Parallel Computing 39 (2013) 769–786

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Characterizing the challenges and evaluating the efficacy of a CUDA-to-OpenCL translator [☆]

Mark Gardner, Paul Sathre, Wu-chun Feng [*], Gabriel Martinez

Department of Computer Science, Virginia Tech, Blacksburg, VA 24060, United States

ARTICLE INFO

Article history:
Available online 1 October 2013

Keywords:
Source-to-source translation
GPU
CUDA
OpenCL

ABSTRACT

The proliferation of heterogeneous computing systems has led to increased interest in parallel architectures and their associated programming models. One of the most promising models for heterogeneous computing is the *accelerator model*, and one of the most cost-effective, high-performance accelerators currently available is the general-purpose, graphics processing unit (GPU).

Two similar programming environments have been proposed for GPUs: CUDA and OpenCL. While there are more lines of code already written in CUDA, OpenCL is an open standard that supports a broader. Hence, there is significant interest in automatic translation from CUDA to OpenCL.

The contributions of this work are three-fold: (1) an extensive characterization of the subtle challenges of translation, (2) CU2CL (CUDA to OpenCL) — an implementation of a translator, and (3) an evaluation of CU2CL with respect to coverage of CUDA, translation performance, and performance of the translated applications.

1. Introduction

Recent trends in processor architectures utilize available transistors to provide large numbers of execution cores, and hence threads, rather than attempting to speed-up the execution of a single thread or a small number of threads. This has led to general interest in parallel architectures and programming models even outside of the high-performance computing (HPC) realm. The accelerator model, where general-purpose computations are performed on the central processing unit (CPU) and data- or task-parallel computations are performed on specialized accelerators, is one of the models being proposed as a way to program heterogeneous computing architectures. By leveraging the economics of graphics cards, particularly gaming cards, graphics processing units or GPUs in graphics cards have been particularly successful in supporting the accelerator model.

GPUs were originally designed to perform a set of computations on a large number of picture elements or pixels simultaneously. Besides producing highly realistic, real-time graphics for gaming, this characteristic could also be harnessed to execute parts of scientific computations in parallel with high performance. Examples of these computations include simulating the physical movements of atoms and molecules as part of an n-body molecular dynamics problem and searching for alignments in nucleotide or protein sequences, as done in [1,2], respectively. When used for more than graphics computations, GPUs are called general-purpose GPUs or GPGPUs. (For brevity and convenience, we refer to GPGPUs simply as GPUs for the remainder of this paper.)

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

* Corresponding author. Tel.: +1 540 231 1192; fax: +1 540 231 9218.
E-mail address: feng@cs.vt.edu (W.-c. Feng).

0167-8191/$ - see front matter © 2013 Published by Elsevier B.V.
http://dx.doi.org/10.1016/j.parco.2013.09.003

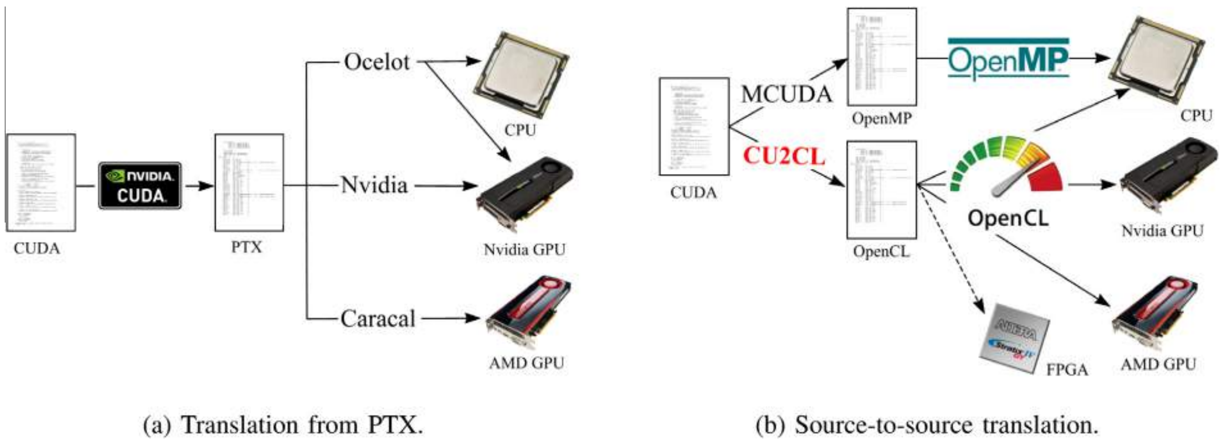


Fig. 1. Two of the approaches for translating CUDA source.

In the accelerator model, the overall logic of an application runs on the CPU, possibly utilizing multiple cores to execute several threads simultaneously. Threads delegate computations to accelerators by spawning off parallel computations, called *kernels*, to the GPUs. At appropriate points during execution on the CPU, the CPU sends data to the GPUs, and the kernels are invoked. The CPU may continue to perform other computations in parallel with the kernels until such time as the results are needed, whereupon the results are transferred from the GPU back to the CPU. Because GPUs have hundreds or even thousands of threads (compared with tens of threads in CPUs), effective utilization of GPUs for computations can accelerate parallel computations tremendously. Because the accelerator model is a straightforward extension to the programming models commonly used, it is readily understood by programmers of widely differing skill levels. As a result, GPU programming has taken off.

Initially, general-purpose computations were programmed using the shader languages developed for graphics operations — a tedious task. However, as the use of GPUs has increased, languages and tools for more conveniently specifying general-purpose computations have been developed. One of the earliest and most successful programming environments to date is NVIDIA's CUDA.

In CUDA [3], kernels are written in a variant of C++, which has additional data types and operations suited to computations on larger chunks of data, according to the accelerator model. It also provides synchronization primitives that ensure correctness. On the CPU side, CUDA extends C++ by including special syntax for invoking kernels. Because of these extensions, however, the CUDA programming environment cannot use standard C++ compilers, and instead, must use a compiler for the extended language. Although CUDA is widely used, it is largely tied to NVIDIA's freely available but proprietary hardware and software development kit (SDK).

OpenCL, short for Open Computing Language, is another programming environment that implements an accelerator programming model that is similar to CUDA. It is an open standard that is maintained by the Khronos group. The OpenCL API consists of a C library supporting device programming in the C99 language (rather than as extensions to C++ as in the CUDA API). As a result, OpenCL can take advantage of existing tools and compilers for a wide variety of host and GPU platforms. Porting OpenCL to a different host platform is a matter of providing an implementation of the runtime library that conforms to the standard. In contrast, CUDA requires a compiler that understands the extended syntax as well as an appropriate runtime library.[1] Thus, the APIs of the two programming environments differ quite a bit. However, learning to program in one yields the same mental model of GPGPU computations as learning in the other, allowing the other programming environment to be understood readily once the differences in syntax and library are taken into account.

With a choice in programming environments comes the need to choose. CUDA has the largest installed base and many time-saving libraries for important functions such as FFT. However, it is only supported on NVIDIA hardware. OpenCL, on the other hand, is an open standard that is supported on a variety of mainstream devices: NVIDIA GPUs, AMD GPUs, and x86/x86–64 CPUs from Intel and AMD. Support is also available for some system-on-chip (SoC) devices [5], including the ARM Cortex-A9 CPU [6] and the PowerVR SGX GPU [7]. Additionally, Intel intends to support OpenCL on their Many Integrated Cores (MIC) architecture [8], and Altera already has a program to develop an OpenCL environment for their FPGAs [9], as represented as a dashed line in Fig. 1(b). Due to the greater functional portability provided by OpenCL (and in spite of the weaker OpenCL library support for important functions), many are choosing OpenCL as their programming environment of choice.

With the great interest in OpenCL comes a challenge: organizations have a large investment in CUDA codes and yet would like to take advantage of wider deployment opportunities afforded by OpenCL. Thus, there needs to be a translator from

[1] Although NVIDIA recently contributed an open-source compiler based upon LLVM to aid researchers [4], there is only a single (proprietary) implementation of the runtime library.

CUDA to OpenCL that not only allows compilation in the new environment but also yields maintainable source code such that development can continue directly in OpenCL. To this end, we created CU2CL, an automatic CUDA-to-OpenCL source-to-source translator that also preserves the comments and formatting in the original source code.

The contributions of this work are three-fold:

1. A characterization of the mapping between CUDA and OpenCL with particular emphasis on areas where the semantics are not trivially equivalent,
2. A discussion of a prototype implementation of an automatic translator from CUDA to OpenCL called CU2CL.
3. An evaluation of the CU2CL translator in three areas: (a) coverage of CUDA constructs translated automatically, (b) translation time, and (c) a comparison of the execution time for automatically translated applications.

The remainder of the paper is as follows: Section 2 discusses related work, followed by relevant background on CUDA and OpenCL in Section 3. Section 4 gives an extensive characterization of many of the challenges in translating CUDA to OpenCL. Section 5 introduces the CU2CL translator, and Section 6 evaluates its performance. Section 7 outlines future work and Section 8 summarizes the contribution of the work.

2. Related work

Several projects exist that enable the running of CUDA source code on hardware platforms other than NVIDIA GPU hardware. There are three main approaches: (1) translating the Parallel Thread Execution (PTX) intermediate representation (IR) from the `nvcc` compiler, (2) translating from one source to another, and (3) modifying the original source code to utilize an abstract interface for which different implementations are provided.

Fig. 1(a) shows the approach taken by the Ocelot project [10] and by Caracal [11]. In this approach, the CUDA source code is first translated to the PTX IR using the `nvcc` compiler from the CUDA SDK. Instead of immediately sending the PTX representation to the device driver, which finishes the compilation and executes the code (as shown by the work flow in the middle of the diagram), Ocelot parses the PTX and utilizes the Low-Level Virtual Machine (LLVM) toolkit to perform transformations and optimizations before generating code for the target architecture. Target architectures currently include x86 CPUs from Intel and AMD and the IBM Cell Broadband Engine. Ocelot can also generate PTX as output from the transformed representation for NVIDIA hardware. Caracal builds upon Ocelot by adding a just-in-time compilation step that translates PTX to AMD's Compute Abstraction Layer (CAL) in order to run CUDA applications on AMD GPU hardware for which there is no CUDA runtime. In both cases, starting with the PTX IR makes it possible to translate code to run on other hardware platforms even if the source code is not available.

Fig. 1(b) shows an alternative translation approach taken by MCUDA [12] and our CU2CL [13]. In contrast with Ocelot and Caracal, which both start with the PTX IR, CU2CL and MCUDA perform a source-to-source translation on the original CUDA source code to generate a semantically equivalent program targeted to another hardware platform. The native compilers, linkers, and other development tools for that platform are then used to prepare the computation for execution. The target programming platform for MCUDA is OpenMP, which runs on a variety of CPU platforms, including non-x86 CPUs. The target programming platform for CU2CL is OpenCL, which runs on a variety of CPUs and GPUs. (Support for other devices, such as FPGAs, are still under development, as represented by a dashed line in the figure.).

The source-to-source approach possesses many advantages. First, it leverages the considerable effort that has been spent over the years creating robust and high-performance tools for the various hardware platforms. Second, automatic translation into another source language gives the option of continued code development in the new environment. This is particularly an advantage if migration is desired. Third, it preserves more of the high-level semantics, and hence, more easily enables transforming the code to achieve additional goals, such as improved performance. As shown in [1] and many other publications the optimizations necessary for better performance differ depending on whether the GPUs are from AMD or NVIDIA. Optimizations necessary for better performance on CPUs are also different from those required for GPUs.

The original source code is a representation of the intent of the programmer and hence contains information that can get lost in the translation to an intermediate form or becomes more difficult to reconstruct from that form. With additional information, better optimization choices can be made by the tool chain without human intervention.

Note that these two approaches, starting with PTX or starting with the source code are beneficial in different circumstances. Translation from PTX is viable whether or not the original source code is available. Source-to-source translation is beneficial for organizations that value platform diversity, desire to migrate their code base, or are invested in existing vendor-specific tool chains.

CU2CL targets OpenCL in order to support the wide variety of multi- and many-core processors and accelerators in the market. The breadth of hardware platforms supported by OpenCL is very compelling, particularly since it gives the flexibility to purchase the platform with the best performance.

It should be noted that there are two CUDA APIs. The high-level runtime API, which provides reasonable defaults for many runtime parameters, and the lower-level device API, which gives the programmer much more control but at the expense of programmability. Most CUDA applications use the runtime API because it requires less attention to detail. CU2CL initially assumes that the CUDA source uses the more common runtime API as that makes the tool more immediately useful.

Translating the CUDA runtime API is also the greater intellectual challenge as the CUDA device API is very similar to the OpenCL API. CU2CL will be extended to also support the CUDA driver API in the future.

As mentioned earlier, there is a third approach to the problem, represented by Swan [14]. Instead of translating the intermediate or source languages, the programmer rewrites all CUDA API calls with Swan equivalents. The code is then compiled and linked with one of two currently supported Swan libraries, `libswan_ocl` or `libswan_cuda`, which permits execution on either of the main GPU platforms. The main disadvantages are the labor involved (which is currently only partially automated) and the fact that the source code is no longer CUDA or OpenCL and hence does not have widespread industry support.

3. Background

This section presents an overview of CUDA and OpenCL before delving into the details of automatic translation. As mentioned earlier, both CUDA and OpenCL are designed to support the accelerator model of computing. Both have provisions for specifying and launching kernels on compute devices, for managing memory, for synchronizing, etc. However, CUDA is more tightly focused on GPUs and provides many GPU-centric features, whereas OpenCL takes a more platform-agnostic approach.

One of the challenges in translating between the two is understanding the terminology. Table 1 lists various CUDA terms and their OpenCL counterparts. The CUDA terms will be used in the discussions as they are likely more familiar to the reader. The differences in terminology are mentioned as applicable below.

3.1. Work allocation model

Work is allocated to the *streaming multiprocessors* on a GPU according to a multidimensional *grid* of *blocks*, as shown in Fig. 2(a). Each block specifies numerous threads, also possibly in a multidimensional configuration. The configurations are specified in the host code during a kernel function invocation. For good performance, the number of threads need to be chosen so that there are sufficient threads to hide the latency of memory accesses or other operations and yet sufficient work per thread to amortize the cost of invoking the kernel.

3.2. Memory model

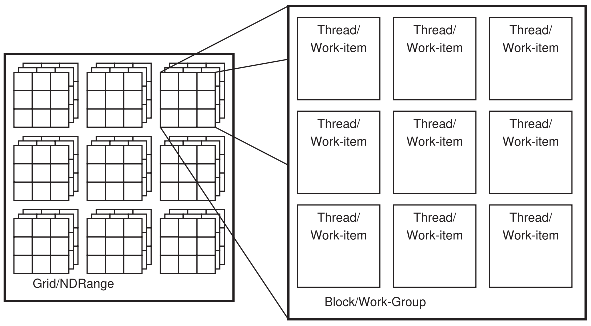
Both CUDA and OpenCL have three separate memory spaces. CUDA refers to these memory spaces as *global memory* — off-device and accessible by all threads in all blocks; *shared memory* — on-device and available to all threads in a block; and *local memory* — owned by one thread. In addition to these memories, 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 limited in size. Furthermore, kernels must use special built-in functions to access data residing in these regions. In general, device memory must be explicitly allocated through CUDA API calls and is usually initialized by copying data from host memory. Fig. 2(b) shows how the memory spaces are laid out hierarchically.

3.3. Host API

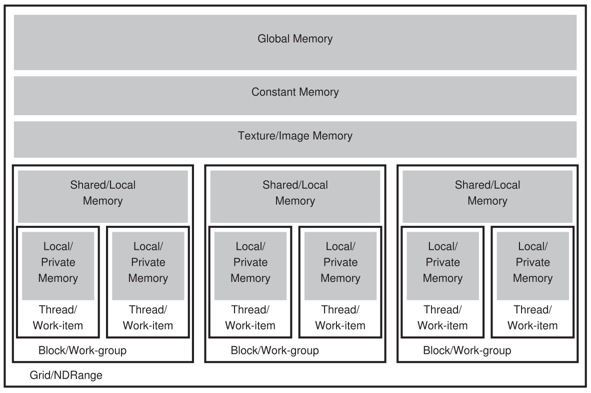
Two CUDA APIs exist for programming host-side code: a low-level *driver* API and a high-level *runtime* API. The driver API gives the programmer great flexibility and control but also requires more setup and configuration to be done explicitly. In addition, CUDA includes an extension of C providing special features like concise kernel launch syntax, known as CUDA C. As an example, consider the host code needed to invoke a `matrixMul` kernel [15]. Fig. 3(a) shows the code for a CUDA C launch while Fig. 3(b) and (c) show the code for the CUDA runtime and driver APIs, respectively. The runtime and driver API versions set the arguments for the kernel invocation explicitly while the CUDA C variant accomplishes the same task with a more

Table 1
CUDA and OpenCL terminology.

CUDA	OpenCL
GPU	Device
Multiprocessor	Compute unit
Scalar core	Processing element
Kernel	Program
Block	Work-group
Thread	Work-item
Global memory	Global memory
Shared memory	Local memory
Local memory or registers	Private memory
Constant memory	Constant memory
Texture memory	Image memory



(a) Work allocation model



(b) Memory model

Fig. 2. Overview of the CUDA and OpenCL models.

```
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(WC / threads.x, HC / threads.y);
matrixMul<<<grid,threads>>>(C, A, B, WA, WB);
```

(a) CUDA C

```
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(WC / threads.x, HC / threads.y);
cudaConfigureCall(grid, threads, 0, 0);
cudaSetupArgument(C, 0);
cudaSetupArgument(A, 4);
cudaSetupArgument(B, 8);
cudaSetupArgument(WA, 12);
cudaSetupArgument(WB, 16);
cudaLaunch("matrixMul");
```

(b) CUDA runtime API

```
cuFuncSetBlockShape(matrixMul, BLOCK_SIZE, BLOCK_SIZE, 1);
cuFuncSetSharedSize(matrixMul, 2 * BLOCK_SIZE
  * BLOCK_SIZE * sizeof(float));
cuParamSeti(matrixMul, 0, C);
cuParamSeti(matrixMul, 4, A);
cuParamSeti(matrixMul, 8, B);
cuParamSeti(matrixMul, 12, WA);
cuParamSeti(matrixMul, 16, WB);
cuParamSetSize(matrixMul, 20);
cuLaunchGrid(matrixMul, WC/BLOCK_SIZE, HC/BLOCK_SIZE);
```

(c) CUDA driver API

```
size_t localWorkSize[] = {BLOCK_SIZE, BLOCK_SIZE};
size_t globalWorkSize[] = {shrRoundUp(BLOCK_SIZE, WC), shrRoundUp(BLOCK_SIZE, workSize)};
clSetKernelArg(matrixMul, 0, sizeof(cl_mem), (void *) &C);
clSetKernelArg(matrixMul, 1, sizeof(cl_mem), (void *) &A);
clSetKernelArg(matrixMul, 2, sizeof(cl_mem), (void *) &B);
clSetKernelArg(matrixMul, 3, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0);
clSetKernelArg(matrixMul, 4, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0);
clEnqueueNDRangeKernel(commandQueue, matrixMul, 2, 0,
  globalWorkSize, localWorkSize, 0, NULL, &GPUExecution);
clFinish(commandQueue);
```

(d) OpenCL API

Fig. 3. Comparison of the CUDA and OpenCL APIs.

succinct syntax. The attention to detail needed for the driver API is generally much greater, so programmers tend to use CUDA C in combination with the runtime API whenever possible.

Table 2
CUDA modules and the OpenCL equivalents.

CUDA	Sample call	OpenCL
Thread	`cudaThreadSynchronize`	Contexts and command queues
Device	`cudaSetDevice`	Platforms and devices
Stream	`cudaStreamSynchronize`	Command queues
Event	`cudaEventRecord`	Events
Memory	`cudaMalloc`	Memory objects

Table 3
CUDA and OpenCL data structures.

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 4
CUDA built-ins 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)`

By way of comparison, Fig. 3(d) shows the equivalent host code for invoking the kernel in OpenCL.[2] The number of lines of code in the OpenCL example is the same as with the CUDA driver API example. Further, there is a close correspondence between the two. Consequently, translating between the CUDA driver API and OpenCL API is straightforward and a much easier task to automate. However, most CUDA codes use the runtime API whenever possible due to its ease of programmability [16,17]. This is why CU2CL supports the CUDA runtime API now and why supporting the driver API is left for future work.

4. Characterization

Two primary concerns drive our source-to-source approach. First, the translated source should not deviate from the semantics of the original source. This is a requirement for *functional portability* or *functional correctness*. Second, the translated code should perform as well as the original source code on similar hardware platforms and on new hardware platforms. This requirement is called *performance portability*.

Functional portability is the most important of the two concerns as it does not matter how fast the code runs if it gets the wrong answer. Once functional portability has been obtained, the performance differences between the two computing platforms can be addressed in order to achieve performance portability.

This paper focuses exclusively on functional portability, leaving the issue of performance portability for future work. The remainder of this section will discuss issues that make CUDA-to-OpenCL source-to-source translation challenging. It will also outline how the problems are solved in the CU2CL translator.

4.1. Translation challenges

At first glance, translating CUDA to OpenCL appears to be a straightforward mapping process. While most CUDA constructs map one-to-one to OpenCL, not all do, as shown in Tables 2–4. As a result, translating certain parts of CUDA requires a deeper understanding of both APIs to identify suitable corresponding constructs. Furthermore, these tables provide only a

[2] Note: The closest equivalent to the CUDA runtime API is to use the OpenCL C++ wrapper API to create succinct abstractions. There are as yet no standard abstractions.

```
float *newDevPtr;
...
cudaMalloc((void **) &newDevPtr, size);
```

(a) Original CUDA source.

```
cl_mem newDevPtr;
...
newDevPtr = clCreateBuffer(clContext,
        CL_MEM_READ_WRITE, size, NULL, NULL);
```

(b) Rewriten OpenCL source.

Fig. 4. Example of rewriting the cudaMalloc API call.

high-level view of the translation process; in practice, more sophisticated techniques are required to perform the translations. For example, in some cases, data must be *tracked throughout the lifetime of the translation* before certain translations can be finalized. Such is the case when rewriting device pointers to `cl_mem` data types, as the rewrite must propagate through types found in parameters and `sizeof` expressions.

As Table 2 shows, each of the CUDA modules possesses a corresponding OpenCL equivalent. However, the bigger challenges in CUDA-to-OpenCL translation only become apparent upon deeper inspection. For example, Table 3 shows some CUDA data structures and their OpenCL equivalents. Some have direct equivalents: `dim3` vs. `size_t[3]`. Others have no direct equivalent, e.g., `cudaDeviceProp`, and have to be synthesized from OpenCL data structures and function calls. The most challenging translations are ones like the CUDA device pointers and their semantically similar, but functionally different, OpenCL `cl_mem` structures. (This particular challenge is discussed in greater depth in Section 4.1.1).

Built-in structures and functions are another area where there are similarities and differences, as shown in Table 4. CUDA indices and dimensions (both grid and block) are variables that have OpenCL counterparts accessible via built-in functions. However, there are some built-in functions, such as `__threadfence()` that have no equivalent functionality in the current version of OpenCL. As a consequence, workarounds must be synthesized.

Below are some specific examples of the challenges in translating CUDA to OpenCL.

4.1.1. Pointers

CUDA refers to device data buffers through pointers to the type of data stored, similar to standard C dynamically allocated buffers. However, rather than initializing a pointer to the dynamically allocated region with a standard malloc call, in general these buffers must be initialized by a call to one of the `cudaMalloc*` variants specified in the CUDA runtime. The device buffer pointers are passed to a CUDA kernel launch in nearly identical fashion as the passing of their standard C counterparts to a host-side function call. Both host and device share the same pointer type for the buffer.

However, OpenCL device buffers use the `cl_mem` type to represent all buffers on the host side, and standard pointer types on the device. When a pointer type is specified as a parameter to an OpenCL kernel, any `cl_mem` buffer which resides in the correct memory access space can be specified for the parameter, regardless of the buffer's intended type. These buffers are then interpreted as an array of the appropriate type by the device code, similar to CUDA. However, this draws attention to an important difference between how the two APIs reference buffers on the host side.

Commonly, we have observed that CUDA device memory allocation and kernel calls reside in separate functions within an application. In these cases, the buffer pointer is frequently passed as a parameter to the kernel call's wrapping function. From a syntax perspective, this pointer parameter has no indication whether the parameter represents a host- or device-side buffer. However, once the device buffer is translated to OpenCL, the wrapped kernel call will require a `cl_mem` type for all device buffers, necessitating a rewrite of the parameter, as well as any functions lower on the call stack which passed the parameter through. As CUDA device buffer pointers have no readily-observable syntax declaring them as such, accurate propagation of type rewrites across these functions is made significantly more complex.

The current translator prototype does not yet perform this full call stack propagation of `cl_mem` types. It only performs a type translation within the scope containing the declaration of the CUDA device buffer pointer. For example, CU2CL translates the CUDA buffer allocation shown in Fig. 4(a) to a form similar to Fig. 4(b). Proper translation is assured if the pointer declaration shares the same scope as the kernel call. Complete type propagation is the subject of future work. (One potentially-viable approach we are currently considering is the simple method of reducing the scope of the `cl_mem` translation, by allowing the buffer to retain the original pointer type and simply providing explicit casts at buffer allocation and kernel invocation, the only two points at which it is required to be a `cl_mem` or should ever be accessed by the host.)

4.1.2. Pre-processing

Source-to-source translation of preprocessed languages, such as C/C++, as well as domain-specific variants such as CUDA and OpenCL, is known to be difficult [18]. The code that the compiler sees can be dramatically different than the code in the source file.

One solution is to run the source code through the preprocessor prior to translation. While this approach can achieve functional correctness, it has the tendency to make the translated source code much less maintainable as features like constants are expanded into opaque values.

An alternative approach, taken by CU2CL, is to translate the un-preprocessed tokens based on guidance from the preprocessed code in the hope that the preprocessing done later will not lead to syntactically or semantically invalid code. While this heuristic cannot guarantee a clean translation, it works well in practice as tricky macro preprocessing is a software maintenance nightmare and is generally avoided.

4.1.3. Separate compilation

Separate compilation is an important tool in the development of software. However, it poses a significant challenge in source-to-source translation for languages like C/C++, where global state is often implicit until the linking stage. This is an important issue as nearly all but the most trivial applications are composed of multiple, separately compiled source files.

As the result of separate compilation, the declaration of a feature and its use may occur in separate files making it challenging to ensure that compatible translations happen in each file. This issue is especially prevalent in code that performs device initialization, memory management, or kernel definition.

As an example, consider the translation of pointers to CUDA memory buffers allocated with `cudaMalloc`, shown in Fig. 4(a), to OpenCL's `cl_mem` which is a pointer to an opaque type, shown in Fig. 4(b). Using a pointer to an opaque type may make it easier to implement OpenCL on widely divergent devices, but it makes translation significantly more difficult as it requires propagation of a type rewrite across all functions which utilize the buffer pointer as a function parameter or return value. If declaration and uses are split across multiple source files — particularly when used as a kernel parameter — care must be taken to ensure the same type change propagates throughout all the sources files.

4.1.4. Precompiled binary code

Another challenge to automatic translation is the use of precompiled or hand-tuned binary code in CUDA applications. A fair number of CUDA applications from the CUDA SDK utilize precompiled libraries and thus thwart the source-to-source translator. Translations of the libraries' header files can be performed but the precompiled code remains inaccessible. This is an important problem as there are a growing number of high-quality high-performance CUDA libraries becoming available and being used.

Not all is lost however. By their very nature, such libraries, e.g., cuFFT, implement common functionality that is widely used. Libraries implementing similar functionality are becoming available for OpenCL, e.g., AMD Accelerated Parallel Processing Math Libraries (APPML) FFT, enabling translators to map from one library to another. As equivalent libraries become available, the translator can be extended to perform the translation. Currently, however, the CUDA library calls pass through the translator unchanged and hence require manual intervention.

A complimentary approach is to extend Ocelot to translate precompiled binaries to OpenCL code and use the translated kernel code in the place of the binary.

4.1.5. C++ syntax

CUDA supports both C and C++ syntax for host and kernel code while OpenCL supports C and C++ bindings for host code but only C99 with select extensions for kernel code. This makes translating CUDA kernel code containing C++ problematic.

Function templates are a concrete example of a C++ feature in CUDA kernels. Other than stipulating that __global__ functions with private scope cannot be instantiated, CUDA provides full support for function templates. Fully-automated translation would require the parsing of function templates and the creation of individual kernel functions specialized for each unique instantiation of a kernel function template.

The current approach is to wait for the OpenCL standard to be extended to support C++ in kernels. Thus, this is another area that temporarily requires human intervention.

4.1.6. Literal arguments

There is a subtle difference in the kernel launch semantics of CUDA and OpenCL, particularly in the way that kernel function parameters are specified.

While CUDA provides several methods of launching device kernels, by far the most popular uses the CUDA C kernel invocation syntax (see Fig. 3(a)), with the semantics that kernel arguments are passed by value. OpenCL, on the other hand, specifies kernel arguments through calls to `clSetKernelArgs` (see Fig. 3(d)), which implements pass-by-reference semantics.

For many kernel arguments, it is sufficient to transform the value represented by a variable to a pointer to the variable using the address-of operator &. This approach does not work for literal constants or macro expressions. In this case, the translator must infer the type of the argument, create a temporary variable of the correct type, assign the argument to the variable, and supply a pointer to the variable's address to `clSetKernelArgs`. This is the approach taken by the current CU2CL prototype.

4.1.7. Kernel function pointers

Another subtle difference between CUDA and OpenCL are the ways in which device kernels are invoked. The OpenCL `cl_kernel` data type is actually a pointer to an opaque type. Hence all kernel invocations are upon kernel function pointers

```
kernelName<<<grid,block>>>(kernelArgs...);
```

(a) Invoking a CUDA function

```
kernelPtr = &kernelName;
...
(*kernelPtr)<<<grid,block>>>(kernelArgs...);
```

(b) Invoking a dereferenced CUDA function pointer

```
kernelPtr = &kernelName;
...
kernelPtr<<<grid,block>>>(kernelArgs...);
```

(c) Invoking a CUDA function pointer directly

```
clKernel = clCreateKernel(program,
  kernelName, &errror);
...
status = clEnqueueNDRangeKernel(commandQueue,
  <clFuncPtr>, workDim, globalWorkOffset,
  globalWorkSize, localWorkSize,
  numEventsInWaitList, eventWaitList, &event);
```

(d) Equivalent OpenCL invocation

Fig. 5. Ways of invoking kernel functions in host code.

via `clEnqueueNDRangeKernel`. CUDA kernel functions, on the other hand can be invoked using CUDA C either directly by referencing the kernel function name or indirectly by a kernel function pointer either implicitly or explicitly. These alternatives are shown in Fig. 5.

The main difficulty with translating CUDA kernel function pointers is that the required rewrites often have non-local scope. If the location which CUDA kernel function pointer is initialized and the location where the pointer is invoked (e.g., Fig. 5(b) and (c)) are in the same function, only the equivalent OpenCL code of Fig. 5(d) need be generated.[3] However, if the kernel function pointer is passed through a set of host functions (signified by the ellipsis), all the host functions from pointer creation to invocation will need to be rewritten to propagate the `cl_kernel` type. in the place of the CUDA kernel pointer. In short, propagating types across function boundaries requires substantially more effort, whether the types are `cl_kernel` or `cl_mem` from Section 4.1.1.

4.1.8. Device initialization

The CUDA runtime API abstracts away many of the details needed to initialize the GPU and establish an execution context. There is no need for explicit initialization (as there is for the driver API or OpenCL). With the runtime API, an execution context is created for each device in the system and a default device assigned. The `cudaSetDevice` function can be used to change the context if needed, although this isn't necessary for many applications. In contrast, OpenCL requires the programmer to not only explicitly select both a compute platform and device, but also manually initialize a compute context and at least one command queue for synchronization. Therefore, a translator from CUDA to OpenCL needs to emulate the implicit initialization behavior of the CUDA runtime API. It also needs to translate explicit context setup as some applications make use of that functionality.

The translator can readily emulate initialization of a default device. However, additional work needs to occur when an application makes use of the cudaSetDevice function. First, as cudaSetDevice takes an integer argument specifying which device context to use, the translator must provide a mechanism for using an integer index to select among all compute devices in a system. Second, the translator must either supplant its own automatic initialization code, or it must intelligently preserve a portion of the OpenCL environment it has automatically initialized - replacing the command queue and context associated with the default device. Finally, it must (re) initialize an OpenCL context based on the device selected by the integer index. However, this method is not necessarily guaranteed to result in use of the same CUDA-capable GPU as intended by the original CUDA source. Particularly in systems in which multiple OpenCL platforms are present, depending on the structure of the device iteration code, it is quite possible that the default device might not even be a NVIDIA GPU.

4.1.9. Textures and surfaces

GPU devices are commonly equipped with special-purpose functional units designed to provide optimized access to data having particular characteristics. One such commonly used hardware unit provides fast reading of textures. Making use of

[3] For semantically-identical translation, `<clFuncPtr>` in Fig. 5(d) must be replaced by `clKernel` for Fig. 5(a) and by `kernelPtr` for Fig. 5(b) and (c).

these special-purpose memory regions can provide distinct performance benefits to some applications. CUDA provides explicit support for accessing these regions via the texture and surface types. Similarly, OpenCL provides the notion of an image type for providing access to these regions. However, despite their similar intentions, the APIs have a number of pronounced differences. Therefore, translation of CUDA textures and surfaces to OpenCL images requires careful consideration.

4.1.10. Graphics interoperability

Due to their heavily GPU-oriented backgrounds, both CUDA and OpenCL support mechanisms for allowing compute code to directly interact with graphics rendering code. Primarily, this is of use to applications which would like to provide in situ visualization of data computed on a device without the extra overhead of transferring data back to the host-side before rendering. A number of applications in the sample population make use of CUDA's OpenGL interoperability functions. While their translation appears achievable, CU2CL does not handle graphic rendering code at this time.

4.1.11. CUDA driver API

As mentioned earlier, CUDA has both a high-level runtime API and a lower-level driver API which provides explicit control over device usage with a corresponding increase in required detail (Fig. 3). The driver API is particularly close to OpenCL's API and hence it should be straightforward to support in CU2CL. As its use is rather low, this is not a high priority yet.

4.1.12. Structure alignment

Both CUDA and OpenCL provide a mechanism for explicitly aligning memory structures for passing between host and device. Without alignment directives, there is a potential conflict between host and device behavior. CUDA uses the `__align__(N)` attribute while OpenCL uses `__attribute__((aligned (N)))`.

The CUDA attribute is defined as a preprocessor macro which — for GNU C compilers — maps to an attribute specifier identical to that of OpenCL. This can create a potential confusion for source-to-source translators which operate on the source after preprocessing as there is effectively no difference in the specification. However, there are two readily-available approaches. First, the translator might simply copy the macro from the CUDA headers and perform no translation of the CUDA alignment specifier. However, this could be viewed as a slight departure from providing a canonical OpenCL version. Otherwise, the translator would need to preserve either the raw byte alignment of the attribute, or the entire 'N' expression of the attribute, and perform a simple substitution of the wrapping attribute syntax.

4.1.13. Warp-level synchronization

Within NVIDIA GPUs, threads are dispatched in groups of 32 sequentially-indexed threads, known as warps, which operate in lock step. Therefore, one can often take advantage of this implicitly synchronized execution to obviate the need for more costly synchronization methods such as fences, barriers, and atomics. However, as OpenCL supports a myriad of underlying compute devices, many of which do not exibit similar dispatch behavior, the preservation of warp-level synchronization is not guaranteed. On contemporary AMD GPUs and CPUs, threads are dispatched in groups of 64 and either 1 or the SIMD width, respectively. Thus, implicit synchronization may be somewhat preserved on AMD GPUs but will not on CPUs.

There is no CUDA syntax or function which provides explicit warp synchronization. Rather the programmer must manually orchestrate thread behavior based on individual indicies. This makes it very difficult for the translator to automatically recognize implicit synchronization and to provide a functionally equivalent translation. Fortunately there have been efforts to address this concern via dependency analysis in the context of CUDA to OpenMP translation [19] that can be leveraged for CU2CL.

5. Implementation

With an understanding of the challenges involved in translating CUDA to OpenCL, it is now time to discuss the translator implementation. While a full discussion of the implementation of our translator prototype, known as CU2CL (<u>CU</u>DA <u>to</u> Open<u>CL</u>) is outside the scope of this work, a brief overview of core facets of its construction is in order. A more thorough exposition is provided in [13].

A number of production-quality and widely-used open-source compilers, along with various research frameworks [20–23], were investigated in order to quickly develop a production-ready tool. Clang [24] was chosen as the basis for CU2CL for three reasons. First, though relatively young, Clang has a large and active community with many new features and rapidly improving quality. Second, Clang is a driver built upon the LLVM compiler libraries, which provide lexing, parsing, semantic analysis, and more. These libraries may be used independently to create other source-level tools. And third, Clang has support for parsing CUDA C extensions. Of all the tools investigated, only Clang and Cetus had that necessary capability. Of those two, Clang appeared to be the most production ready.

5.1. Architecture

As implicitly noted in the Fig. 6, CU2CL is a Clang plugin that ties into the main driver, allowing Clang to handle parsing and abstract syntax tree (AST) generation, as during normal compilation, after which CU2CL walks the generated AST to

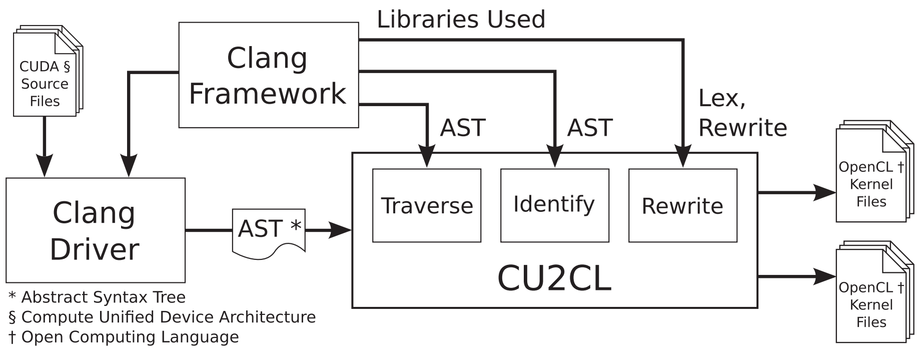


Fig. 6. High-level overview of the CU2CL translation process.

perform the rewrites. Of particular interest in this context are the AST, Basic, Frontend, Lex, Parse, and Rewrite libraries from Clang. 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, a robust CUDA-to-OpenCL translator was created in less than 3400 source lines of code (SLOC).*

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, a CU2CL-specific method is used to traverse the AST in a recursive descent fashion, using AST node iterators to recurse into each node's children.

5.2. AST-driven string-based rewriting

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 locally; therefore, this approach is called *AST-Driven String-Based Rewriting.*

This approach is quite useful in translating CUDA to OpenCL as only the CUDA-related constructs need be modified. The remainder of the source code passes through untouched. Unlike the traditional approach of generating the output directly from the AST, AST-driven string-based rewriting preserves almost all of the comments and formatting that is so important for maintainability [25]. In general, the scope of the translations are very small. As a document's structure and comments are of vital importance to developers [26], leaving them intact is an important benefit to CU2CL as the translated source can now serve as the basis for further development.

5.3. Translating 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 with a few exceptions. CUDA 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.

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 may be found in host or device code and as kernel arguments. OpenCL vector types have slightly different names depending on where they are found — i.e., `cl_float4` in host code vs. `float4` in device code — 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.

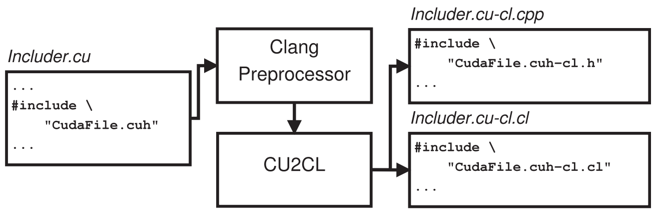


Fig. 7. Example of rewriting an `#include` directive.

CUDA function calls to be rewritten can be processed separately. However, for the purposes of source-to-source translation, it is preferable to generalize as much of the rewriting as possible. An important pattern is a pointer to a data structure that is passed in to be filled. The equivalent OpenCL functions instead return a pointer to an opaque structure, as shown in Fig. 4. To translate from CUDA to OpenCL, a 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. The subexpression with this evaluated type is 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, comparison to the equivalent OpenCL procedure and pertinent error codes will help in properly rewriting parents that use the returned error.

5.4. Rewriting includes

As part of translation, `#include` preprocessor directives for CUDA header files must be removed or rewritten. Because `#include`s are not resident in the AST, the rewriting has been implemented using the Clang driver preprocessor, as shown in the block diagram of Fig. 7. CU2CL registers a callback with the preprocessor that is invoked whenever a new `#include` is 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. 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 `#include`s pointing to CUDA and system header files that are no longer needed and (2) rewriting `#include`s 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. These header files are identified as those included using the angle bracket notation as opposed to double quotes.[4] In the second case, CU2CL rewrites `#include`s 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.cuh* will become *cudaFile.cuh-cl.h* and *cudaFile.cuh-cl.cl*). Therefore, CU2CL rewrites the original `#include`s so that they point to the new OpenCL files. Fig. 7 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 `#include`d by the host.

5.5. Error reporting

Compilers and translators have another difficult task besides generating correct output. They also need to provide good error reports when the inputs are incorrect or ambiguous to guide programmers in fixing the errors. In this regard, starting with Clang is definitely a benefit as many consider it to have some of the most accurate and useful error reporting of any compiler. Source-to-source translation needs to not only report errors, but to also clearly mark where the translator was unable to successfully translate a construct so that they can be handled manually.

CU2CL provides a unified mechanism for reporting issues that arise during the translation in two ways. First, similar to a standard compiler, all notifications are emitted with available source file, line, and column information to the terminal error output stream, with a severity level and brief textual description of the issue. However, to aid manual intervention in corner cases which are not automatically translated, notifications are also emitted into the translated application source as comments adjacent to the code of interest. These comments include both an easily searchable severity level as well as a textual description which can be either identical to the message emitted to the terminal error stream, or specified separately.

[4] A comparison with system header files could be done instead in order to be more forgiving of programmer carelessness in using angle brackets instead of double quotes but has not been done yet.

Primarily we provide four levels of notifications: translation errors, untranslatable syntax, currently untranslated syntax, and "advisories." Translation errors are emitted to notify that CU2CL has encountered a source construct which it has no mechanism for actively handling. In general emission of this level of notification is only used as a catchall default in branching logic when the translator cannot make a concrete decision on how to proceed. These indicate an area where the translator is incomplete.

More frequently, the translator encounters syntax which it can recognize, but due to differences in CUDA and OpenCL, cannot automatically translate. In these cases, a "CU2CL Untranslated" notification is emitted. Similarly, the translator will also emit a notification when it reaches an actively recognized syntax element which has an OpenCL equivalent but it does not yet support translating.

Finally, it can emit notes, which serve as advisories that the translator has had to utilize some special purpose code to handle a structure, such as adding a variable storing the result of a temporary expression for use as a kernel argument. These cases do not require manual intervention but are emitted as a courtesy to support using the translated code as a basis for new development.

6. Evaluation

In this section, CU2CL is evaluated using three metrics: the speed of translation, the performance of translated applications, and the amount of the CUDA runtime API covered. The frequency of the translation challenges discussed in Section 4 is also presented.

6.1. Translation speed

While CU2CL may 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 metric of interest, particularly if the end user wishes to convert multiple CUDA applications from the well-established CUDA ecosystem. Or perhaps there is a desire to continue development in CUDA but to translate to OpenCL in order to gain access to a greater breadth of accelerator platforms. Thus it is important to evaluate the speed of translation on several GPU applications. The applications chosen are from the CUDA SDK and the Rodinia benchmark suite. The full list of applications can be found in the appendix; a subset are shown in Table 5.

For each application, the total time to translate the code from CUDA to OpenCL was averaged 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. The overall run time was measured using the `time` command while the portion of the run time attributed to CU2CL was measured with `gettimeofday`. In all cases, the width of the 95% confidence interval centered around the mean is less than $\pm 5\%$.

Tables 5 summarizes the results. The test applications vary in length from more than a hundred source lines of code (SLOC) to several 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 constructs tend to take longer to translate. In most cases, CU2CL translates the applications in well under a second. The longest translation time is for the `particles` application from the CUDA SDK which contains five CUDA files containing 1,184 SLOC and still only takes a little more than two seconds. Thus, CU2CL is a feasible choice for porting a large number of CUDA programs.

Table 5
Translation time and coverage of CU2CL translation.

Source	Application	CUDA lines	Total translation time (s)	CU2CL time (μs)	Manual OpenCL lines changed	Percent automatically translated
CUDA SDK	asyncAPI	135	0.14	163	5	96.3
	bandwidthTest	891	0.28	289	5	98.9
	BlackScholes	347	0.27	200	14	96.0
	fastWalshTransform	327	0.15	208	30	90.8
	matrixMul	351	0.14	211	9	97.4
	scalarProd	251	0.16	226	18	92.8
	vectorAdd	147	0.14	97	0	100.0
Rodinia	Back Propagation	313	0.14	174	24	92.3
	Breadth-First Search	306	0.14	200	35	88.6
	Gaussian	390	0.14	210	26	93.3
	Hotspot	328	0.14	204	2	99.4
	Needleman–Wunsch	430	0.14	191	3	99.3
Ref. [27]	Fen Zi	17768	0.35	3491	1786	89.9
Ref. [28]	GEM	524	0.14	182	15	97.1
Ref. [29]	IZ PS	8402	0.21	1091	166	98.0

6.2. Translated application performance: auto vs. manual

In this section, the performance of translated applications is evaluated using the execution time as a metric. The performance of thirteen automatically translated CUDA-to-OpenCL codes is considered: seven from the CUDA SDK, five from the Rodinia benchmark suite, and the GEM molecular modeling application [30].

For all of the experiments, the applications are complied and run on a desktop machine with an AMD Phenom II X6 1090T Processor (six-cores, 3.2 GHz) with 16-GB RAM running 64-bit Ubuntu 12.04 with Linux kernel `3.2.0–35`. The GPU is a NVIDIA GeForce GTX 480 with 1.5-GB RAM (480 total cores) using the NVIDIA driver version 310.32 and CUDA Runtime 5.0. Run times were measured using the `time` command.

Table 6 summarizes the performance comparisons between the original CUDA code and CU2CL's automatically-generated OpenCL. Each code was executed a total of ten times and their runtimes were averaged.

In all but three applications, the automatically-translated OpenCL performs better than the original CUDA, though not by much. This contrasts starkly with previous work which demonstrated that automatically-translated OpenCL often gave significant slowdown when using the CUDA 3.2 Runtime [13]. As the OpenCL ecosystem has matured and NVIDIAs OpenCL implementation has improved, the performance difference has been largely eradicated. Therefore with modern hardware platforms and software stacks, no significant performance penalty is incurred when automatically-translated applications are executed on the same NVIDIA device, but with the added gain of access to other OpenCL-supporting devices.

6.3. Translator coverage

CU2CL supports a large majority of the subset of the CUDA runtime API that existed with version 3.2 of the CUDA SDK. Given that the 4.X and 5.0 versions of the SDK add several new features which are largely specific to NVIDIA GPUs, work is still ongoing to determine for which features equivalent OpenCL functionality exists, and mechanisms for handling the remainder. 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. This is a natural result of selecting the most frequently used calls from the CUDA SDK and Rodinia benchmark for implementation first.

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 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 the manual changes, none are particularly difficult to handle and automated support for these will be added as CU2CL continues to evolve.

6.4. Frequency of translation challenges

Section 4 discusses a number of translation challenges. Table 7 lists the challenges and their frequency of occurrence in the CUDA SDK and the Rodinia benchmark suite that are used in the evaluation above. The first lists the challenge, the second and third columns give the percentage of the applications in the respective suites that exhibited the challenge.

The first observation is that the CUDA SDK exhibits instances of all the challenges. This is to be expected since example applications in the SDK are intended to instruct programmers in how to use CUDA. The second observation is that Rodinia did not exhibit some of the challenges. Some, like graphics interoperability, are not surprising considering the Rodinia benchmarks are intended to test compute performance not graphics. The other omissions are largely due to the more homogeneous programming style employed.

Table 6
Run times of CUDA applications and OpenCL ports on an NVIDIA GTX 480.

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

Table 7
CUDA-to-OpenCL translation challenges and frequency of affected applications.

Challenge	CUDA SDK frequency (%)	Rodinia frequency (%)
Separate compilation	54.4	29.4
CUDA libraries	10.1	0.0
Kernel templates	21.5	0.0
`cudaSetDevice`	54.4	29.4
Textures	27.8	23.5
Graphics interoperability	24.1	11.8
CUDA driver API	8.9	5.9
Literal arguments	19.0	17.6
Aligned types	6.3	5.9
Constant memory	17.7	29.4
Shared memory	46.8	70.6

Perhaps the most useful result from the table is the order in which to tackle the challenges to maximize benefit while minimizing effort. According to the data, device initialization is first followed by separate compilation, textures, and literal arguments. As CU2CL is undergoing continued development, partial support has already been added for cudaSetDevice, literal kernel arguments, constant memory, and shared memory. Additionally, code to actively identify instances of kernel templates has already been integrated with CU2CL's error reporting mechanism discussed in Section 5.5.

7. Future work

While much has been done in to create a usable CUDA to OpenCL translator, there are additional items that need to be completed. The most important is to finish handling all types of device initialization challenges, implement separate compilation, finish implementing complete type propagation, finish the remaining cases of structure alignment, and to increase support for later versions of the CUDA API. Note that support for CUDA code containing C++ will not be feasible until the OpenCL standard supports C++.

Longer term, CU2CL needs to be extended to support the CUDA driver API, as well as extend and leverage Ocelet to perform conversion of binary PTX code into OpenCL or specific IRs. Finally, the most interesting challenge to address will be performance portability between GPUs from different manufactures and radically different devices once functional portability, the topic of this paper, is completed.

8. Conclusion

The CUDA programming environment for heterogeneous processors, namely GPUs in this case, debuted approximately two years before the arrival of the open-standard OpenCL. In light of the significant time and effort invested in creating GPU-accelerated codes in CUDA, there exists a treasure trove of CUDA applications that end users desire to migrate to an open-standard programming platform in order to preserve their intellectual investment while gaining greater breadth in the number and types of parallel computing devices that are supported. Such parallel computing devices include AMD and Intel x86 CPUs, ARM CPUs, AMD APUs (i.e., accelerated processing units, where the CPU and GPU cores are "fused" onto the same processor die), AMD and NVIDIA GPUs, and even FPGAs, to name a handful. To address the above, we created an automated CUDA-to-OpenCL source-to-source translator that enables CUDA programs to be automatically translated and run on any parallel computing device that supports an OpenCL ecosystem [13].

The work presented here seeks to characterize the challenges faced in creating a robust CUDA-to-OpenCL translator, present our instantiation of a CUDA-to-OpenCL (CU2CL) source-to-source translator, and evaluate its efficacy on real CUDA codes. We have shown that although it is not straightforward and (currently) subject to some important limitations, robust automatic source translation from CUDA to OpenCL is largely achievable. Further we have shown that once translated, when executed on the same device, application performance is retained, suggesting that the improved portability of OpenCL codes no longer results in reduced performance on CUDA devices. Finally, we presented a robust automatic translator capable of reducing the man-weeks required for manual translations to the order of seconds.

Appendix A

Tables 8 and 9 provide the performance results of running CU2CL on the Rodinia benchmark applications and on the examples from the CUDA SDK, respectively. Each table provides the name of the application, the number of CUDA source lines in the application, and the total translation time and the portion taken by CU2CL to perform the translation.

Table 8
CUDA SDK translation time.

Application	CUDA lines	Total translation time (s)	CU2CL time (μs)
alignedTypes	316	0.16	239
asyncAPI	135	0.14	163
bandwidthTest	891	0.28	289
bicubicTexture	1251	0.78	482
bilateralFilter	864	0.89	415
binomialOptions	443	0.64	328
BlackScholes	347	0.27	200
boxFilter	980	0.74	339
clock	162	0.15	149
concurrentKernels	177	0.27	177
conjugateGradient	196	0.06	170
convolutionFFT2D	1175	0.65	488
convolutionSeparable	363	0.75	288
convolutionTexture	368	0.63	295
cppIntegration	247	0.73	261
dct8x8	1715	0.29	539
deviceQuery	165	0.57	160
deviceQueryDrv	150	0.58	150
dwtHaar1D	598	0.16	281
dxtc	886	0.43	472
eigenvalues	3109	0.48	1116
fastWalshTransform	327	0.15	208
FDTD3d	870	0.99	405
fluidsGL	811	0.28	330
FunctionPointers	1004	0.76	449
histogram	545	0.90	436
imageDenoising	1305	0.75	512
lineOfSight	337	0.17	228
Mandelbrot	2528	0.93	922
marchingCubes	1571	0.80	540
matrixMul	351	0.14	211
matrixMulDrv	525	0.72	378
matrixMulDynlinkJIT	301	0.46	158
mergeSort	954	0.65	412
MersenneTwister	310	0.27	193
MonteCarlo	1014	0.79	726
MonteCarlo Multi GPU	994	0.79	743
nbody	2088	1.54	824
oceanFFT	1037	0.76	452
particles	1184	2.41	1001
postProcessGL	1291	0.88	489
ptxjit	132	0.58	120
quasirandomGenerator	510	0.90	504
radixSort	2387	1.37	1103
randomFog	888	1.34	345
recursiveGaussian	883	0.77	417
reduction	1063	0.78	583
scalarProd	251	0.16	226
scan	495	0.75	322
simpleAtomicIntrinsics	197	0.15	155
simpleCUBLAS	244	0.10	149
simpleCUFFT	249	0.15	173
simpleGL	603	0.73	350
simpleMPI	208	0.84	274
simpleMultiCopy	351	0.27	254
simpleMultiGPU	226	0.47	202
simplePitchLinearTexture	274	0.15	180
simplePrintf	1066	0.43	893
simpleStreams	243	0.15	193
simpleSurfaceWrite	207	0.15	201
simpleTemplates	458	0.16	248
simpleTexture	239	0.15	186
simpleTexture3D	506	0.78	305
simpleTextureDrv	392	0.72	379
simpleVoteIntrinsics	341	0.15	218
simpleZeroCopy	149	0.15	147
smokeParticles	2016	1.21	531
SobelFilter	780	0.75	360

Table 8 (*continued*)

Application	CUDA lines	Total translation time (s)	CU2CL time (μs)
SobolQRNG	10698	1.73	5275
sortingNetworks	657	0.90	487
template	187	0.15	158
threadFenceReduction	791	0.17	483
threadMigration	434	0.72	393
transpose	571	0.27	271
vectorAdd	147	0.14	97
vectorAddDrv	351	0.60	281
volumeRender	884	0.78	393

Table 9
Rodinia translation time.

Application	CUDA lines	Total translation time (s)	CU2CL time (μs)
Back propagation	313	0.14	174
Breadth-first search	306	0.14	200
CFD	2371	1.07	1230
Gaussian	390	0.14	210
Heartwall	2018	0.17	532
Hotspot	328	0.14	204
Kmeans	494	0.14	241
LavaMD	240	0.14	192
Leukocyte	624	0.28	386
LU decomposition	332	0.28	277
MummerGPU	3786	0.18	655
Nearest neighbor	278	0.17	170
Needleman–Wunsch	430	0.14	191
Particle filter	1517	0.31	582
Path finder	235	0.14	186
SRADv1	541	0.15	366
Stream cluster	443	0.26	211

References

[1] M. Daga, T. Scogland, W. Feng, Architecture-Aware Mapping and Optimization on a 1600-Core GPU, in 17th IEEE International Conference on Parallel and Distributed Systems, Tainan, Taiwan, December, 2011.

[2] S. Xiao, H. Lin, W.-C. Feng, Accelerating protein sequence search in a heterogeneous computing system, in Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International, 2011, pp. 1212–1222.

[3] NVIDA Corporation, Nvidia CUDA C Programming Guide, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDAProgrammingGuide.pdf.

[4] NVIDIA Corporation, NVIDIA Contributes CUDA Compiler to Open Source Community, http://nvidianews.nvidia.com/Releases/NVIDIA-Contributes-CUDA-Compiler- to-Open-Source-Community-7d0.aspx, May 9, 2012.

[5] J. Leskela, J. Nikula, M. Salmela, OpenCL Embedded Profile Prototype in Mobile Device, in IEEE Workshop on Signal Processing Systems, Oct 2009, pp. 279–284.

[6] HPCwire, The Portland Group Ships OpenCL Compiler for Multi-core ARM, press release at http://www.hpcwire.com/hpcwire/2012-02-28/theportlandgroupshipsopen clcompilerformulti-corearm.html, Feb 28, 2012.

[7] Imagination Technologies, Imagination Submits POWERVR SGX Cores for OpenCL Conformance, press release http://www.imgtec.com/news/Release/index.asp?NewsID=610, Feb 14, 2011.

[8] Y. Aridor, Discussing Intel's OpenCL With Technical Lead Yariv Aridor - Parallel Programming Talk #117, video at http://software.intel.com/en-us/blogs/2011/07/27/discussing-intels-open cl-with-technical-lead-yariv-aridor-parallel-programming-talk-117/, July 27, 2011.

[9] Altera, White Paper: Implementing FPGA Design with the OpenCL Standard, http://www.altera.com/b/opencl.html, Nov 2011.

[10] G.F. Diamos, A.R. Kerr, S. Yalamanchili, N. Clark, Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems, in: 19th International Conference on Parallel Architectures and Compilation, Techniques, 2010, pp. 353–364.

[11] R. Domínguez, D. Schaa, D. Kaeli, Caracal: Dynamic Translation of Runtime Environments for GPUs, in 4th Workshop on General Purpose Processing on Graphics Processing Units, 2011, pp. 5:1–5:7.

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

[13] G. Martinez, M. Gardner, W. Feng, CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures, in IEEE 17th Intl, Conference on Parallel and Distributed Systems, Dec, 2011. pp. 300–307.

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

[15] S. Rosendahl, CUDA and OpenCL API Comparison, Presentation for T106.5800 Seminar on GPGPU Programming, Spring 2010, https://wiki.aalto.fi/download/attachments/40025977/Cuda+and+OpenCL+API +comparisonpresented.pdf.

[16] NVIDIA, CUDA Toolkit, http://developer.nvidia.com/cuda/cuda-toolkit.

[17] Rodinia: A Benchmark Suite for Heterogeneous Computing, http://lava.cs.virginia.edu/Rodinia.

[18] D. Spinellis, Global analysis and transformations in preprocessed languages, IEEE Transactions on Software Engineering 29 (11) (Nov 2003) 1019–1030.

[19] Z. Guo, E. Zhang, X. Shen, Correctly treating synchronizations in compiling fine-grained SPMD-threaded programs for CPU, in: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011, pp. 310–319.

[20] D. Quinlan, ROSE: Compiler Support for Object-Oriented Frameworks, Parallel Processing Letters 2 (3) (2000) 215–226.

[21] V. Eelco, Program Transformation with Stratego/XT, in Domain-Specific Program Generation, ser, Lecture Notes in Computer Science 3016 (2004) 315–349.
[22] I. Baxter, C. Pidgeon, M. Mehlich, DMS: program transformations for practical scalable software evolution, in: 26th International Conference on Software Engineering. IEEE Computer Society, 2004, pp. 625–634.
[23] S. Lee, T. Johnson, R. Eigenmann, Cetus–an extensible compiler infrastructure for source-to-source transformation, Languages and Compilers for Parallel Computing 9703180 (2004) 539–553.
[24] clang: a C language family frontend for LLVM, http://clang.llvm.org/.
[25] J. Van Wijngaarden, J. Van Wijngaarden, E. Visser, Program Transformation Mechanics: A classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems, Utrecht University: Information and Computing Sciences, Tech. Rep. UU-CS 2003-048, 2003.
[26] M.L. Van De Vanter, Preserving the Documentary Structure of Source Code in Language-Based Transformation Tools, Workshop on Source Code Analysis and Manipulation, 2001, pp. 131–141.
[27] B.A. Bauer, J.E. Davis, M. Taufer, S. Patel, Molecular Dynamics Simulations of Aqueous Ions at the LiquidVapor Interface Accelerated using Graphics Processors, Journal of Computational Chemistry 32 (3) (2011) 375–385.
[28] R. Anandakrishnan, T.R. Scogland, A.T. Fenley, J.C. Gordon, W. chun Feng, A.V. Onufriev, Accelerating electrostatic surface potential calculation with multi-scale approximation on graphics processing units, Journal of Molecular Graphics and Modelling 28 (8) (2010) 904–910.
[29] D. Yudanov, M. Shaaban, R. Melton, L. Reznik, GPU-based simulation of spiking neural networks with real-time performance and high accuracy, in: Neural Networks (IJCNN), The 2010 International Joint Conference on, July, pp. 1–8.
[30] R. Anandakrishnan, T.R. Scogland, A.T. Fenley, J.C. Gordon, W. chun Feng, A.V. Onufriev, ‘‘Accelerating Electrostatic Surface Potential Calculation with Multi-Scale Approximation on Graphics Processing Units’’, Journal of Molecular Graphics and Modelling 28 (8) (2010) 904–910.