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

Gabriel Martinez
Dept. of Computer Science
Virginia Tech, USA
Email: mystal@vt.edu

Wu-chun Feng
Dept. of Computer Science
Virginia Tech, USA
Email: wfeng@vt.edu

Mark Gardner
Office of IT
Virginia Tech, USA
Email: mkg@vt.edu

***Abstract*—The use of graphics processing units (GPUs) in high-performance parallel computing continues to 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 runtime 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 typically 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 CUDA runtime API, and we have successfully translated many applications from the CUDA SDK and Rodinia benchmark suite. The performance of our automatically translated applications via CU2CL is on par with their manually ported countparts.

I. Introduction

The introduction of the NVIDIA CUDA ecosystem [1] in 2007 spurred a flurry of activity in the use of the graphics processing unit (GPU) as a programmable device for general-purpose computing. As a result, the past four years has seen tremendous growth in GPU-accelerated applications using CUDA. However, CUDA is only available on NVIDIA GPUs.

In an effort to democratize the use of GPUs for general-purpose computing and not be behooven to a single vendor, Apple developed OpenCL and submitted it to the Khronos Group to develop as an open-source standard [2]. OpenCL is a vendor-neutral framework for writing programs that run on heterogeneous computing platforms consisting of CPUs, GPUs, or other processors, not just GPUs.

Though still relatively new, OpenCL has several 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 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 parallel computing platform.

However, adoption of OpenCL has been slow for a number of reasons. One hindrance is the OpenCL API, which is lower level than the commonly used CUDA API, thus requiring more time and effort to set-up devices and execute kernels. Of greater impact has been CUDA's established presence in the arena of general-purpose computation on the GPU (GPGPU), which has made it the de facto GPGPU programming environment. Not surprisingly, there are significantly more 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 relatively straightforward given OpenCL's GPU origins. Nevertheless, performing this process by hand can be tedious and error-prone. Although there is some initial work to alleviate this issue [3], nothing has been done to automate the process.

We propose CU2CL, an automated CUDA-to-OpenCL source-to-source translator built using the Clang compiler framework. As shown in Figure 1, CU2CL takes an application's CUDA source files and rewrites them into equivalent OpenCL host *and* kernel files. In this process, it adds all the OpenCL "boilerplate" code necessary to set-up the compute environment and translates the most-used CUDA features, while delivering a framework to handle future versions (or larger subsets) of CUDA in the future.

In addition to providing a robust translator framework that will generate maintainable OpenCL code with little to no manual porting. the framework will enable architecture-aware optimization passes for different compute devices. (As previously shown [4]–[6], OpenCL kernels must be optimized for different architectures.) CU2CL is simply the first, albeit critical, step in this process.

Our contributions in this paper include the following:

- The first framework for the automatic translation of GPU applications from CUDA to OpenCL, i.e., CU2CL.
- General insights for designing source-level tools within our framework, including (1) *common patterns* that arise when performing the translations, (2) a technique for *recursively rewriting expressions*, and (3) a process for *rewriting #includes*.
- An evaluation of our CU2CL prototype with respect to source-to-source translator performance, performance of the translated applications, and translator coverage.

The remainder of this paper is organized as follows. Sec-

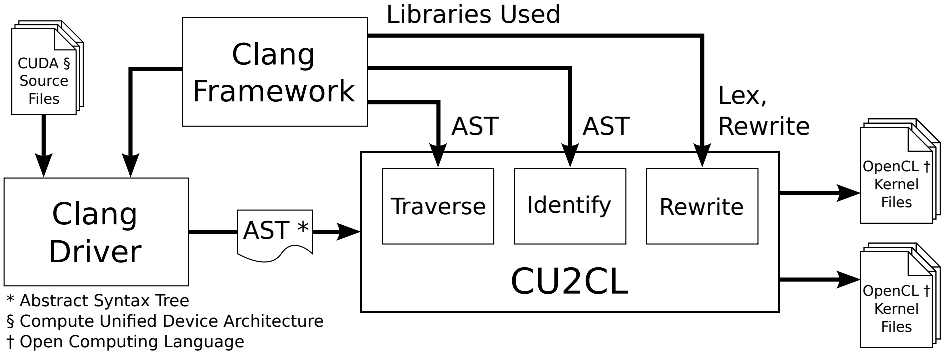


Fig. 1. High-level overview of the CU2CL translation process.

tion II presents related work in the areas of GPGPU computing and source-to-source translation. Section III gives an overview of the CUDA and OpenCL frameworks, focusing on how their similarities and differences influence the translation effort. Section IV discusses our approach in designing and implementing CU2CL, in particular, what influenced our decision to use the Clang framework and how we overcame challenges associated with the choice. In Section V, we evaluate CU2CL's translation performance, the performance of the automatically translated applications, and an analysis of the CUDA coverage. Lastly, we present future work in Section VI and summarize our work and contributions in Section VII.

II. Related Work

There exist several projects that translate to (or from) the CUDA programming model. Of particular note is an OpenMP-to-CUDA source-to-source translator [7]. Working towards a similar goal, but in the reverse direction, MCUDA [8] is a source-to-source translator that instead translates CUDA to multi-threaded CPU code. Both translators are built using Cetus [9], a source-to-source translator framework for C and other C-based languages.

Closer to our goal, Swan [3] is tool that is meant to ease the transition between OpenCL and CUDA. However, Swan is not actually a source-to-source translator like CU2CL; instead Swan provides a higher-level library that abstracts both CUDA and OpenCL, such that an application makes calls to Swan and allows Swan to take care of the details in mapping them to CUDA or OpenCL. The Swan API is limited, as it currently only abstracts a few features in common between CUDA and OpenCL. On the other hand, it provides a simple Perl script that can automatically translate some kernel code, essentially performing a regular expression-based search and replace operation on the source. Much of the work, however, is left up to the developer to port his or her application (especially host code) and its kernels to use Swan's libraries.

Ocelot [10] is primarily a PTX-to-LLVM translator and runtime system that can decide whether to run the PTX on a GPU device or on a CPU after just-in-time (JIT) compiling it to LLVM [11]. In this regard, Ocelot 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. In today's compilers, this is a typical approach to take, converting a higher-level language to some intermediate representation and then compiling it to the target architecture.

For GPUs and other OpenCL-capable devices, multiple intermediate languages can be targeted, such as NVIDIA PTX, AMD IL, and LLVM. Ocelot would need to implement a PTX-to-AMD IL backend in order to execute CUDA applications on an AMD GPU, as done in [12]. More generally, a new backend must be made for each intermediate language that is to be supported. With CU2CL, the OpenCL implementation for the desired device will simply handle compiling the code to the proper bytecode. As a result, CUDA applications, once ported, are automatically available on several architectures.

Independent from our work, the University of Illinois is also developing a CUDA-to-OpenCL translator [13], but using Cetus and the CUDA parser from MCUDA. This work has not been published yet nor has the code been released, but it is stated that many CUDA features are not yet working.

III. GPGPU Frameworks

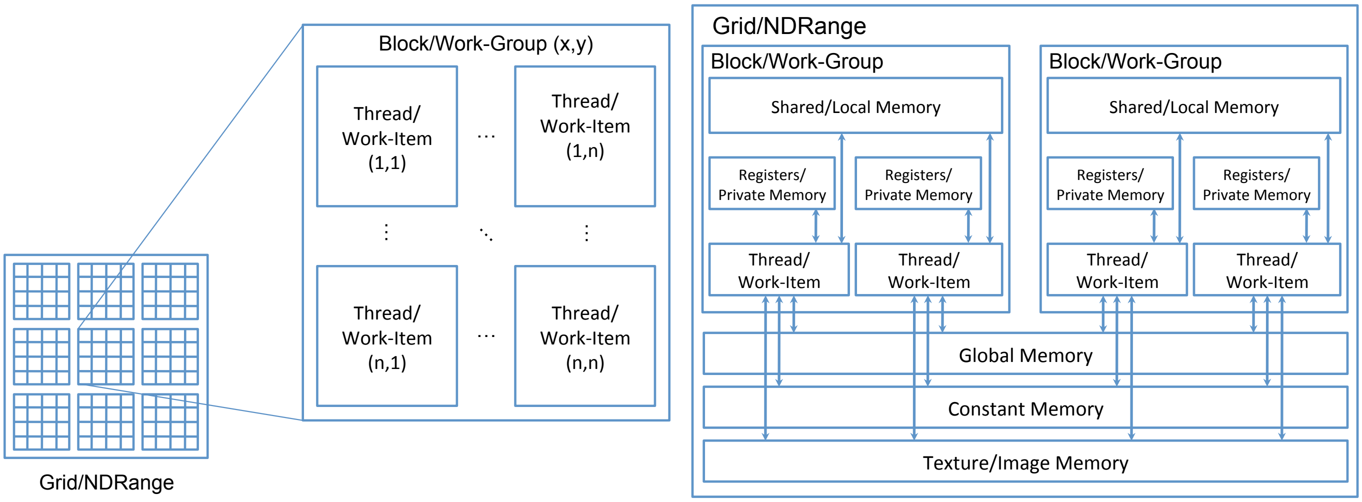
CUDA and OpenCL are frameworks designed for general-purpose GPU computation. Both have kernels that execute on compute devices, threads that run in parallel within them, and methods for managing device memory and launching compute kernels. However, since CUDA is meant for GPUs, CUDA provides many GPU-centric features in that are not found in OpenCL. OpenCL, on the other hand, provides a platform-agnostic framework.

Below we examine version 3.2 of the CUDA API—the latest stable release—along with the OpenCL 1.0 standard.[1]

A. CUDA

CUDA is a programming framework and environment that enables data-parallel, SIMD computations to be offloaded onto a GPU. Users write device code in a C-like language, which then runs on the streaming multiprocessors of NVIDIA GPUs. Kernels functions (i.e., SIMD procedures launched from the host) are executed across a possibly multi-dimensional *grid* of *blocks*. Each *block* contains numerous *threads* in another possibly multi-dimensional configuration. These configurations are

[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 AMD and NVIDIA GPUs.



(a) Programming models

(b) Memory models

Fig. 2. Overviews of the CUDA and OpenCL models. Items with multiple labels first give the CUDA and then the OpenCL term.

specified by the host during a kernel invocation. Figure 2a gives an overview of the CUDA programming model.

CUDA's memory model has three separate memory spaces: *global* memory, off-chip and accessible by all threads in all blocks—*shared* memory—on-chip and available to all threads in a block—and *registers*—owned by one thread. In addition to these, two special-use memory spaces provide faster memory operations: *constant* memory and *texture* memory. *Constant* memory is cached for fast reads, but is limited in size and does not support writes. *Texture* memory allows for fast reads as well as writes, but is also rather limited in size. Furthermore, kernels must use special built-in functions to access data residing in the region. In general, device memory must be explicitly allocated through CUDA API calls and is usually initialized by copying data from host memory. Figure 2b shows how the memory spaces are laid out in CUDA.

CUDA consists of two different APIs: a low-level driver API and a high-level runtime API. The driver API is CUDA's lower-level API for programming GPUs. It requires that a GPU context be created for each GPU in use, as nothing is implicitly done behind the scenes for the developer. Setting up and executing kernels requires several API calls to set arguments and the execution configuration. On the other hand, the driver API allows for much finer control over GPU devices.

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 system initializes a GPU device on the first call to any CUDA runtime API method.

CUDA also provides a small extension to C, coined CUDA C, that allows kernel code and host code to be intermixed. Device code and memory are distinguished by CUDA-specific function or variable qualifiers added to their declarations. CUDA C also extends C by introducing a special notation for executing kernels, which allows the number of `blocks` and `threads` per `block` to be specified. To take advantage of these extensions, applications in CUDA C must be compiled with NVIDIA's compiler, `nvcc`, which handles the C extensions and properly sorts host and kernel declarations.

Most applications opt to use CUDA C along with the runtime API [14], [15]. Thus, we initially focus our CU2CL source-to-source translator on this combination.

B. OpenCL

The OpenCL standard is an open, vendor-neutral programming model and environment for executing general-purpose computations. More general than CUDA, it allows for the use of arbitrary compute devices. Vendor-provided implementations map the abstract compute and memory models to real hardware. OpenCL device code, written in a C99 variant, is typically compiled at runtime by a vendor-provided compiler through OpenCL API calls. Kernels consist of *work-groups* (similar to CUDA *blocks*) each of which consist of *work-items* (akin to *threads* in CUDA), as shown in Figure 2a.

Figure 2b shows that CUDA and OpenCL also have similar memory models: *global* memory, equivalent to CUDA's *global* memory; *local* memory, which resembles CUDA's *shared* memory; and *private* memory, analogous to *registers* in CUDA. Support for *constant* and *image memory*—like CUDA's texture memory—also exists.

The OpenCL standard defines one API which is very similar to CUDA's driver API. Users must be aware of the low-level concerns and write some of the code that the CUDA runtime API handles automatically. Additionally, OpenCL adds the concept of platforms—an abstraction of the set of installed implementations—and device command queues, similar to CUDA streams, for sending commands to a particular device. On the other hand, the OpenCL kernels are very similar to those in CUDA, containing constructs that map almost one-to-one to the CUDA equivalents. Noteworthy exceptions include how image memory is accessed and the lack of some synchronization functions.

C. Mapping CUDA to OpenCL

Many assume that translating CUDA to OpenCL is effectively a one-to-one mapping process. While most CUDA constructs map one to one to OpenCL, not all do, as shown in Tables I, II, and III. As a result, translating certain parts of CUDA require a deeper understanding of both APIs to find suitable corresponding constructs. Furthermore, these tables provide only a high-level view of the translation process; in practice, sophisticated techniques to perform the transformations are required. For example, in some cases, data must be *tracked throughout the lifetime of the translation* before certain translations can be finalized. Such a case is found when rewriting device pointers to `cl_mem`s, as the rewrite must propagate through types found in parameters and `sizeof` expressions.

CUDA	**OpenCL**
Device pointers	`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`

TABLE I
COMMON CUDA DATA STRUCTURES AND THEIR OPENCL EQUIVALENTS.

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 II
CUDA API MODULES 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 III
COMMON CUDA KERNEL BUILTIN FUNCTIONS AND VARIABLES AND THEIR OPENCL EQUIVALENTS.

IV. DESIGN AND IMPLEMENTATION OF CU2CL

Having covered how the two GPGPU frameworks differ, this section presents the design of our CU2CL source-to-source translator as a Clang plugin.

A. Approach

Several mechanisms for source-to-source translation are in common use—from simple tools that use 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.

Our project seeks to produce a tool that can be rapidly adopted by the CUDA and OpenCL communities. While numerous frameworks and tools for source-to-source translation exist [16]–[18], 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 [19], primarily for the three following reasons. First, though relatively young, Clang has a large and active community with many new features and better quality every day. 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 lexing, parsing, semantic analysis, and much more, may be used independently of the driver to create other source-level tools. Finally, Clang recently added support for parsing CUDA C extensions.

As implicitly noted in Figure 1, CU2CL is a Clang plugin that ties into the main driver, allowing Clang to handle parsing and generate an AST, as during normal compilation. Our CU2CL then takes over and walks the generated AST to perform the rewrites. Of particular interest in designing CU2CL were the following Clang libraries: AST, Basic, Frontend, Lex, Parse, and Rewrite. These libraries facilitate file management (Basic), AST traversal and retrieval of information from AST nodes (AST), plugin interface and access to the compiler instance (Frontend), preprocessor access and token utilities (Lex), and the actual rewriting mechanism (Rewrite). *By uniquely composing the libraries and classes included within each, we created a robust CUDA-to-OpenCL translator with less than 2000 source lines of code (SLOC).*

B. 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 plugin, CU2CL provides an AST consumer that traverses the AST, searching for nodes of interest. While Clang's AST library provides several simple methods of traversing the tree, we use our own method totraverse the AST in a recursive descent fashion, using AST node iterators to recurse into each node's children. Figure 1 gives an overview of how CU2CL's translation procedure traverses the AST for both host and device code, locating nodes of interest and rewriting 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 categorize our approach as AST-driven and string-based.

In translating CUDA to OpenCL, this approach is quite useful. Because the two languages are based on C, giving a common ground between the two, only the CUDA-specific constructs must be translated to OpenCL. Compared to typical applications that make use of CUDA, the scope [20] of our translations are very small. Rewriting only the parts of interest and leaving everything else in the original source as it was, allows for most of the original structure and the original comments to be retained. 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 [21], leaving them intact is a requirement in CU2CL.

C. AST-Driven, String-Based Translation

We have determined 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 on common structures found in CUDA C, identifying which are of interest, and handling their rewriting in a modular way. 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, as described below. Finally, we demonstrate how to *locate* and *rewrite #include*s by leveraging a preprocessor, such as the one found in Clang's Lex library. All three of the above are insights that should aid future work in source-to-source translation based on the Clang framework, if not in more general cases.

1) Common Patterns: In translating CUDA constructs to OpenCL, some patterns 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 save for a few exceptions. CUDA API 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.

CUDA-specific type declarations may occur in several places. 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 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 be declared anywhere in a CUDA C application.

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

Fig. 3. Example of rewriting a CUDA API call which expects a pointer to an OpenCL call which does not.

Rewriting CUDA API calls can be logically separated into their distinct modules. However, for the purposes of CU2CL's source-to-source translation, it preferable to generalize as much of the rewriting as possible. In their arguments, the most important pattern is when a pointer to a data structure that is to be filled is passed in. The equivalent OpenCL API procedures instead return a new structure, as shown in Figure 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.

2) Recursively Rewriting Expressions: In performing string-based rewriting using Clang, several complications arise. Of these, being able to properly rewrite expressions and their subexpressions is of significant importance (and challenge). For example, when rewriting a kernel, one may encounter an expression such as the one in Figure 4. In order for the outer expression, `__powf`, to be rewritten, the argument expressions should be processed first. In the

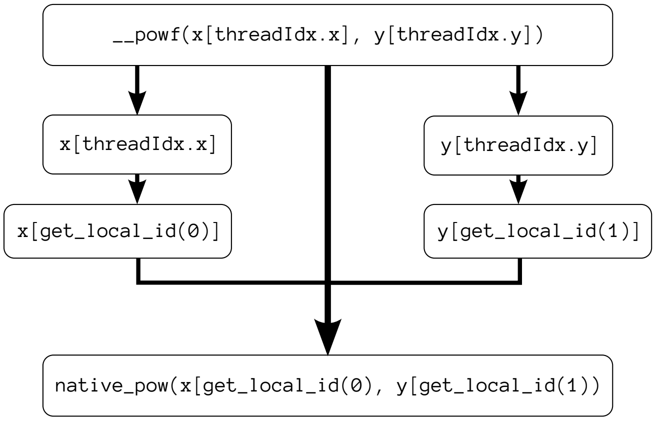

__powf(x[threadIdx.x], y[threadIdx.y])
x[threadIdx.x]
y[threadIdx.y]
x[get_local_id(0)]
y[get_local_id(1)]
native_pow(x[get_local_id(0), y[get_local_id(1))

Fig. 4. Example of rewriting an expression and its subexpressions.

```
procedure RewriteExpression($expr$)
    $type \leftarrow$ Type($expr$)
    if $type$ is interesting then
        return RewriteType($expr$)
    else
        $r \leftarrow$ SourceRange($expr$)
        for all $subexpr$ in SubExpressions($expr$) do
            $s \leftarrow$ RewriteExpression($subexpr$)
            if rewrite occurred then
                $subr \leftarrow$ SourceRange($subexpr$)
                ReplaceSource($subr, s$)
            end if
        end for
        return GetSourceWithRewrites($r$)
    end if
end procedure
```

Fig. 5. Algorithm detailing how expressions are recursively rewritten.

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. This library is also used at the top level for rewriting the input source files. 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 they represent. Following along with Figure 5, the source range is vital to performing string-based rewrites. When an expression that is not interesting—one containing no CUDA constructs—is encountered, CU2CL recurses down into its child subexpressions in a depth-first manner, invoking the recursive expression rewriting mechanism. If a subexpression is rewritten, the function returns 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.

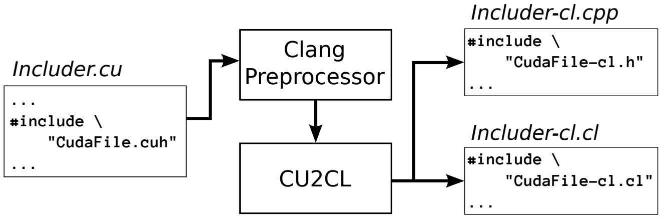


Fig. 6. Example of rewriting an #include directive.

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. Because #includes are not resident in the AST we transform, this rewriting has been implemented using the Clang driver's preprocessor, as shown in Figure 6. CU2CL registers a callback with the preprocessor that 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: (1) removing #includes pointing to CUDA and system header files and (2) 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*) 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 #includes so that they point to the new OpenCL files. Figure 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 runtime compilation of device code, so it is not #included.

D. Challenges

1) Maintainable Code: In some cases, automatically translating CUDA to OpenCL makes 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. This is a direct consequence of what the C preprocessor is capable of and little can be done to mitigate the issue.

2) Rewriting Macros: In Clang, macros are represented as a series of tokens. While its libraries can provide access to the tokens they are simply raw and unparsed. Hence, the process of rewriting macros would require at least partial parsing of the tokens contained within. This is a complex task that is beyond the scope of this paper.

3) Use of Closed-Source Libraries: When CUDA applications make use of closed-source libraries built on top of CUDA, such as the CUBLAS or CUFFT libraries in the CUDA toolkit, CU2CL (or any other pure CUDA translator) cannot fully translate these applications because the library will continue to expect CUDA constructs. As a result, users would need to either re-implement those libraries from scratch in OpenCL or find other libraries written in OpenCL that with equivalent functionalities. On the other hand, if a CUDA library's source code is available, it could be translated using CU2CL and the problem resolved.

4) Function Rewriting: User functions expecting CUDA constructs or results from CUDA calls as arguments cannot be handled entirely without rewriting the functions first. This includes utility functions like those in the CUDA SDK's `cutil` library. This is also seen in benchmark suites where common CUDA code is shared across applications, e.g., SHOC.

V. Evaluation

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

A. Translation Performance

While CU2CL will ideally only be run once on a given CUDA application, the speed at which CU2CL translates the CUDA source code to OpenCL source code is a performance metric of interest, particularly if the end user wishes to convert multiple CUDA applications from the well-established CUDA ecosystem. Thus, we evaluate its speed of translation on several GPU applications from the CUDA SDK and Rodinia benchmark suite. For each application, we averaged the total time to translate the code from CUDA to OpenCL over ten runs. This translation time includes the time for the Clang driver to perform parsing and semantic analysis of the program, in addition to CU2CL's translation procedure. Table IV summarizes our results.

The test applications vary in length from more than a hundred source lines of code (SLOC) to nearly a thousand SLOC. However, one can see that the translation time is not strictly dependent on the length. In general, programs with more CUDA constructs or more complicated ones tend to take longer to translate. Even so, CU2CL takes no more than a few seconds to translate many applications, making it a feasible choice for porting a large number of CUDA programs.

B. Translated Application Performance: Auto vs. Manual

We evaluate the performance of three automatic vs. manually translated CUDA-to-OpenCL codes: vectorAdd from the CUDA SDK and Needleman-Wunsch and SRAD from the Rodinia benchmark suite. vectorAdd is an application that generates two random vectors in host memory and copies them 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.

Source	Application	Translation Time	Lines
CUDA SDK	asyncAPI	2.96s	136
	bandwidthTest	5.76s	891
	BlackScholes	5.52s	347
	matrixMul	5.53s	351
	scalarProd	2.96s	171
	vectorAdd	2.59s	147
Rodinia	Back Propagation	2.68s	313
	Breadth-First Search	2.67s	306
	Hotspot	2.64s	328
	Needleman-Wunsch	2.71s	418
	SRAD	2.71s	541

TABLE IV
TIME FOR CU2CL TO TRANSLATE AN APPLICATION RELATIVE TO THE LINES OF CODE IN THE ORIGINAL CUDA APPLICATION.

Needleman-Wunsch is a global sequence aligner that is 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 that removes noise from images produced by ultrasonic or radar imagery applications and doing so without losing any of the important features present in the pictures. 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.

Table V summarizes the performance comparisons between the original CUDA code, CU2CL's automatically-generated OpenCL, and our manually-ported OpenCL. Each code was executed a total of ten times and their runtimes were averaged.

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.

On the other hand, SRAD's OpenCL performance is roughly 25% worse than its CUDA version. The OpenCL Needleman-Wunsch code performs about 30% worse than the CUDA

Application	CUDA	Automatic OpenCL	Manual OpenCL
vectorAdd	0.050s	0.051s	0.052s
Needleman-Wunsch	6.65s	8.77s	8.77s
SRAD	1.25s	1.55	1.54s

TABLE V
RUNTIMES OF FOUR CUDA APPLICATIONS AND THEIR OPENCL PORTS ON AN NVIDIA GTX 280

version. These results are typical as the NVIDIA OpenCL implementation is known to not perform as many optimizations as CUDA does [4].

Anecdotally, while it took a typical computer science graduate student three *weeks* to manually translate the three above codes from CUDA to OpenCL, our robust CU2CL prototype automatically translated these three programs in $2.59 + 2.71 + 2.71 = 8.01$ *seconds*, as noted in Table IV.

C. Translator Coverage

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 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 nearly in their entirety. Table VI 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 the manual changes, none are particularly difficult to handle and automated support for these will be added in the coming weeks, as CU2CL continues to evolve.

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 VI
CU2CL'S AUTOMATIC TRANSLATION COVERAGE OF A RANGE OF APPLICATIONS.

VI. FUTURE WORK

There still remains work to be done that could extend CU2CL's capabilities. To begin, we aim to support larger subsets of the CUDA runtime API, in particular, the texture management module and several procedures found in the memory management module. Along the same lines, we plan to add support for the CUDA driver API. In terms of CU2CL's design, we plan to identify other common patterns in CUDA that will allow for further modularization of our translator.

Finally, we will support the application of device-specific optimizations as a backend to our CU2CL translator. Why? Preliminary results from running CU2CL's automatically-translated OpenCL applications in Section V on an AMD Radeon HD 5870 (rather than an NVIDIA GTX 280) deliver mediocre results. Although the AMD GPU has higher theoretical peak performance than the NVIDIA GTX 280, its execution times are 0.075s, 15.24s, and 2.11s for vectorAdd, Needleman-Wunsch, and SRAD, respectively. These values are all at least 50% worse than the OpenCL run times on the NVIDIA GPU presented in Table V. So, while we have enabled the potential to run CUDA GPU codes on any OpenCL-capable device, it does *not* mean that the these GPU codes will perform well without device-specific optimizations, as shown in [4], [5]. Thus, as long-term future work, CU2CL will automatically apply the manual optimizations identified in [4], [5] on its generated OpenCL code.

VII. CONCLUSION

We have presented CU2CL, an automated source-to-source translator from CUDA to OpenCL. By leveraging the Clang compiler framework, we took advantage of its powerful source-level tools to create a robust translator in less than 2000 source lines of code.

In designing and implementing CU2CL, we determined useful patterns that may be used in future Clang-based source-to-source translators. We also demonstrated methods of recursively rewriting expressions and of efficiently rewriting *#include* directives through the use of Clang's preprocessor.

We have shown that the currently supported subset of CUDA covers most of the CUDA runtime API found in many applications. In practice, CU2CL can translate several application almost in their entirety with little to no manual effort. 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 that are manually translated.

ACKNOWLEDGMENT

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

REFERENCES

[1] NVIDIA, "Cuda zone," http://www.nvidia.com/object/cuda_home_new.html, [Online; accessed 15-April-2011].

[2] K. Group, "Opencl," http://www.khronos.org/opencl/, [Online; accessed 15-April-2011].

[3] M. Harvey and G. De Fabritiis, "Swan: A tool for porting CUDA programs to OpenCL," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093–1099, Apr. 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0010465511000117

[4] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of OpenCL programs," in *The Fifth International Workshop on Automatic Performance Tuning (iWAPT2010)*, 2010. [Online]. Available: http://vecpar.fe.up.pt/2010/workshops-iWAPT/Komatsu-Sato-Arai-Koyama-Takizawa-Kobayashi.pdf

[5] M. Daga, T. R. W. Scogland, and W.-c. Feng, "Architecture-Aware Optimization on a 1600-core Graphics Processor," Department of Computer Science, Virginia Tech, Blacksburg, Virginia, Tech. Rep. TR-11-08, 2011.

[6] "Opencl$^{\text{TM}}$ 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].

[7] S. Lee, S. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2009, pp. 101–110. [Online]. Available: http://portal.acm.org/citation.cfm?id=1504194

[8] J. Stratton, S. Stone, and W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," *Languages and Compilers for Parallel Computing*, pp. 16–30, 2008. [Online]. Available: http://www.springerlink.com/index/x361x32j1q840072.pdf

[9] S. Lee, T. Johnson, and R. Eigenmann, "Cetus–an extensible compiler infrastructure for source-to-source transformation," *Languages and Compilers for Parallel Computing*, no. 9703180, pp. 539–553, 2004. [Online]. Available: http://www.springerlink.com/index/52gtac2nllw9rh7v.pdf

[10] G. Diamos, "The design and implementation ocelot's dynamic binary translator from ptx to multi-core x86," *Center for Experimental Research in Computer Systems*, 2009. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.163.8500\&amp;rep=rep1\&amp;type=pdf

[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *Optimization*, 2004. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/CGO.2004.1281665

[12] R. Dominguez, D. Schaa, and D. Kaeli, "Caracal: Dynamic translation of runtime environments for gpus," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, no. March. ACM, 2011, p. 7. [Online]. Available: http://portal.acm.org/citation.cfm?id=1964186

[13] D. Nandakumar, "Automatic Translation of CUDA to OpenCL and Comparison of Performance Optimizations on GPUs," 2011. [Online]. Available: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:Automatic+Translation+of+CUDA+to+OpenCL+and+Comparison+of+Performance+Optimizations+on+GPUs\#0

[14] NVIDIA, "Cuda toolkit & sdk," http://developer.nvidia.com/cuda-toolkit-sdk, [Online; accessed 15-April-2011].

[15] "Rodinia: A benchmark suite for heterogeneous computing," http://lava.cs.virginia.edu/Rodinia/rodinia.htm, [Online; accessed 10-July-2011].

[16] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Issues*, vol. 2, no. 3, pp. 215–226, 2000. [Online]. Available: http://www.worldscinet.com/abstract?id=pii:S0129626400000214

[17] E. Visser, "Program transformation with Stratego/XT: Rules, strategies, tools, and systems," *Domain-Specific Program Generation*, no. February, 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.6036\&amp;rep=rep1\&amp;type=pdf

[18] I. Baxter, C. Pidgeon, and M. Mehlich, "DMS: Program Transformations for Practical Scalable Software Evolution lution," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 625–634. [Online]. Available: http://portal.acm.org/citation.cfm?id=999466

[19] "clang: a c language family frontend for llvm," http://clang.llvm.org/, [Online; accessed 15-April-2011].

[20] J. Van Wijngaarden and E. Visser, "Program transformation mechanics: a classification of mechanisms for program transformation with a survey of existing transformation systems," *Technical report UU-CS*, no. 2003-048, 2003. [Online]. Available: http://igitur-archive.library.uu.nl/math/2007-1123-200750/UUindex.html

[21] M. 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*, pp. 131–141, 2001. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=972674