-ported OpenCL code. This is to be expected as the differences between the two versions for each application are minor, and would not be expected to have much performance impact at all.

5.3 Coverage of the CU2CL Translator

We have stated that CU2CL can handle the majority of CUDA constructs found in most applications. In order to justify this claim, we studied the top CUDA calls in the CUDA SDK and Rodinia benchmark suite, summarized in Tables 5.3 and 5.4. In each, the fifteen most used calls are given, alongside the number of times they were found, a running count of the total percentage those calls cover, and how many files they were used in. As can be seen from the tables, the top three CUDA runtime API calls, by the number of times called, are `cudaFree`, `cudaMalloc`, and `cudaFree`, the most basic device memory operations. This is expected, as doing any useful work on the GPU will require device memory allocation and copying to and from host memory—again shown in the number of files they are found in. CU2CL already supports these CUDA calls among several others, as detailed in Chapter 3. In supporting just the top five calls in the CUDA SDK—though it can translate many more than that—CU2CL covers 51.5% of the total CUDA runtime API calls found. In supporting five of the top calls in Rodinia, it covers 73.0% of the total calls.

While these theoretical results initially led the decision of which CUDA API calls to support first, they do not provide much insight into how many real applications are covered by CU2CL. For more concrete results, we noted the number of lines that had to be manually ported when translating applications from the CUDA SDK and Rodinia. This study shows how successful our translator already is.

CU2CL supports a large majority of the CUDA runtime API. In particular, it can automatically translate API calls from the major CUDA modules: Thread Management, Device

CUDA Call	**Times Called**	**Running % of Total**	**Files Used In**
`cudaFree`	222	12.69	81
`cudaMalloc`	220	25.27	80
`cudaMemcpy`	207	37.11	68
`cudaThreadExit`	126	44.31	70
`cudaThreadSynchronize`	126	51.52	57
`cudaGetErrorString`	58	54.83	14
`cudaGetDeviceProperties`	57	58.09	54
`cudaSetDevice`	53	61.12	52
`cudaEventRecord`	50	63.98	11
`cudaEventDestroy`	29	65.64	11
`cudaBindTextureToArray`	26	67.12	17
`cudaEventCreate`	25	68.55	10
`cudaGraphicsUnmapResources`	25	69.98	18
`cudaGraphicsUnregisterResource`	25	71.41	16
`cudaGraphicsMapResources`	25	72.84	18

Table 5.3: Top 15 CUDA calls in the CUDA 3.2 SDK.

CUDA Call	**Times Called**	**Running % of Total**	**Files Used In**
`cudaFree`	104	22.37	17
`cudaMemcpy`	99	43.66	17
`cudaMalloc`	78	60.43	17
`cudaMemcpyToSymbol`	33	67.53	9
`cudaThreadSynchronize`	21	72.04	12
`cudaMemset`	19	76.13	8
`cudaCreateChannelDesc`	17	79.78	3
`cudaUnbindTexture`	13	82.58	3
`cudaGetErrorString`	10	84.73	5
`cudaSetDevice`	9	86.67	8
`cudaBindTexture`	8	88.39	4
`cudaGetLastError`	8	90.11	5
`cudaFreeArray`	6	91.40	2
`cudaGetDeviceProperties`	6	92.69	6
`cudaMemcpy2D`	6	93.98	2

Table 5.4: Top 15 CUDA calls in the Rodinia benchmark suite.

Source	Application	Lines	Changed	%
CUDA SDK	asyncAPI	136	4	97.06
	bandwidthTest	891	9	98.99
	BlackScholes	347	4	98.85
	matrixMul	351	2	99.43
	scalarProd	171	4	97.66
	vectorAdd	147	0	100.00
Rodinia	Back Propagation	313	5	98.40
	Breadth-First Search	306	8	97.39
	Hotspot	328	7	97.87
	Needleman-Wunsch	418	0	100.00
	SRAD	541	0	100.00

Table 5.5: CU2CL's automatic translation coverage of a range of applications.

Management, Stream Management, and Event Management. The translator also supports the most commonly used methods of the Memory Management module, including calls to allocate device and pinned host memory. As a result of CU2CL's robust translation methods, alongside its support for many CUDA constructs, it can automatically translate many applications almost in their entirety. Table 5.5 shows this for applications from the CUDA SDK and the Rodinia benchmark suite. In each case only a few lines of host or kernel code had to be manually ported.

Of manual changes, none are particularly difficult to handle and support will be added as CU2CL evolves, save for one exception. As pointed out in work for the other CUDA to OpenCL translator [19], device memory declarations rewritten to `cl_mem`s may require a propagation of type rewrites in other declarations, such as methods that expect a device pointer. CU2CL does not currently cover this. This is not a limitation of the Clang framework, but is instead caused by the difficulty of the problem, requiring control flow analysis of the program.

Chapter 6

Conclusion

6.1 Summary

In this thesis we have presented CU2CL, an automated source-to-source translator from CUDA to OpenCL. By leveraging the production Clang compiler framework, we were able to take advantage of its powerful source-level tools to perform the translation. We used Clang's driver to efficiently parse the CUDA source files and also employed its versatile AST and Rewrite libraries to traverse CUDA applications and go through with complicated transformations in an elegant manner.

In designing and implementing CU2CL, we determined common patterns that could be of use for future work in Clang-based source-to-source translators. We also demonstrated methods of recursively rewriting expressions and of efficiently rewriting include directives through the use of the Clang driver's preprocessor.

We gave a detailed procedure of CU2CL's automatic translation, showing how specific CUDA constructs mapped to OpenCL and highlighting portions that have no direct equivalent in OpenCL. Experiments on sample applications from the official CUDA SDK and the Rodinia benchmark suite showed that the OpenCL code generated by CU2CL can perform as well as codes manually-translated. Unfortunately both versions of the ported program were slower than the native CUDA; however, previous work [7, 15] shows that by applying some optimizations to the OpenCL kernels and host code, the performance gap can be narrowed.

As an early prototype, we focused on flexibility and correctness, therefore we chose to support a useful subset of the language. We have shown that the subset chosen covers most of the CUDA runtime API calls used in many applications. To further display CU2CL's CUDA coverage, we showed how several applications could be translated almost in their entirety without any manual translation. In effect, our translator can already perform the majority of the OpenCL porting effort.

6.2 Future Work

The work presented in this thesis may be extended in a several ways. A few possible directions are listed here:

Generalize rewriting patterns to rewriting framework: The common patterns introduced in Chapter 4 and used to modularize CU2CL's rewriting can be applied outside of translating CUDA to OpenCL. Rewriting types in a general manner along with

making intelligent use of uniformity in an API that is to be rewritten are both useful in many other source transformation scenarios. Therefore, generalizing these kinds of patterns into a framework for rewriting program source codes would be very useful work. This could be done via polymorphism, similar to how Clang currently provides visitors for statement nodes in the AST.

Create an optimizing compiler for device-specific OpenCL optimizations: Although OpenCL kernels may run on any OpenCL-capable device does not mean they will perform well without device-specific optimizations, as already shown in [7,15]. Preliminary results from running CU2CL's automatically-translated OpenCL of the applications in Chapter 5 on an AMD Radeon HD 5870 gave poor results. Though the GPU has a higher theoretical peak performance than the NVIDIA GTX 280, its run times were 0.075s, 2.11s, and 15.24s, for vectorAdd, SRAD, and Needleman-Wunsch, respectively. The values are all at least 50% worse than the OpenCL run times on the NVIDIA GPU presented in Table 5.2. Buildling an optimizing compiler like [28] on top of CU2CL would allow for CUDA applications to be both translated and optimized for specific devices. As in the optimizing compiler, CU2CL would need to identify regions in kernels and host code where optimizations could be applied. Further down the road, a framework for user-supplied optimization passes could be implemented.

Extend CU2CL's CUDA support: Currently CU2CL's translation is incomplete; we have shown that it "only" covers 50% of the CUDA runtime API, with no support for the driver API. Therefore, we are interested in seeing other modules of the CUDA run-

time API supported, especially the Texture and Graphics modules (including OpenGL support). Along the same lines support for the CUDA driver API in CU2CL would come in handy for many applications. Given that the OpenCL API and CUDA driver API are very similar this should be easier than translating CUDA runtime API has been thus far. Along the same lines, CU2CL's translation process has areas that could still be improved. Of utmost importance would be a method for translating CUDA applications with source files that are separately compiled, as this remains quite a limiting factor. We would also like to add the ability to propagate type changes throughout an application, as is already done in CUDAtoOpenCL [19]. While several other possible features are being worked on, the two mentioned here affect more CUDA applications than the rest.

Translate OpenCL to CUDA: It has been shown that NVIDIA's OpenCL compiler produces slower code than its CUDA compiler [11, 15]. Therefore, one possible "optimization" for OpenCL applications when running on NVIDIA GPUs is to convert them to CUDA. This would allow developers to continue to write their applications in OpenCL and yet take advantage of CUDA's higher performance. Translating from OpenCL to the CUDA runtime API would be difficult, as OpenCL gives you access to features not available in the runtime API. However, given the similarities between OpenCL and the CUDA driver API, translating between the lower-level APIs should be much simpler. Furthermore, such future work could make use of the translation techniques presented in this thesis for source-to-source translation with Clang, which should aid the process.

Bibliography

[1] "clang": a C language family frontend for LLVM. http://clang.llvm.org/. [Online; accessed 15-April-2011].

[2] OpenCL™ Optimization Case Study: Diagonal Sparse Matrix Vector Multiplication. http://developer.amd.com/documentation/articles/pages/opencl-optimization-case-study.aspx. [Online; accessed 10-July-2011].

[3] Parboil Benchmark suite. http://impact.crhc.illinois.edu/parboil.php. [Online; accessed 15-April-2011].

[4] PTX ISA 2.2. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/ptx_isa_2.2.pdf. [Online; accessed 10-July-2011].

[5] Rodinia: A Benchmark Suite for Heterogeneous Computing. http://lava.cs.virginia.edu/Rodinia/rodinia.htm. [Online; accessed 10-July-2011].

[6] I.D. Baxter, Christopher Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 625–634. IEEE Computer Society, 2004.

[7] Mayank Daga, Thomas R W Scogland, and Wu-chun Feng. Architecture-Aware Optimization on a 1600-core Graphics Processor. Technical report, Virginia Tech, 2011.

[8] Anthony Danalis, Gabriel Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, and J.S. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[9] Gregory Diamos. The design and implementation ocelot's dynamic binary translator from ptx to multi-core x86. *Center for Experimental Research in Computer Systems*, 2009.

[10] R. Dominguez, Dana Schaa, and David Kaeli. Caracal: Dynamic translation of runtime environments for gpus. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, number March, page 7. ACM, 2011.

[11] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multiplatform GPU Programming, 2010.

[12] Khronos Group. OpenCL. `http://www.khronos.org/opencl/`. [Online; accessed 15-April-2011].

[13] M.J. Harvey and G. De Fabritiis. Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*, 182(4):1093–1099, April 2011.

[14] Martin Jurecko, Jana Kocisova, Jan Busa Jr., Tomas Kasanicky, Marek Domiter, and Marian Zvada. Evaluation Framework for GPU Performance Based on OpenCL Standard. *2010 First International Conference on Networking and Computing*, pages 256–261, November 2010.

[15] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, H. Takizawa, and H. Kobayashi. Evaluating performance and portability of OpenCL programs. In *The Fifth International Workshop on Automatic Performance Tuning (iWAPT2010)*, 2010.

[16] Chris Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Optimization*, 2004.

[17] Seyong Lee, S.J. Min, and Rudolf 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*, pages 101–110. ACM, 2009.

[18] S.I. Lee, T.A. Johnson, and Rudolf Eigenmann. Cetus–an extensible compiler infrastructure for source-to-source transformation. *Languages and Compilers for Parallel Computing*, (9703180):539–553, 2004.

[19] Deepthi Nandakumar. Automatic Translation of CUDA to OpenCL and Comparison of Performance Optimizations on GPUs. 2011.

[20] NVIDIA. CUDA Toolkit & SDK. `http://developer.nvidia.com/cuda-toolkit-sdk`. [Online; accessed 15-April-2011].

[21] NVIDIA. CUDA Zone. `http://www.nvidia.com/object/cuda_home_new.html`. [Online; accessed 15-April-2011].

[22] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Issues*, 2(3):215–226, 2000.

[23] J. Stratton, S. Stone, and W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. *Languages and Compilers for Parallel Computing*, pages 16–30, 2008.

[24] R Strzodka, M Doggett, and a Kolb. Scientific computation for simulations on programmable graphics hardware. *Simulation Modelling Practice and Theory*, 13(8):667–680, November 2005.

[25] M.L. Van De Vanter. Preserving the documentary structure of source code in language-based transformation tools. *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 131–141, 2001.

[26] J. Van Wijngaarden and Eelco Visser. Program transformation mechanics: a classification of mechanisms for program transformation with a survey of existing transformation systems. *Technical report UU-CS*, (2003-048), 2003.

[27] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems. *Domain-Specific Program Generation*, (February), 2003.

[28] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*, page 86, 2010.