Lost in Translation: Challenges in Automating CUDA-to-OpenCL Translation

Paul Sathre
Dept. of Computer Science
Virginia Tech, USA
Email: sath6220@cs.vt.edu

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

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

***Abstract*—The use of accelerators in high-performance computing is increasing. The most commonly used accelerator is the graphics processing unit (GPU) because of its low cost and massively parallel performance. The two most common programming environments for GPU accelerators are CUDA and OpenCL. While CUDA runs natively only on NVIDIA GPUs, OpenCL is an open standard that can run on a variety of hardware processing platforms, including NVIDIA GPUs, AMD GPUs, and Intel or AMD CPUs.**

Given the abundance of GPU applications written in CUDA, we seek to leverage this investment in CUDA and enable CUDA programs to "run anywhere" via a CUDA-to-OpenCL source-to-source translator. The resultant OpenCL versions permit the GPU-accelerated codes to run on a wider variety of processors that would not otherwise be possible. However, *robust* source-to-source translation from CUDA to OpenCL faces a myriad of challenges. As such, this paper identifies those challenges and presents a classification of CUDA language idioms that present practical impediments to automatic translation.

I. Introduction

The use of diverse multi- and many-core devices for accelerated performance has grown to become a key technique in developing high-performance computing (HPC) applications. GPUs have been at the forefront of this recent growth, in large part due to the success of NVIDIA's CUDA environment, and more recently, OpenCL, both of which aim to ease the general-purpose programming of GPUs. However, while the APIs are similar in many respects, the CUDA environment is available only to NVIDIA GPU devices, whereas the OpenCL standard may be implemented to run on any vendor's hardware, such as traditional CPUs, and GPUs. OpenCL's code portability provides both developers and users with a standard interface for extracting parallel performance from diverse hardware resources.

Due to the large number of early adopters of CUDA, as well as its approximately 18-month head start over OpenCL, a considerable body of CUDA code has already amassed, representing many person-hours of development. However, despite having similar APIs and functionality, there is no simple one-to-one mapping for all operations. Furthermore, each provides some degree of exclusive functionality. Thus, porting applications between the two languages remains a non-trivial endeavor, requiring considerable programmer time and effort and being prone to introducing translation errors.

This work was supported in part by an AMD Research Faculty Fellowship and the NSF Center For High-Performance Reconfigurable Computing supported through NSF I/UCRC Grant IIP-0804155.

This paper presents insights needed for successful CUDA to OpenCL translation, gained from experiences profiling a diverse population of typical CUDA applications. Specifically, we present a taxonomy of several generalized CUDA language idioms that are currently barriers to automation. We further classify these idioms based on their relative translatability to drive reasoning about potential methods for their implementation in OpenCL.

After providing background information, including related work as well as a high-level overview of both the CUDA and OpenCL APIs, we briefly discuss the prototype translator used to help identify language constructs of interest in this paper. Next, we focus on a discussion and evaluation of the core contributions of the work, namely, a characterization of the challenges for fully-automated translation, which includes

- A generalized classification of significant untranslated language idioms.
- Generalized solution mechanisms for mistranslations observed in the prototype translator.
- Conceptual foundations for solutions to more significant translation impediments.

Finally, we conclude by presenting opportunities for future development of automatic CUDA-to-OpenCL translators and drawing general conclusions on the applicability of our work.

II. Background

As previously noted, CUDA and OpenCL both provide frameworks for performing general-purpose GPU (GPGPU) computations with a number of remarkable similarities. First, both provide the abstraction of an accelerator device — NVIDIA GPUs for CUDA and *any* device with a supporting implementation for OpenCL — that is used for the execution of parallel kernel code. Further, both share similar threading and memory models and a similar kernel language. However, due to their differing emphases — proprietary GPUs vs. multi-vendor compatibility on general parallel computing devices — enough differences exist between CUDA and OpenCL that pose significant challenges towards realizing a *robust* CUDA-to-OpenCL source-to-source translator.

A. Related Work

There exist a number of related efforts to automatically translate software either to or from the CUDA framework. The parallel performance afforded by GPUs has created a demand for programming models and tools which simplify the development of GPU-accelerated software.

A popular approach to providing software portability is the use of a lower-level intermediate representation (IR) between high-level source and architecture-specific binary. Ocelot is a compilation framework that translates CUDA's PTX IR to x86 CPU code via the LLVM framework [1]. Ocelot can dynamically execute CUDA applications on both GPUs and CPUs without recompiling [2]. However, this approach requires that a separate backend be developed for each target IR, such as Caracal, a PTX-to-CAL runtime environment for execution of CUDA code on AMD GPUs [3].

Portability can also be provided through the insertion of an additional abstraction layer between the developer and the GPU programming framework. This is the approach taken by Swan, a tool for switching between CUDA and OpenCL. Applications written to use the Swan API in place of the CUDA API are compiled for either CUDA or OpenCL simply by changing the tool's build target and compiling against the respective Swan library [4]. However, this approach still requires manual translation from CUDA code to Swan's API.

The shared memory programming (SMP) model has seen widespread use in the development of parallel applications. OpenMP is a popular open standard for implementing the SMP model, and there are efforts underway to translate OpenMP to and from CUDA. One such effort, known as OpenMPC, has demonstrated an extension to OpenMP, which provides autonomous compilation and tuning of CUDA executables from OpenMP source code [5][6].

A separate effort, underway at the University of Illinois at Urbana-Champaign (UIUC), implements the CUDA programming model on multi-core CPUs. This framework, known as MCUDA, has demonstrated techniques for performing automatic transformation and tuning of an abstract syntax tree (AST) from CUDA to OpenMP [7].

Another project at UIUC, CUDAtoOpenCL [8], implements a modified preprocessor and AST IR, allowing parsing of CUDA source. Transformations on the AST for generating host and device code are performed in separate phases. A recursion-based mechanism for propagating translations to OpenCL data types has been demonstrated. However, some features are said to be unsupported. In addition, its source code has not been released, and as such, cannot be compared directly to our translator, CU2CL. We note that both CUDAtoOpenCL and CU2CL have recursion-based expression rewriting, propagation of the cl_mem type, and support for the most commonly-used environment setup, memory management, and kernel launch functions [8] [9].

B. CUDA

NVIDIA's CUDA has been at the forefront of GPGPU development since its initial release in 2006 and has recently reached its fourth major revision. At its heart, CUDA is designed to allow programmers easier access to the raw computational performance of NVIDIA's GPU architectures without the need to transform algorithms in order to use graphics APIs as was previously the case. While primarily a programming framework, CUDA provides a complete environment for the development of GPGPU applications via a combination of extensions to the C language (CUDA C), a corresponding compiler and visual profiler, and both a high-level runtime API and a lower-level driver API.

CUDA's primary compute functionality is achieved through host-side enqueuing of kernel functions, which are then executed on the GPU device asynchronously by the CUDA runtime [10]. This queue can be user configured to launch kernels and complete memory transfers either in strictly sequential order or an arbitrary order. An additional level of explicit synchronization is available through the use of user-defined events. The kernel launch model provides a hierarchical organization of device *threads*, consisting of a one- to three-dimensional *grid* of thread *blocks*, each of which represents a one- or two-dimensional subdomain of the grid space. At each kernel launch, both the grid and block dimensions must be declared in order to partition the workload to available streaming multiprocessors. Figure 1a provides a summary of this execution model.

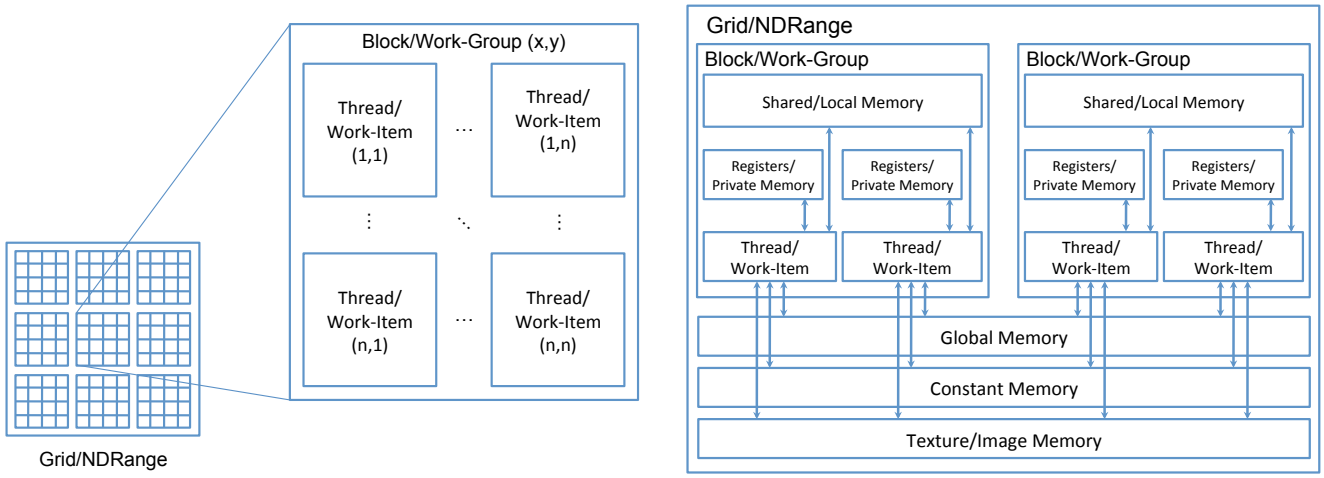
CUDA utilizes a hierarchical memory model distributed across both host and device. As of the CUDA 4.0 release, it also provides a uniform address space [10]. Device memory is partitioned into several distinct units, which conceptually, function much like the tiers of a cache albeit with relaxed consistency. (In fact, NVIDIA GPUs with compute capability 2.0 or higher provide an adjustable L1 cache.)

Global memory makes up the vast majority of GPU memory space; it is accessible from both the host and GPU and allows for data communication to and from the GPU. In addition to this bulk global memory, *texture* and *constant* regions are provided but are somewhat limited due to their small size, the read-only nature of constant memory, and the need for special-purpose functions to access texture data. *Shared memory* is on-chip and significantly faster, but it is far smaller and access is restricted to only threads that reside in the same block. Finally, *GPU registers*, similar to their CPU counterparts, are few in number and only accessible by the owning thread, but are the fastest by far. An overview of this memory model is provided in Figure 1b.

C. OpenCL

OpenCL was first devised by Apple in an attempt to avoid being restricted to any given hardware vendor's proprietary API. It has since been transferred to the Khronos Group to steward as an open standard for heterogeneous computing [11]. OpenCL provides a mechanism through which parallel applications can be developed for diverse accelerator hardware, without the need for time-intensive manual adaptation.

Conceptually, OpenCL provides a threading model similar to CUDA's that allows the software designer to abstract away



(a) Kernel Execution models

(b) Memory models

Fig. 1. CUDA and OpenCL Kernel Execution and Memory Models

all but the most critical considerations of the underlying architecture and express parallelism in terms of a large *ND Range*. This grouping of the device's thread hierarchy represents a direct one-to-one mapping to CUDA's grid. In addition, OpenCL uses the terms *work-group* and *work-item* in place of CUDA's blocks and threads, respectively.

OpenCL's memory hierarchy is another area of similarity. *Global* and *constant* memories are effectively unchanged. CUDA's shared memory is referred to as *local* memory in OpenCL and is specific to a work-group. CUDA registers are referred to as *private* memory in OpenCL which is specific to a given work-item. Figures 1a and 1b show these CUDA-to-OpenCL mappings.

III. Translator Prototype

Much of the results presented in this paper were enabled by our prototype source-to-source translator, known as CU2CL (i.e., CUDA-to-OpenCL). While a full discussion of CU2CL's implementation is outside the scope of this paper, we provide a brief introduction to its high-level construction as context. A more detailed exposition is available in [9].

CU2CL is built on the Clang compiler framework [12], a strategic choice based on its extensible, library-based construction, as well as its built-in CUDA parser. By building the translator within this framework, the vast majority of core functionality could be implemented by the clever composition of pre-existing functionality, requiring only that a new plugin be written to provide a new recursive descent AST walker as well as the actual CUDA-to-OpenCL rewriting process.

CU2CL is "AST-driven and string-based" [9] and relies on Clang for parsing and generation of an AST, which is then used by CU2CL to identify sections of interest for translation. However, contrary to many translation efforts, the resultant source code is *not* derived strictly from the translated AST. Instead, string insertions and replacements are performed within a local context. This ensures that meta-content embedded in the source, such as commenting and formatting, is nearly completely preserved in the resulting OpenCL output. This is a definite boon to the use, extension, and maintenance of automatically translated code. Further, [9] has provided an early analysis of the CU2CL prototype, demonstrating that automatically translated code offers performance on par with hand-translated versions when executed on the same underlying CUDA device.

IV. Approach

As a preliminary step towards the future development of CUDA-to-OpenCL translators, including the extension of our CU2CL prototype, we have undertaken a study of sample CUDA applications in search of software idioms that provide fundamental impediments to translation. As a byproduct, we also discovered a small number of CU2CL idiosyncrasies.

In order to encompass a significant majority of the full diversity of the CUDA language, we directed our attention towards 79 sample applications provided along with the CUDA SDK [13], as well as 17 from the Rodinia suite of parallel applications [14]. The SDK samples were chosen for their considerable breadth of coverage, whereas the Rodinia suite was chosen as representative of CUDA usage in practice. Rodinia also conveniently provides availability of manually translated applications for future comparison.

We began by subjecting each of the sample applications to a raw "first-pass" translation by CU2CL and examining the generated source artifacts from the original CUDA and the emitted errors for distinctive patterns. Upon analysis, we saw the emergence of distinct idioms that arose consistently across multiple applications. By profiling the population of sample applications for "markers" of these idioms – specific data types, function calls, and header files – we developed an understanding of the scope and breadth of these idioms as well as insight into solutions for realizing a more robust translator. Additionally, we manually removed code sections that were associated with translation failures in affected applications and then subjected these applications to additional passes through

the CU2CL prototype in order to attempt partial translation of the remaining source.

Out of this process emerged three general classes of language idioms: (1) fundamentally untranslatable idioms, (2) difficult-to-translate idioms, and (3) CU2CL-specific translation idioms. Additionally we note the largely language-agnostic challenge of providing cohesive translation across multiple source files which are compiled separately as disjoint objects underlying a larger application.

Table I presents an overarching summary of the frequency of these challenges within the sample populations. We discuss these in depth in Section V.

TABLE I
CUDA-TO-OPENCL TRANSLATION CHALLENGES AND FREQUENCY OF AFFECTED APPLICATIONS

Challenge	CUDA SDK Frequency (%)	Rodinia Frequency (%)
Separate Compilation	54.4	35.3
Untranslatable Idioms		
- CUDA Libraries	17.7	0
- PTX	7.6	0
- Kernel Templates	7.6	0
Difficult-to-Translate Idioms		
- Device Identifiers	77.2	0
- Static Constant Buffers	19.0	5.9
- Textures	32.9	23.5
- Graphics Interoperability	24.1	0
- CUDA Driver API	8.9	5.9
CU2CL-Specific Idioms		
- Kernel Literal Arguments	20.2	17.6
- Aligned Types	6.3	5.9
Miscellaneous Idioms		
- Surfaces	7.6	0
- K&R Style Declarations	0	5.9
- False System Includes	0	19.4

V. CHALLENGES

A. Separate Compilation

Separate compilation of CUDA applications from multiple source files is a challenge that is not necessarily specific to translation from CUDA to OpenCL. Rather, it is specific to source translation, which occurs at a file scope on languages or runtimes that have global state – such as the CUDA runtime API. Fundamentally, a source-to-source translator which operates on single source file at a time must propagate source rewrites to other software components which may not be integrated until the linking phase of compilation. This represents a significant challenge to any translator's ability to provide robust translation of real-world applications, as separate compilation plays a key role in the modular construction of full-scale applications.

This issue is highly relevant to CUDA-to-OpenCL translation as OpenCL requires significantly more initialization "boilerplate" code than the CUDA Runtime API. (In this regard, OpenCL is similar to the CUDA Driver API.) It requires a priori knowledge of the specific accelerator kernels that will be utilized by the application. However, we have observed that these commonly reside in source files separate from those containing main methods, and without explicit linking or manual intervention, it becomes a significant challenge for an automatic translator to ensure that the appropriate initialization has occurred. Doing so introduces the complexity of maintaining some state across translator executions.

As an example of this challenge, our prototype currently inserts initialization code into the main method for all CUDA kernels that share the same effective file scope (after preprocessing of includes), but only provides partial translation without initialization in other cases. Through clever usage of include directives, it is relatively easy to force full translation of many of these applications by giving CU2CL the appearance of operating on a single source, but this introduces an additional manual preprocessing step as well as obviating other potential benefits provided by modular, separate compilation. We find this to be an imperfect solution in the long term and are currently working to develop a robust mechanism for collecting necessary boilerplate across multiple, separately translated source files and deferring insertion of initialization and deconstruction shims into the main method when the containing code module is translated.

B. Fundamentally Untranslatable Idioms

Among the insights produced by our profiling were several CUDA syntax constructs and GPU programming practices that simply lie outside the practical scope of CUDA-to-OpenCL translation. We present three such idioms, which can be recognized, but not automatically translated at the present time for reasons stated below.

1) CUDA Libraries: The very nature of source-to-source translation restricts the ability to translate applications that make use of non-source components. In practice, this is primarily encountered in the use of third-party libraries that make internal use of the CUDA API to provide GPU-accelerated implementations of commonly used functions. In our analysis, we found instances of five such libraries: Boost, CUDPP, CUFFT, CUBLAS, and CURAND, which together appeared in 14 of the 79 samples taken from the CUDA SDK. It is relatively easy to identify applications that make use of these libraries by searching for their header files and function calls. However, without available CUDA source or equivalent OpenCL libraries, these calls will remain effectively outside the reach of a translator. As equivalent OpenCL libraries are developed, translation then becomes relatively straightforward. However, due to the presence of such libraries in real code, it must be gracefully handled by automatic translation efforts in the mean time, primarily by noting their presence via an error message.

2) PTX: Another less prevalent variant of the non-source component problem that we identified is the use of NVIDIA's PTX IR for distribution of highly-targeted device kernels within software projects. Due to OpenCL's abstractions around compute devices and the myriad of potential underlying implementations necessary to accommodate all supported OpenCL devices, translation would require deep analysis of the PTX source to construct equivalent OpenCL source; essentially a form of decompiling. This would require the use of an entirely

new front-end parser solely for constructing abstract meaning from highly device-tuned low-level operations. Further, the compilation of PTX requires usage of the CUDA Driver API, which is not currently supported by any CUDA-to-OpenCL translator.

3) Kernel Templates: Finally, the third idiom that lies largely outside the reach of CUDA-to-OpenCL translators is the ability to use C++ object-oriented constructs within CUDA device kernels. We recognize that this does not fundamentally present a class of untranslatable constructs as C++ objects are themselves implemented through simpler mechanisms, but such translation would effectively require their reimplementation at the present time. These represent a *current* language barrier solely due to OpenCL's continued reliance on purely C99 constructs within device code. Should future versions of the OpenCL standard allow for the inclusion of C++ object-oriented constructs such as templates within device code, then their automatic translation would be more readily achieved.

C. Difficult-to-Translate Idioms

This class of idioms represents those for which complete or near-complete functional equivalence can be realized in an automatic translator, but for which direct one-to-one mappings either do not exist or fail to ensure coverage of the full expressivity of the CUDA API. We demonstrate two cases of expressivity limitations – device IDs and statically allocated constant buffers – and three cases of functional equivalence requiring more elaborate syntax adaptations – textures, OpenGL interoperability, and the CUDA Driver API. The two cases of expressivity limitations demonstrate situations in which there exists a one-to-one mapping of CUDA-to-OpenCL behavior, but for which CUDA offers additional syntax and functionality variants that are less directly realized within OpenCL. The three remaining cases represent sections of the APIs that provide essentially equivalent functionality but with vastly differing syntax.

1) Device IDs: The single most common idiom that we observed within the sample of applications was the initialization of a CUDA device from a simple integer ID. The CUDA Runtime API provides a mechanism for selecting a specific CUDA-capable device from among the multiple devices which may be present in a system. However, the explicit call is rarely used due to the convenience of auto-initialization of a default device by the runtime when the first CUDA call is performed.

Currently, CU2CL assumes a similar auto-initialization behavior and simply chooses the default zeroth GPU device returned by the OpenCL runtime. However, for any translator to effectively emulate the optional functionality of choosing a device based on its integer index among all compute devices in a system, it must provide an OpenCL device initialization mechanism to create a structure which consists of all valid OpenCL devices and use the integer index to then make a distinction between them. Figure 2 in Section VII demonstrates OpenCL code that would effectively achieve this functionality.

2) Statically Allocated Constant Buffers: CUDA allows device-side constant memory buffers to be declared as static *constant* arrays using a variant on the C syntax for stack-allocated arrays. While OpenCL does *not* provide an equivalent method for the declaration of statically defined constant memory regions, translation of such a declaration to make use of the proper clCreateBuffer syntax could be easily added. However, the subtle difficulty presented by this idiom is OpenCL's need for constant buffers to be explicitly specified as kernel arguments. Proper detection and insertion of these additional arguments in only the relevant kernels will require a disciplined approach to identifying usage of such buffers within individual kernel functions.

3) Textures: The next idiom that we identified and that can potentially be implemented via functional equivalency is the mapping of CUDA textures to OpenCL images. Fundamentally, both provide access to special-purpose regions of a GPU device that allow for efficient access to data through image-centric addressing modes. However, CUDA and OpenCL present differing APIs for handling such data, requiring more significant changes than simple one-to-one mappings. Further, CUDA's one-dimensional texture type has no direct analog within the OpenCL specification before version 1.2 [11] and requires intelligent mapping of the one-dimensional texture onto a two-dimensional OpenCL image. This idiom was first noticed in two unpublished sample kernels that were provided by an industrial collaborator and reinforced within 26 of the CUDA SDK samples as well as four of the Rodinia applications. Work is still ongoing to create generalized mechanisms for performing texture-to-image mappings (via experimental manual translations of affected applications).

4) OpenGL Interoperability: Both CUDA and OpenCL provide mechanisms for directly interoperating with OpenGL objects that share data with compute kernels. These are primarily of interest for in-situ visualization of computations without the need for excess data movement to and from the CPU as an intermediate step between computation and rendering. (While CUDA textures may be considered to loosely fall under this umbrella, their usage in practice has been observed to frequently be completely agnostic to other graphics interopability, and as such, have been presented as a distinct idiom.)

5) CUDA Driver API: CUDA provides access to this low-level driver API that allows programmers more explicit control over the execution of software on CUDA devices. There are many parallels between CUDA's Driver API and OpenCL's specification, particularly the dramatically increased explicitness required to manage a compute device within the Driver API over the high-level CUDA Runtime API. Currently, no translator makes any attempt to translate constructs from the CUDA driver API. Based on the limited usage that we have observed (only seven CUDA samples and one Rodinia sample) as well as the drastically increased difficulty of programming with the CUDA Driver API, we place this at a low priority for our translator.

D. CU2CL-specific Idioms

The final set of idioms encompasses translated constructs that emit invalid OpenCL source under specific conditions due

to assumptions or oversights in the translator prototype, but require only relatively small additions to be supported. We stress that the implementation of solutions for these issues will improve the already high effective coverage of the translator and bring additional clarity to the search for as yet unidentified idioms by removal of excess noise.

1) Aligned Types: The only example of a simple one-to-one mapping that was not translated was alignment attributes. Both OpenCL and CUDA support explicit specification of data structure alignment in order to ensure a consistent memory layout on structures passed between host and device. However, our examination of CUDA source and OpenCL output revealed (and inspection of the prototype source later corroborated) that CUDA's *align(n)* construct was not being translated to OpenCL's *attribute ((aligned (n)))* construct. We encountered this construct in only five of the CUDA samples and only a single Rodinia application. However, we anecdotally note that the relevance of this issue is likely greater than indicated by our study due to the utility afforded by structurally organized memory.

2) Kernel Literal Arguments: Another potential translation pitfall that we have identified, which is typical of subtle assumptions made about incoming CUDA syntax, is the use of literal arguments to CUDA kernels. Within CUDA, arguments to device-side compute kernels are specified using pass-by-value semantics, as opposed to OpenCL's pass-by-reference mechanism of specifying kernel arguments. Currently, the prototype assumes every value passed to a compute kernel is stored in a specific variable (i.e., memory address) and simply translates variable names into appropriate pointer references. However, in the case of literal arguments – numerical constants, macro evaluations, and other inline mathematical operations – there is often no variable in the application source that ever contains the result of the expression. Further, the string replacement method of performing pointer translation results in mangled forms of the expression in the resultant OpenCL source. While this fundamentally represents a functional equivalency similar to those in Subsection V-C, an appropriate emulation can be rather directly achieved.

E. Miscellaneous and Unclassified Constructs

In addition to the well-defined idioms mentioned in the previous three subsections, we have identified additional constructs that present barriers to translation, but that are still being studied for generalizable patterns, as well as potential translatability before classification. These remaining constructs are all currently undergoing preliminary analysis to determine their membership into extant or new classifications.

The final subset of language constructs that we have identified, but not yet classified, consists of those which were exposed only in a small subset of the population or during late-stage passes by the prototype after earlier-identified untranslatable constructs had been manually removed from source samples. We are currently in the process of examining CUDA source, automatically translated OpenCL, and CU2CL's source itself to develop a similar understanding of these constructs.

First, the CUDA SDK samples provided several applications that make use of CUDA's *surface* data type, which is closely related to the *texture* type. However, before the release of the OpenCL 1.2 standard, there existed no relatively analogous type. It is not yet clear if OpenCL's recently introduced surface types offer full functional equivalence to CUDA's.

Second, a number of the samples taken from the Rodinia benchmarks suite use system-style preprocessor syntax for includes rather than user-space include syntax to reference relevant project header files. While this does not inhibit the prototype from translating the included files, the default behavior for rewriting the system-style include statement is to assume no rewrite occurred within the header and maintain the untranslated reference.

Finally, the Rodinia suite contains a single application, Backprop, which contains function declarations that enumerate arguments in K&R style that cause translation failure. We are still in the process of determining if this is a limitation of Clang's CUDA parser or the CU2CL translator library.

VI. Contribution

The GPGPU community has expressed great interest in translation from CUDA to OpenCL. While CUDA provides access to high raw computation performance, it is limited in its general applicability due its exclusivity to NVIDIA hardware, whereas OpenCL provides access to high performance computation on NVIDIA devices as well as those from other vendors. Primarily, the boon of OpenCL driving the desire for translation is functional portability. By converting existing codebases to OpenCL, the cost of developing software that is compatible with a multitude of underlying compute devices is drastically reduced. Further, OpenCL codebases are future-ready as existing code will be capable of executing on any future device that implements an OpenCL runtime.

Therefore, the primary relevance of this work is a contribution to the general knowledge of CUDA-to-OpenCL translation. We have drawn attention to a number of fundamental issues encountered when translating from CUDA to OpenCL. Further, where possible, we have worked to generalize the language idioms that we have encountered to reduce reliance on single-use, application-specific mappings of language constructs. This serves to facilitate a dialog on general methods for Affecting translations of CUDA to OpenCL. Thus, these insights provide benefits to both manual and automated translation efforts.

VII. Future Work

Based on the increased understanding of the nuances of CUDA-to-OpenCL translation, we can now target future development of our prototype to achieve the most rapid useful gains. Further, we have already begun reasoning about solutions to many of these idioms, which gives a substantial boost to their implementation. We have chosen to direct our immediate development efforts towards the following three areas.

1) Support for Separate Compilation: Support for separate compilation is a necessity for CU2CL to be considered a production-ready tool. Therefore, we have elected to devote much of our immediate effort towards generating and implementing a robust solution to this problem. We have already constructed several potential mechanisms for achieving this functionality. One such mechanism requires the generation of a small separate file into which the translator would progressively add various OpenCL initialization calls as they are identified in the individual CUDA source files in order to assemble unified initialization and finalization functions. Once a source file containing a main function is read, corresponding calls to initialization and finalization functions will be inserted. However, we are still in the process of generating other potential options and evaluating their respective merits.

2) Robust Error Reporting: The handling of untranslatable constructs presents a definite area of improvement for the translator. Currently only a small subset of unsupported CUDA operations emit intelligible errors during translation, but even in these cases, the error messages do not effectively direct programmer attention for manual translation. Further, a number of yet-to-be-supported constructs, including templates, library usage, and separate compilation originally resulted in segmentation faults without providing an indication as to the cause. We have implemented modifications to recognize and avoid such constructs, eliminating the majority of observed segmentation faults as a precursor to realizing a robust error reporting facility in CU2CL.

Next, we cannot simply comment out or remove untranslatable or difficult to translate regions as this destroys program semantics and will result in numerous silent errors. However, we also cannot simply pass over these constructs, emitting their original CUDA into the OpenCL output files, as this would cause inelegant compile and runtime failures in translated code. Hence, we propose an error reporting system that actively reports all such constructs, along with their source location at translation time, and adds easily searchable comment flags to the emitted OpenCL source to aid in the remaining manual translation efforts.

3) Enhanced Translator Coverage and Robustness: The resolution of the simple mistranslations that we have identified will noticeably improve the effective coverage and robustness of the translator. Thus, we aim to implement solutions to the aligned types, device identification, and kernel literal argument idioms as we have already conceptualized mechanisms to support their translation.

First, the translator must ensure that the data alignment attribute is properly recognized, the alignment width extracted, and appropriately reinserted using the OpenCL syntax when translating struct definitions.

Next, we have begun the process of adding support for kernel literal arguments by modifying the prototype to actively recognize expressions within kernel call parameter lists which are *not* variable identifiers and emit an appropriate notification within the terminal error stream as well as output OpenCL file. What remains is to introduce a mechanism for creating a temporary variable of appropriate type, using it to store the result of the expression, and passing a reference to the variable to OpenCL's setKernelArg function. However, some care must be taken to ensure that creation of the temporary variable does not create conflicts with other preexisting variables.

Finally, selection of a compute device among all devices within a system can be achieved by enumerating over all

Fig. 2. OpenCL Device Enumeration

```
int i;
cl_uint num_platforms = 0;
cl_uint num_devices = 0;
cl_uint p_dev_count, d_idx;

//allocate space for platforms
clGetPlatformIDs(0, 0, &num_platforms);
cl_platform_id * platforms =
    (cl_platform_id *) malloc(
    sizeof(cl_platform_id)
    * num_platforms);

//get all platforms
clGetPlatformIDs(num_platforms,
    &platforms[0], 0);

//count devices over all platforms
for (i = 0; i < num_platforms; i++) {
    p_dev_count = 0;
    clGetDeviceIDs(platforms[i],
        CL_DEVICE_TYPE_ALL, 0, 0,
        &p_dev_count);
    num_devices += p_dev_count;
}

//allocate space for devices
cl_device_id * devices = (cl_device_id *)
    malloc(sizeof(cl_device_id)
    * num_devices);

//get all devices
d_idx = 0;
for ( i = 0; i < num_platforms; i++) {
    clGetDeviceIDs(platforms[i],
        CL_DEVICE_TYPE_ALL, num_devices,
        &devices[d_idx], &p_dev_count);
    d_idx += p_dev_count;
    p_dev_count = 0;
}

//use a device
cl_context context = clCreateContext(
    0, 1, &devices[deviceID], 0, 0, 0);

free(devices);
free(platforms);
```

OpenCL devices over all OpenCL platforms and using an integer index into an array to return a single *cl_device_id* reference. Figure 2 demonstrates a mechanism for achieving this functionality using OpenCL.

4) Validation: Due to the inability of any translator to completely translate from CUDA to OpenCL, it is difficult to provide a complete quantification of the validity of automatically translated code. Currently, we have established the correctness of fully-translated applications based on direct comparison of output from the original CUDA application against the output from unedited automatically translated OpenCL. However, some applications will never fully translate due to untranslatable idioms. (At least until OpenCL is extended to provide similar functionality.) Therefore, work remains to establish a correctness metric for partial translations.

VIII. Conclusion

We have examined a large population of sample CUDA applications through the lens of a source-to-source translator prototype and identified a number of CUDA language idioms which, at present, emerge as significant issues in any source-to-source translation effort. We have provided a generalized classification of these idioms as a preliminary step towards creating robust mappings from the original CUDA constructs to functionally equivalent OpenCL code. Further, we have examined the relative frequency of these idioms within our sample population to aid in focusing future development efforts for greatest benefit. Finally, we have generated or are in the process of generating functionally equivalent OpenCL solutions to a number of the identified issues, which we aim to deploy in the next generation of the CU2CL translator prototype.

References

[1] 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

[2] 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\&rep=rep1\&type=pdf

[3] 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

[4] 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

[5] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.36

[6] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504194

[7] 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

[8] D. Nandakumar, "Automatic Translation of CUDA to OpenCL and Comparison of Performance Optimizations on GPUs," 2011. [Online]. Available: http://hdl.handle.net/2142/24279

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

[10] NVIDIA, "CUDA Zone," http://www.nvidia.com/object/cuda_home_new.html, 2012, [Online; accessed April 6, 2012].

[11] Khronos, "OpenCL," http://www.khronos.org/opencl/, 2012, [Online; accessed April 6, 2012].

[12] LLVM, "Clang: a C language family frontend for LLVM," http://clang.llvm.org/, 2012, [Online; accessed April 6, 2012].

[13] NVIDIA, "CUDA Toolkit & SDK," http://developer.nvidia.com/cuda-toolkit-sdk, 2012, [Online; accessed April 6, 2012].

[14] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5650274