Bridging OpenCL and CUDA: A Comparative Analysis and Translation*

Junghyun Kim, Thanh Tuan Dao, Jaehoon Jung, Jinyoung Joo† , and Jaejin Lee
Department of Computer Science and Engineering, Seoul National University, Seoul, Korea
{junghyun, thanhtuan, jaehoon, jinyoung}@aces.snu.ac.kr, jaejin@snu.ac.kr
http://aces.snu.ac.kr

ABSTRACT

Heterogeneous systems are widening their user-base, and heterogeneous computing is becoming popular in supercomputing. Among others, OpenCL and CUDA are the most popular programming models for heterogeneous systems. Although OpenCL inherited many features from CUDA and they have almost the same platform model, they are not compatible with each other. In this paper, we present similarities and differences between them and propose an automatic translation framework for both OpenCL to CUDA and CUDA to OpenCL. We describe features that make it difficult to translate from one to the other and provide our solution. We show that our translator achieves comparable performance between the original and target applications in both directions. Since each programming model separately has a wide user-base and large code-base, our translation framework is useful to extend the code-base for each programming model and unifies the efforts to develop applications for heterogeneous systems.

Categories and Subject Descriptors

D.3.4. Software [**Programming Languages**]: Processors

Keywords

Heterogeneous computing, CUDA, OpenCL, Source-to-source translation

*This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2013R1A3A2003664). ICT at Seoul National University provided research facilities for this study.

†This work was done when he was a graduate student at Seoul National University. He is currently with TmaxData.

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807621

1. INTRODUCTION

Accelerators are popular in high performance computing to boost performance and to reduce energy consumption. As of June 2015, 18% of the supercomputers in Top500 are equipped with accelerators[20], and it is expected that the number will be increasing continuously. To easily use accelerators, new programming models have been proposed and developed. Among those newborn programming models for accelerators, CUDA and OpenCL are widely used. NVIDIA proposed CUDA (Compute Unified Device Architecture)[14, 15]. It is continuously supported and maintained by NVIDIA. However, it only works for NVIDIA GPUs. OpenCL (Open Computing Language)[6] was proposed by Khronos group, and it is supported by many vendors including Altera, AMD, Apple, ARM, IBM, Imagination, Intel, MediaTek, NVIDIA, Qualcomm, Samsung, Xilinx. OpenCL works for any device that provides an OpenCL framework. It has code portability for different devices from different vendors. The devices include CPUs, GPUs, and other accelerators, such as Intel Xeon Phi coprocessors and FPGAs.

Even though OpenCL and CUDA share many concepts, the basic differences will remain (*e.g.*, the host code and the device code are separated or mixed, how to launch a kernel, how to allocate device memory, etc.). To port programs written in CUDA to OpenCL manually, or *vice versa*, the programmer should be familiar with both programming models. Moreover, the porting process would be cumbersome and error-prone even if the programmer knows both programming models very well. If there is an automatic translator, the programmer who knows only CUDA can execute a CUDA program on an OpenCL framework easily, or *vice versa*.

Since each programming model has a wide user-base and many programs written in it, translating one to the other would be very useful for various reasons. It would be good if we could translate a CUDA program to an equivalent OpenCL program to exploit OpenCL's portability. While CUDA programs run only on NVIDIA GPUs, OpenCL programs run on any computing devices, such as CPUs, GPUs, and accelerators from various vendors without any modification, as long as the device supports an OpenCL platform.

On the other hand, translating an OpenCL program to CUDA is also useful. NVIDIA's current OpenCL framework only supports OpenCL 1.1 even though OpenCL 2.0[6] has been released in November 2013. This implies that an OpenCL program written in OpenCL 1.2 or 2.0 do not work with NVIDIA's current OpenCL framework. However,

translating a program written in OpenCL 1.2 or 2.0 to an equivalent CUDA program makes the CUDA program run on NVIDIA GPUs without NVIDIA's OpenCL framework. Moreover, since CUDA provides a richer development environment than OpenCL for NVIDIA GPUs (*e.g.*, profilers), the translated CUDA program can exploit CUDA's richer development environment.

Since translating OpenCL/CUDA programs in both directions is meaningful, and the demand for the translation has been increasing, we describe an automatic translation framework between OpenCL and CUDA in this paper and show the effectiveness of our approach. The API functions in the source framework are implemented as wrapper functions using features available in the target framework. If such wrappers cannot be made for an API function in the source, we statically translate it to the target framework.

As CU2CL[5] discussed, there are many challenges in source-to-source translation between CUDA and OpenCL, especially problems incurred by separate compilation of multiple files. Translated type information needs to be propagated from one file to another. However, this is impossible unless we rely on a whole-program analysis at the linker level. Our wrapper function approach provides a nice solution to this problem by propagating type information at run-time through the execution path. Our hybrid approach that combines wrapper functions and source-to-source translation alleviates many difficulties of static translation.

Since there exist inherent differences between CUDA and OpenCL, we do not insist that all programs written in one programming model are translatable to those written in the other programming model. Our translation framework provides a rapid development and prototyping tool in the transition from one programming model to the other. Currently, we are targeting OpenCL 1.2 and CUDA compute capability 3.5 for this paper.

The major contributions of this paper are the following:

- We analyze the similarities and differences between OpenCL and CUDA in detail. The differences make porting programs written in one model to the other difficult.
- We describe how a program written in one model can be translated to an equivalent program written in the other. We propose a hybrid translation approach that combines wrapper functions and source-to-source translation. To the best of our knowledge, this paper is the first one that presents OpenCL-to-CUDA translation techniques.
- CUDA textures play a key role in image processing to boost performance. We translate CUDA textures to OpenCL images in CUDA-to-OpenCL translation. This issue has never been addressed in previous studies.
- We evaluate our approach with 54 OpenCL applications for the OpenCL-to-CUDA translation, and with 39 CUDA applications for the CUDA-to-OpenCL translation. We compare the performance of the applications in each translation direction on an NVIDIA GPU. In addition, we evaluate resulting OpenCL applications of the CUDA-to-OpenCL translation on an AMD GPU to see the portability of translated OpenCL programs.
- Since the Rodinia benchmark suite provides both OpenCL and CUDA versions for the same application, we also compare the performance of the translated version in each programming model with that of the original version in the same model.

The rest of the paper is organized as follows. Section 2 describes some prior translation approaches between OpenCL and CUDA. Section 3 explains the similarities and differences between OpenCL and CUDA and presents how programs are translated from one programming model to the other. Section 4 and Section 5 describe the techniques used to translate memory object allocation and image objects related features. Section 6 presents the evaluation result of our approach. Finally, Section 7 concludes the paper.

2. RELATED WORK

Some efforts have been made to execute CUDA programs in different platforms rather than just GPUs. MCUDA[19] allows CUDA programs to be executed on multicore CPUs with help from a source-level translator and a runtime system. Ocelot[3] also allows CUDA programs to be executed on multicore CPUs. It includes a dynamic binary translator to translate the PTX ISA to x86 and other ISAs. MCUDA and Ocelot allow CUDA programs to be executed on multicore CPUs, but it is impossible to run the translated program on GPUs or accelerators other than NVIDIA's. Our approach translates a CUDA program to an OpenCL program at the source level to take advantage of OpenCL's portability, *i.e.*, the resulting OpenCL program can be executed on any architecture as long as the architecture supports OpenCL.

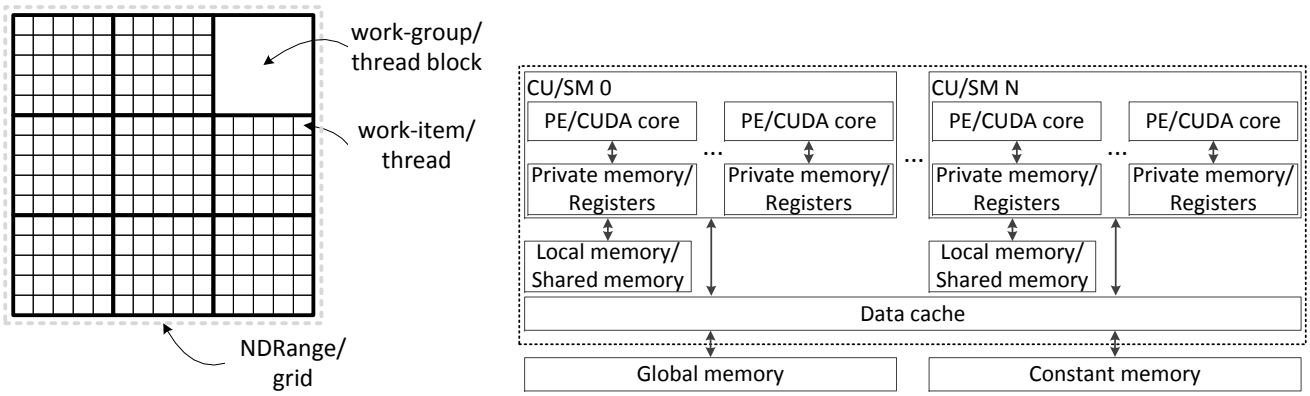
Some studies have been done to translate a CUDA program to an OpenCL program[8, 12, 13, 17]. CU2CL[5, 12, 17] tries to translate CUDA APIs to the corresponding OpenCL APIs. It not only translates a CUDA program to an OpenCL program but also helps programmers to develop the translated OpenCL code seamlessly. To do so, CU2CL tries to statically translate each of CUDA host API functions to a corresponding OpenCL API function.

SWAN[8] is another CUDA-to-OpenCL translator at the source level. Unlike CU2CL, it requires manual translation of the CUDA code to the SWAN code. Then, the SWAN code is automatically translated to OpenCL code. The SWAN code can also be translated to CUDA code.

CUDAtoOpenCL[13] is yet another source-to-source translator using the Cetus source-to-source compilation framework[2]. However, it does not provide mechanisms of translating CUDA textures to OpenCL.

Previous CUDA-to-OpenCL translators are based on static translation. Since the host code is more complicated compared to the device code, it is very hard to statically translate all host code written in CUDA to OpenCL. One example is separate compilation. The source code of a program can be separated into several files. Consider a function `foo` that receives a memory object as a parameter. Since CUDA uses a variable of `void*` as a handle of a memory object, the parameter type is `void*`. To distinguish it from a generic pointer, the translator should see the call site. However, if the call site is outside of the file where `foo` is defined, the translator cannot determine if the parameter is a handle or not. It requires uncommon and difficult whole program analysis at the source level.

This problem can be solved using wrapper functions and run-time type cast. By casting `cl_mem` to `void*` in a wrapper function at run time, an OpenCL memory object is treated



(a) Kernel execution space (b) Accelerator architecture

Figure 1: The kernel execution space and accelerator architecture of OpenCL and CUDA.

as a CUDA memory object. For example, CUDA API function `cudaMalloc()` returns a memory object in its first parameter. An OpenCL wrapper `cudaMalloc()` in CUDA-to-OpenCL translation calls `clCreateBuffer()` and the returned variable of type `cl_mem` is cast to a variable of type `void*`. Then the value is returned through the first parameter of wrapper `cudaMalloc()`.

Our approach is hybrid in the sense that it exploits both static translation and wrapper functions implemented in a runtime library. For all the host API functions but three special cases (CUDA kernel calls, `cudaMemcpyFromSymbol()`, and `cudaMemcpyToSymbol()`), a wrapper function can be made. For the three special cases, we perform source-to-source translation.

Currently, we cannot find any OpenCL-to-CUDA translator. To our knowledge, this paper is the first paper that presents differences of CUDA and OpenCL extensively and provides a translation solution in both directions, CUDA-to-OpenCL and OpenCL-to-CUDA. In addition, no previous study addresses the way how to translate CUDA textures to OpenCL images.

3. OVERVIEW OF TRANSLATION

In this section, we describe common techniques used in the OpenCL-to-CUDA translation and the CUDA-to-OpenCL translation. Then, we describe similarities and differences between OpenCL and CUDA. Translation of similar features between OpenCL and CUDA is straightforward. However, differences introduce non-trivial translation. We describe how each different feature is translated between OpenCL and CUDA. Translation techniques are implemented at the source level using in clang 3.3[11].

3.1 Execution and Memory Models

Before we describe our translation techniques, we briefly introduce the execution model and the memory model of OpenCL and CUDA. Figure 1 shows the kernel execution space and platform architecture of OpenCL and CUDA. Since they often use different names for the same component, some components in the figure have two different names separated by a slash. The first is for OpenCL and the last is for CUDA.

Figure 1 (a) shows a kernel execution space of OpenCL and CUDA. Before a kernel is launched to a compute device (e.g., an accelerator), the space is defined by the host program. Each point in the space is occupied by a work-item in OpenCL. A work-item is an execution instance of the kernel. Several work-items are grouped to a work-group. The whole execution space is called an NDRange.

CUDA has similar terms to indicate the kernel execution space. For example, a work-item and a work-group correspond to a thread and a block in CUDA, respectively. An NDRange corresponds to a grid in CUDA, but there is a difference. An NDRange is a group of work-items while a grid is a group of blocks. For example, the size of the NDRange in Figure 1 (a) is (15, 15), and the size of the grid is (3, 3). This is one thing to be taken care when translating between them.

To specify memory spaces allocated to a variable used in the device code, special keywords are provided by both programming models. OpenCL uses `__private`, `__local`, `__global`, and `__constant` to specify spaces in private, local, global, and constant memory, respectively. CUDA uses `__shared__`, `__device__`, and `__constant__` for shared, global, and constant memory, respectively.

3.2 Commonalities in Translation

Host code translation. The host code is basically untouched. Each host API function becomes a wrapper function that implements the same function in the target programming model. There are three exceptional cases for the CUDA-to-OpenCL translation: CUDA kernel calls, `cudaMemcpyToSymbol()`, and `cudaMemcpyFromSymbol()`. A CUDA kernel call cannot be parsed in an OpenCL framework because of different syntax. CUDA API functions `cudaMemcpyToSymbol()` and `cudaMemcpyFromSymbol()` transfer data between the host and a device or between devices using a non-local variable located in the device memory. Since an OpenCL program cannot have a non-local variable shared by the host and a device, we cannot implement these functions using wrappers.

On the other hand, since all host-side OpenCL constructs are implemented with host API functions, the OpenCL-to-CUDA translation does not require automatic source code translation. Wrappers are enough.

Device code translation. The device code is translated by our automatic translator in each direction. OpenCL device code is translated to CUDA device code directly and

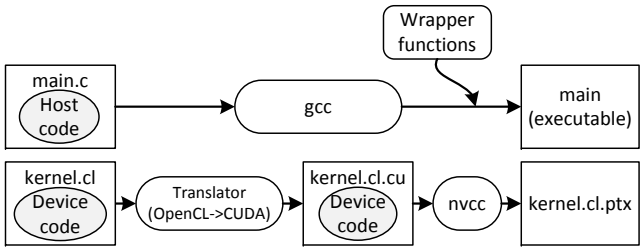


Figure 2: The process of translating OpenCL to CUDA.

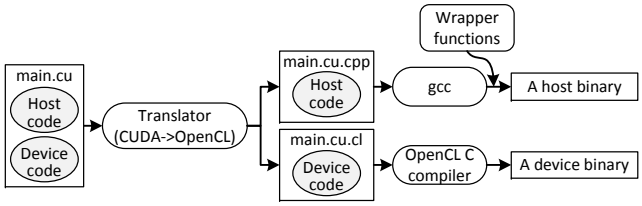


Figure 3: The process of translating CUDA code to OpenCL.

vice versa. In addition, most device constructs and built-in functions have a one-to-one correspondence between CUDA and OpenCL. Even though they look different, their meaning is exactly the same. Our translator exploits the one-to-one correspondence and implements the translation.

3.3 One-to-one Correspondence

Most host API functions have the same meaning in OpenCL and CUDA. They have a one-to-one correspondence, and their translation from one framework to the other is straightforward. For example, `clCreateBuffer()` and `cudaMalloc()` are used to allocate a memory object in the global memory. All functions that have a one-to-one correspondence are translated using wrapper functions.

Both CUDA and OpenCL have language extensions (*e.g.*, qualifiers) and built-in functions to support computations on devices. The translation of these extensions and built-in functions from one framework to the other is also straightforward. Due to the page limit, we leave out the detailed description of the other similarities. This paper focuses on the differences that introduce non-trivial translation between OpenCL and CUDA.

3.4 Executable Building Process

OpenCL and CUDA have different compilation policies even though both distinguish the host code and the device code. OpenCL forces the device code be separated from the host code even though the device code can be embedded in the host code in the form of character strings. On the other hand, the device code is literally embedded in the middle of the host code in CUDA.

OpenCL to CUDA. The process of translating OpenCL to CUDA is illustrated in Figure 2. We translate the device code and the host code separately. The OpenCL device code (*e.g.*, `kernel.cl`) is directly translated to the CUDA device code (*e.g.*, `kernel.cl.cu`) by our source-to-source translator. All OpenCL host API functions are implemented as wrapper functions. The wrappers are implemented with CUDA driver API functions.

In OpenCL, the device code is built using `clBuildProgram()` at run-time. We implement this host API function in the following way. First, we translate the OpenCL device code to the CUDA device code by invoking the source-to-source translator at run-time. Then, the CUDA device code is compiled to PTX code using NVIDIA's `nvcc` compiler. The resulting PTX code is loaded by invoking `cuModuleLoad()`.

CUDA to OpenCL. The process of the CUDA-to-OpenCL translation is shown in Figure 3. In CUDA, the host and the device code are mixed in source files. We preprocess these files using our translator to separate the device code from the host code. For each source file, the translator generates two new files. For example, assume that we have a CUDA source file named `main.cu`. Our translator generates an OpenCL host code file named `main.cu.cpp` and an OpenCL device code file named `main.cu.cl`. Then, a native compiler, such as gcc, compiles the host code and generates an executable binary. At this time, the OpenCL wrapper library functions are linked to the executable.

Unlike CUDA, OpenCL builds the device code at run-time. To follow the semantics of CUDA, the device code should be built at the same time when the host code is built. However, this is inconvenient for OpenCL users who want to execute the same device code on different devices. It is preferable to build the device code at run-time. Thus, our translation framework builds the device code when any CUDA API function is called for the first time at run-time.

3.5 Kernel Execution

How to invoke a kernel function in the host code is much different between OpenCL and CUDA. Figure 4 shows an example of launching a kernel. To launch a kernel in OpenCL, kernel arguments need to be set with `clSetKernelArg()` in the host code at line 4 in Figure 4 (b). Then, a kernel is launched using `clEnqueueNDRangeKernel()` with the size of the NDRange `gws` and the size of a work-group `lws` at line 26 in Figure 4 (b).

To launch a kernel in CUDA, an execution configuration (*i.e.*, <<< and >>>) is used at line 22 of Figure 4 (c). It has three arguments: the size of the grid, the size of a thread block, the size of dynamic shared memory per thread block used by the kernel. In addition, Figure 4 (d) shows how to launch a kernel using CUDA driver API.

OpenCL to CUDA. To translate the kernel launching API function (`clEnqueueNDRangeKernel()`), we use the corresponding CUDA driver API function `cuLaunchKernel()` that simplifies our translation (Figure 4 (d)). The parameters of `cuLaunchKernel()` require an array of the kernel arguments and the amount of the local memory size used in the kernel. This information is collected when `clSetKernelArg()` is called at run-time. Note that `clSetKernelArg()` is implemented as a wrapper.

CUDA to OpenCL. As described earlier, our translator translates a CUDA kernel call to an appropriate OpenCL kernel launch sequence. Since a CUDA kernel call contains all the information to launch a kernel, our source-to-source translator converts it to an OpenCL kernel call (`clEnqueueNDRangeKernel()`) and multiple `clSetKernelArg()` functions with appropriate kernel index space definitions.

```
__constant const int static_constant[32] = {1,2,3,4};

__kernel void opencl_kernel(int n,
  __local int* dyn_shared1, __local int* dyn_shared2,
  __constant int* dyn_constant, __global int* dyn_global) {
  __local int static_shared[32];
  ...
}
```

(a) OpenCL device code

```
int main(void) {
  int buf[32];
  ...
  clSetKernelArg(opencl_kernel, 1, 32*sizeof(int), NULL);
  clSetKernelArg(opencl_kernel, 2, 32*sizeof(int), NULL);

  cl_mem mem_dyn_constant;
  mem_dyn_constant = clCreateBuffer(context,
                  CL_MEM_READ_ONLY,
                  32*sizeof(int), NULL, &cl_error);
  clEnqueueWriteBuffer(cmq, mem_dyn_constant, CL_TRUE,
                  0, 32*sizeof(int), buf, 0, NULL, NULL);
  clSetKernelArg(opencl_kernel, 3, sizeof(cl_mem),
                 (void*)&mem_dyn_constant);

  cl_mem mem_dyn_global;
  mem_dyn_global = clCreateBuffer(context,
                  CL_MEM_READ_WRITE,
                  32*sizeof(int), NULL, &cl_error);
  clEnqueueWriteBuffer(cmq, mem_dyn_global, CL_TRUE,
                  0, 32*sizeof(int), buf, 0, NULL, NULL);
  clSetKernelArg(opencl_kernel, 4, sizeof(cl_mem),
                 (void*)&mem_dyn_global);

  size_t gws[3] = {N, 1, 1}; size_t lws[3] = {32, 1, 1};
  clEnqueueNDRangeKernel(cmd_queue, opencl_kernel,
    3, NULL, gws, lws, 0, NULL, NULL);
  ...
}
```

(b) OpenCL host code

```
__constant__ int static_constant[32] = {1,2,3,4};
__constant__ int static_constant_runtime_init[32];
__device__   int static_global[32];

__global__ void cuda_kernel(int n, int* dyn_global) {
   __shared__ int static_shared[32];
   extern __shared__ int dynamic_shared[];
   ...
}

int main(void) {
  ...
  int buf[32] = {1,2,3,4};
  cudaMemcpyToSymbol(static_constant_runtime_init,
    buf, 32*sizeof(int));
  cudaMemcpyToSymbol(static_global, buf, 32*sizeof(int));

  int* dyn_global;
  cudaMalloc(&dyn_global, 32*sizeof(int));
  cudaMemcpy(dyn_global, buf, 32*sizeof(int),
    cudaMemcpyHostToDevice);
  cuda_kernel<<<N/32,32,32*sizeof(int)>>>(N, dyn_global);
   ...
}
```

(c) CUDA code using CUDA runtime API

```
int main(void) {
  ...
  void* kernelParams[2] = {&N, &dyn_global};
  cuLaunchKernel(cuda_kernel, N/32, 1, 1, 32, 1, 1,
    32*sizeof(int), hStream, kernelParams, NULL);
  ...
}
```

(d) CUDA code using CUDA driver API

Figure 4: Sample programs in OpenCL and CUDA.

3.6 Language Extensions

C++. CUDA and OpenCL allow C++ features in the host code. OpenCL does not support any C++ features in the device code while CUDA supports some C++ features in the device code, such as templates, reference types, and C++ type casting.

Built-in vector types. CUDA supports one-, two-, three-, and four-component vectors derived from the basic integer and floating-point types. OpenCL supports two-, three-, four-, eight-, and sixteen-component vectors. OpenCL does not support one-component vectors, and CUDA does not support eight- and sixteen-component vectors. CUDA supports a vector of type `longlong`, but OpenCL does not. OpenCL provides various ways to access components (`x, y, z, w, hi, lo, even, odd,` and `S`i) while CUDA provides a subset of them (`x, y, z,` and `w`). For example, let `v` be a vector. `v.lo.x` and `v.xx` are allowed in OpenCL while they are not allowed in CUDA. Note that `v.lo.x` refers to the first component of the lower half of `v`, and `v.xx` is a two-component vector expanded from the first component of `v`.

Address space qualifiers. OpenCL and CUDA have various address space qualifiers that indicate where the variable is located. In most of the cases, only keywords for the qualifiers are different. An exception is pointers. There are two address spaces related to a pointer: the space of the pointer and the space of data it points to. An address space qualifier in OpenCL indicates the space of data while it indicates the space of the pointer in CUDA. For example, in OpenCL, `__global int* p` indicates the data pointed by `p` is located in the global memory. In CUDA, `__device__ int* p` indicates that `p` itself is located in the global memory.

OpenCL to CUDA. OpenCL does not allow any C++ features in the device code while CUDA allows. Hence, nothing needs to be taken of for OpenCL-to-CUDA in terms of C++ features.

Since OpenCL supports eight- and sixteen-component vectors that are not supported by CUDA, we implement them using C structures in CUDA. Since CUDA supports only a subset of OpenCL vector component expressions, our translator translates OpenCL vector component expressions to multiple CUDA expressions. For example, OpenCL `v1.lo=v2.lo;` is translated to CUDA `v1.x=v2.x; v1.y=v2.y;` where the type of `v1` and `v2` is a vector type `float4`.

The translation of address space qualifiers for variables other than pointers is straightforward. For pointers, our translator just removes the address space qualifiers.

CUDA to OpenCL. According to the OpenCL 2.1 provisional specification[7] released in January 2015, OpenCL will support C++ features in the device code. Currently, our translator translates some C++ features in CUDA to C features in OpenCL. First, a reference variable is translated to a pointer variable. Second, a template function is specialized. Finally, a C++-style type casting (*e.g.*, `static_cast<int>(a)`) is translated to C-style type casting (*e.g.*, `(int) a`). Function pointers and C++ classes cannot be translated currently. We expect that C++ features used in CUDA device code will be able to be directly translated to OpenCL 2.1 kernel code without any difficulties.

There are two vector-type problems in the CUDA-to-OpenCL translation: one-component vectors and type `long-`

Table 1: Device memory allocation

		OpenCL	CUDA
Local/shared memory allocation	Static	O	O
	Dynamic	O	O
Constant memory allocation	Static	O	O
	Dynamic	O	X
Global memory allocation	Static	X	O
	Dynamic	O	O

`long`. In our translation, a one-component vector is replaced with a scalar type in OpenCL. OpenCL does not support a vector of type `longlong`. However, it is identical to type `long` because the size of `long` and `longlong` is the same. Thus, our translator replaces a vector of `longlong` to a vector type with `long`.

The translation of address space qualifiers for variables other than pointers is straightforward. For pointers, our translator adds an appropriate address space qualifier to a pointer using type information. However, there is a problem when a pointer points to two or more different address spaces in CUDA. In this case, our translator generates a new pointer variable for each address space.

3.7 Model-specific Features

There are features supported by only one of CUDA and OpenCL. They cannot be translated because the other programming model does not provide any counterpart.

Host API and kernel built-in functions. CUDA exposes more hardware features to programmers than OpenCL because it supports only NVIDIA GPUs. CUDA has more built-in functions related to the GPU hardware, such as `__all`, `__shfl`, `clock`, etc., which do not have any counterpart in OpenCL.

There are two built-in functions that have different semantics between OpenCL and CUDA: atomic increment and decrement. CUDA's `atomicInc()` receives two parameters: a variable and the maximum value. If the value of the variable is less than the maximum value, it is increased by one. Otherwise, it is set to zero. However, OpenCL's `atomic_inc()` receives a variable only. It is always increased by one. Due to the different semantics, `atomicInc()` cannot be directly translated to `atomic_inc()`. However, implementing `atomic_inc()` using `atomicInc()` is not a problem.

`cudaMemGetInfo()` in CUDA returns the size of free and total global memory in a device. However, there is no corresponding API function in OpenCL.

Unified virtual address space. CUDA supports *unified virtual address space*. It enables a memory region allocated by `cudaHostAlloc()` to be accessed by both the host and devices. Thus, the host can pass a structure containing pointers that point to the device memory regions to a kernel. This functionality can be implemented in OpenCL using `clSVMalloc()` that is available in OpenCL 2.0[6]. Since our translator targets OpenCL 1.2 currently, we do not translate this feature at this time.

PTX code. CUDA can load and execute low-level PTX code while OpenCL does not support PTX code.

Subdevices. OpenCL provides a feature that divides a device into multiple sub-devices using `clCreateSubDevices()`. There is no corresponding feature in CUDA. Thus, the translation is practically impossible, and our framework does not support it.

4. DEVICE MEMORY ALLOCATION

In this section, we describe how our translator assigns device memory spaces in the target model to various types of memory objects in the source model. Table 1 summarizes device memory allocation schemes available in OpenCL and CUDA. In the table, `O` stands for 'available' and `X` stands for 'not available'. In addition, *static* means the allocation occurs at compile time while *dynamic* means the allocation occurs at run time.

Before describing how to allocate memory objects in the device, we first discuss how device memory objects are handled in the host code. In OpenCL, `clCreateBuffer()` is used to allocate a device memory object. It returns a handle (a variable of type `cl_mem`) for the allocated memory object (line 8 in Figure 4 (b)). OpenCL API functions manipulating the memory object in the host code use the handle returned by `clCreateBuffer()`. Type `cl_mem` is the same type as `struct _cl_mem*` and defined by `typedef` in the OpenCL header file `cl.h` provided by Khronos Group. Thus, `cl_mem` is a pointer type.

In CUDA, the handle of a memory object is a pointer. To allocate a memory object, `cudaMalloc()` is used. It returns the address of the allocated memory object in the first parameter of `cudaMalloc()` (line 19 in Figure 4 (c)). CUDA API functions manipulating the memory object in the host code use the pointer (the address of the memory object).

Our translation framework uses wrapper functions for device memory allocation API functions, and the handle for a device memory object is a pointer both in OpenCL and in CUDA. The translation is done by just casting one type to the other type. For example, in CUDA-to-OpenCL translation, `cudaMalloc()` is a wrapper of OpenCL `clCreateBuffer()`. `cudaMalloc()` returns a variable of type `void*`. Thus, its type is cast from `cl_mem` that is the type of a pointer returned by `clCreateBuffer()`.

4.1 Local/Shared Memory Spaces

In OpenCL, there are two ways of allocating a memory object in the local memory. One is static allocation that allocates a memory object in a kernel function with `__local` (line 6 in Figure 4 (a)). The other is dynamic allocation that specifies the size of the memory object using `clSetKernelArg()` (line 4 in Figure 4 (b)). The kernel receives it through a kernel parameter qualified with `__local` (line 4 in Figure 4 (a)). The size of the variable `dyn_shared1` is the same as the third parameter of `clSetKernelArg` (*i.e.*, `32*sizeof(int)` at line 4 in Figure 4 (b)).

In CUDA, there are also two ways of allocating a memory object in the shared memory. One is static allocation in a kernel function with qualifier `__shared__` (line 6 in Figure 4 (c)). This method is similar to that of OpenCL. The other is dynamic allocation. However, it differs from the case of OpenCL. The size of the shared memory is specified in the CUDA kernel call with the third parameter of the execution configuration (*i.e.*, <<<...>>>) (line 22 in Figure 4 (c)). The dynamically allocated shared memory object is accessed through a variable declared with qualifiers `__shared__` and `extern` (line 7 in Figure 4 (c)).

However, there is no way to dynamically allocate multiple shared memory objects for the same kernel in CUDA. Unlike CUDA, OpenCL allows dynamically allocating multiple local memory objects for the same kernel using `clSetKernelArg()` multiple times (line 4 in Figure 4 (a) and line 4-5

```
1  __kernel void opencl_kernel(int n,
2   __local int dyn_shared1[], __local int dyn_shared2[],
3   __constant int* dyn_const1, __constant int* dyn_const2) {
4    ...
5  }
```

(a) OpenCL device code

```
1  extern __shared__ char __OC2CU_shared_mem[];
2  __constant__ char __OC2CU_const_mem[MAX_CONST_SIZE];
3
4  __global__ void opencl_kernel(int n,
5       size_t shared1_size, size_t shared2_size,
6       size_t const1_size, size_t const2_size) {
7    int *dyn_shared1, *dyn_shared2;
8    dyn_shared1=(int *)(__OC2CU_shared_mem);
9    dyn_shared2=(int *)(__OC2CU_shared_mem+shared1_size);
10
11   int *dyn_const1, *dyn_const2;
12   dyn_const1=(int*)(__OC2CU_const_mem);
13   dyn_const2=(int*)(__OC2CU_const_mem + const1_size);
14   ...
15 }
```

(b) Translated CUDA device code

Figure 5: Dynamically allocating shared and constant memory spaces in OpenCL-to-CUDA.

in Figure 4 (b)).

OpenCL to CUDA. Translating static local memory allocation is straightforward. We replace OpenCL `__local` with CUDA `__shared__`. On the other hand, translating dynamic local memory allocation is complicated because OpenCL allows allocating multiple dynamic local memory objects while CUDA allows allocating only a single shared memory object for a kernel. If multiple shared memory objects are allocated in the OpenCL source, a big single shared memory object is allocated in the CUDA target and is divided into multiple memory spaces in the CUDA kernel function. This process is shown in the target CUDA device code at line 8-9 in Figure 5 (b). The total size of shared memory objects is obtained from `clSetKernelArg()` at run time.

CUDA to OpenCL. Translating statically allocating shared memory space is straightforward. We replace `__shared__` with `__local`. Translating dynamically allocating shared memory spaces introduces relatively complicated source-to-source translation because the size of the shared memory object is specified in the third parameter of a CUDA kernel call, and the associated variable in the CUDA kernel is declared with `extern` and `__shared__`. For example, the size of the shared memory object in CUDA (the 3rd parameter at line 22 in Figure 4 (c)) is translated to a call of `clSetKernelArg()` (line 4 in Figure 4 (b)) in OpenCL. The associated variable is added to the parameter list of the OpenCL kernel (line 4 in Figure 4 (a)).

4.2 Constant Memory Spaces

In OpenCL, there are two ways of allocating a memory object in the constant memory. One is static allocation that declares a variable with `__constant` in the device code (line 1 in Figure 4 (a)). In this case, there is no way to transfer data from the host to a device. Initialization should be done in the device code. The other is dynamic allocation that allocates a memory object using `clCreateBuffer()` with the flag `CL_MEM_READ_ONLY` (line 8 in Figure 4 (b)). Then, the contents can be copied to the memory object using `clEnqueueWriteBuffer()` from the host (line 11 in Figure 4 (b)). Then, the created buffer is passed to a kernel as an argument using `clSetKernelArg()` (line 13 in Figure 4 (b)). In the device code, the corresponding kernel parameter is qualified with `__constant` (line 5 in Figure 4 (a)).

On the other hand, CUDA supports only static allocation. In other words, the size of the constant memory object cannot be determined at run time. However, the contents of the object can be initialized either at compile time or run time. The compile time initialization is similar to the static allocation in OpenCL. An example of this is shown at line 1 in Figure 4 (c). Line 2 shows the declaration of a variable that is initialized at run time in CUDA. It is initialized in the host code via `cudaMemcpyToSymbol()` at line 14 in Figure 4 (c) at run time.

OpenCL to CUDA. Similar to statically allocating local memory spaces, we replace OpenCL `__constant` with CUDA `__constant__`. However, CUDA does not support dynamically allocating constant memory spaces. To solve this problem, a non-local variable `__OC2CU_const_mem` is declared with the maximum size of the available constant memory in the CUDA target (line 2 in Figure 5 (b)). A constant memory object pointer in the OpenCL kernel is replaced with the size of the corresponding constant memory object in the CUDA kernel (line 6 in Figure 5 (b)). Then, a pointer to the object is declared in the CUDA kernel body with an appropriate offset in the constant memory space `__OC2CU_const_mem` (line 11-13 in Figure 5).

If the object is declared with `__constant` in the OpenCL kernel, its space is allocated in the constant memory. One thing needed to be taken care of is that there is no way to distinguish whether a memory object is placed in the global memory or the constant memory when it is created by `clCreateBuffer()`. The place is known when a kernel that accesses it launches. When `clEnqueueWriteBuffer()` is called for the memory object before launching the kernel, we do not know where the memory object is allocated.

To solve this problem, we copy the data to a global memory space when `clEnqueueWriteBuffer()` is called. When the kernel is launched at `clEnqueueNDRangeKernel()` in the CUDA target, the data in the global memory space is copied to the constant memory space if the memory object is declared with `__constant` in the kernel. Note that we know all the information about the kernel when we launch the kernel because the OpenCL kernel has been translated to the CUDA kernel before the kernel launches.

CUDA to OpenCL. CUDA allows only statically allocating constant memory spaces. However, the allocated space can be initialized either statically or dynamically. The static initialization of a constant memory object is translated easily by replacing CUDA `__constant__` with OpenCL `__constant`.

For dynamic initialization, CUDA uses `cudaMemcpyToSymbol()` to copy data to such a variable in the device. The first parameter of this function is the name of the variable as shown at line 14 in Figure 4 (c). However, such dynamic initialization is not allowed in OpenCL because the same variable cannot be seen both in the host code and in the device code. Our source-to-source translator translates the dynamic initialization in CUDA to the dynamic constant memory space allocation and a series of API function calls in OpenCL. Our source-to-source translator performs the following:

1. A variable, say v, is declared with `__constant` in the

argument list of the OpenCL kernel.
2. A `cl_mem` variable is declared in the host code.
3. A memory object is created using `clCreateBuffer()` in the host code.
4. The memory object is copied to the device using `clEnqueueWriteBuffer()` that corresponds to `cudaMemcpyToSymbol()` in the CUDA host.
5. The memory object is passed through v to the kernel using `clSetKernelArg()` in the host code.

`static_constant_runtime_init` in Figure 4 (c) is the variable seen by both the host and the device. It is inserted to the kernel parameter list at line 5 in Figure 4 (a). The lines 14-15 in the CUDA code Figure 4 (c) are translated to lines 7-14 in the OpenCL code Figure 4 (b).

4.3 Global Memory Spaces

In CUDA, there are two ways of allocating a memory object in the global memory. One is static allocation that declares a variable with `__device__` in the device code (line 3 in Figure 4 (c)). To transfer data from the host to the allocated memory object, `cudaMemcpyToSymbol()` is used (line 16 in Figure 4 (c)). The other is dynamic allocation that allocates a memory object using `cudaMalloc()` (line 19 in Figure 4 (c)). Data can be copied to the memory object using `cudaMemcpy()` from the host (line 20 in Figure 4 (c)). Then, the memory object is passed as a kernel argument at the kernel launch (line 22 in Figure 4 (c)).

OpenCL does not support static allocation in the global memory. OpenCL forces the device code be separated from the host code. Thus, it is impossible to allocate a non-local variable that is seen by both the host code and the device code. As a result, it is impossible to make an interface that is similar to `cudaMemcpyToSymbol()` in OpenCL. On the other hand, OpenCL supports dynamic allocation in the global memory via `clCreateBuffer()` (line 17 in Figure 4 (b)). `clEnqueueWriteBuffer()` transfers data to the allocated memory object (line 20 in Figure 4 (b)). The buffer (*i.e.*, the memory object) is passed as a kernel argument using `clSetKernelArg()` (line 22 in Figure 4 (b)). In the device code, the corresponding kernel parameter has qualifier `__global` (line 5 in Figure 4 (a)).

OpenCL to CUDA. OpenCL allows only dynamic allocation of global memory spaces. Our OpenCL-to-CUDA kernel translator removes `__global` in the parameter list of the target CUDA kernel function. In the host code, `cudaMalloc()` is called in `clCreateBuffer()` to dynamically allocate a global memory space.

CUDA to OpenCL. Since static global memory allocation is impossible in OpenCL, our translator translates it to the dynamic global memory allocation in OpenCL. Our translator performs the same task as that of the dynamic initialization of the constant memory space allocation. The variable `static_global` in Figure 4 (c) is the variable seen by both the host and the device. It is inserted to the OpenCL kernel parameter list at line 5 in Figure 4 (a). The code at lines 18-21 Figure 4 (c) are translated to the OpenCL code at lines 16-23 in Figure 4 (b).

To dynamically allocate global memory spaces, `cudaMalloc()` and `cudaMemcpy()` become OpenCL wrapper functions.

```
1  class CLImage {
2  public:
3    void* ptr;
4    size_t size;
5    cl_image_format image_format;
6    cl_image_desc image_desc;
7    size_t row_pitch;
8    size_t slice_pitch;
9    size_t elem_size;
10   size_t channels;
11 };
12 typedef CLImage* image_t;
13 typedef CLImage* image1d_t;
14 typedef CLImage* image2d_t;
```

Figure 6: The structure of `CLImage`.

5. IMAGE PROCESSING

Both OpenCL and CUDA support image processing mechanisms for GPUs to speed up processing one-, two-, or three-dimensional image data. Both programming models provide special features, such as channel description, normalized coordinates, addressing mode, and filter mode.

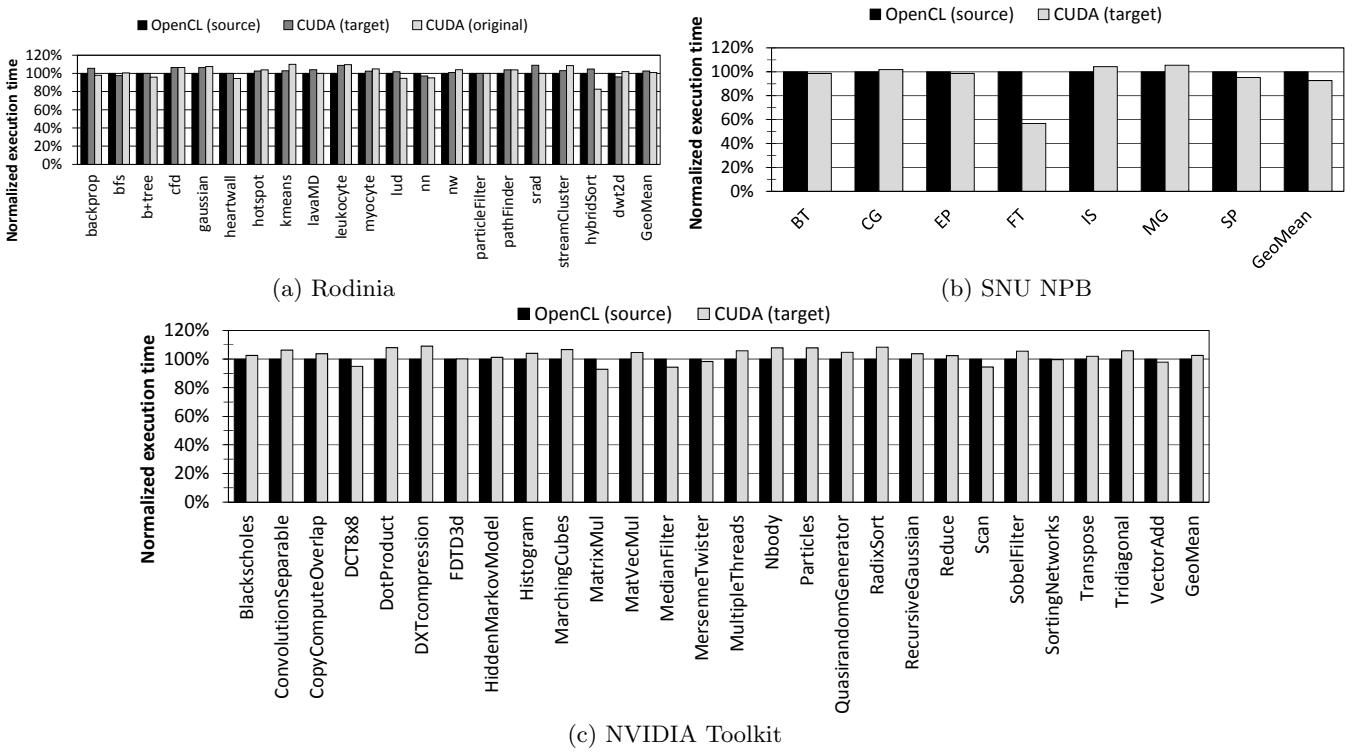
However, there is a noticeable difference between CUDA and OpenCL in image processing. The maximum width for a 1D texture of CUDA linear memory is up to 2^{27} pixels. On the other hand, the maximum width of a 1D image buffer is up to the maximum width of a 2D image in OpenCL. The maximum width $\times$ maximum height of a 2D image is typically 65536$\times$65535 for NVIDIA GPUs. Because of this discrepancy, 1D textures of CUDA linear memory are not completely translatable to an OpenCL 1D image buffer. It depends on the 1D texture size. However, a CUDA 2D texture is completely translatable to an OpenCL 2D image.

According to OpenCL 2.0[6], the maximum width for a 1D image buffer is increased (the exact amount depends on the hardware and OpenCL 2.0 is not supported by NVIDIA GPUs). We expect that CUDA 1D textures will be easily translated to OpenCL 1D image buffers in the near future.

OpenCL to CUDA. We support all OpenCL image-related functions, such as image creation, image read, image write, *etc.* We implement an OpenCL image object using a CUDA memory object. We define a class `CLImage` for the CUDA memory object. Its structure is shown in Figure 6. When the CUDA wrapper `clCreateImage()` is called, a CUDA memory object is created and pointed by the member `ptr` of the `CLImage` object. The CUDA memory object holds the contents of the image. `clCreateImage()` transfers the image contents to the CUDA memory object using `cudaMemcpy()`. An object with type `CLImage` is used as a CUDA kernel argument.

We also implement `clEnqueueWriteImage()` and `clEnqueueReadImage()` as CUDA wrappers. They perform reading and writing image data through the pointer member `ptr` of the `CLImage` object. In the kernel code, we use CUDA wrapper functions to implement OpenCL kernel built-in functions that are related to reading and writing images. These functions include `read_imageX()` and `write_imageX()` where `X` is either `i`, `ui`, or `f`. These wrappers handle the `CLImage` object to read or to write image data through pointer `ptr`.

In addition, the OpenCL built-in types `image_t, im-`



(a) Rodinia

(b) SNU NPB

(c) NVIDIA Toolkit

Figure 7: OpenCL to CUDA.

Table 2: System configurations

CPU	Intel Xeon E5-2650 x2
RAM	128GB DDR3 1333Mhz
GPUs used	NVIDIA GeForce GTX Titan AMD Radeon HD7970
NVIDIA CUDA Toolkit	7.0
AMD APP SDK	2.7
Host compiler	gcc-4.4.7

`age1d_t`, and `image2d_t` are defined as pointers to `CLImage` objects as shown in Figure 6 in the resulting CUDA application.

CUDA to OpenCL. A texture reference (variable) in CUDA is typically seen by both the host code and the device code. On the other hand, OpenCL does not allow such a reference (variable). To solve this problem, we pass the reference to the OpenCL kernel explicitly in the kernel parameter list. For a texture reference in a CUDA kernel, two kernel parameters are added in the target OpenCL kernel: an image object and a sampler object. Then, texture functions, such as `tex1Dfetch()`, `tex1D()`, `tex2D()`, and `tex3D()` in CUDA are translated to built-in `read_imageX()` functions in OpenCL, where `X` is one of `f`, `u`, and `ui`. For the inserted kernel parameters, appropriate `clSetKernelArg()` functions are inserted before launching the kernel in the OpenCL host code.

6. EVALUATION

In this section, we evaluate our approaches. We would like to show that the target application after translation achieves comparable performance to the translation source. This makes our framework a good rapid prototyping and performance evaluation tool. It can be used in the transition from one programming model to the other.

6.1 Methodology

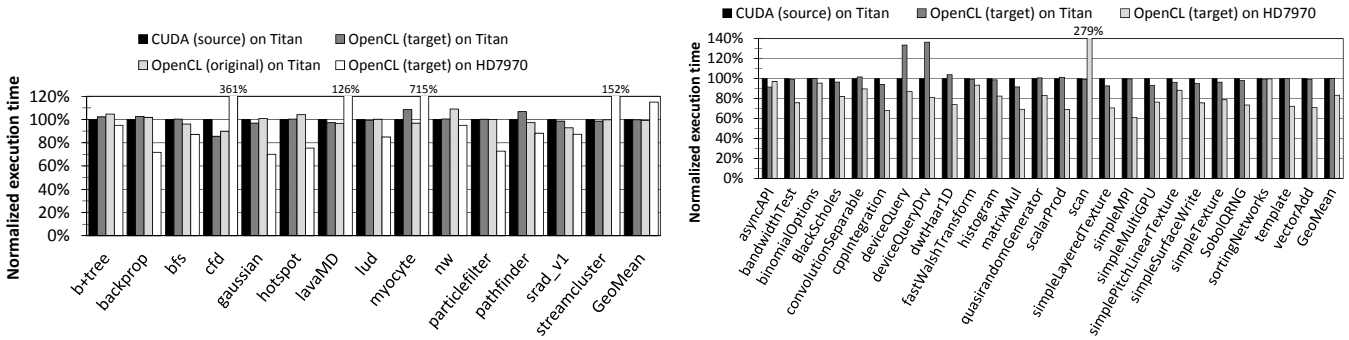
System configuration. The system configuration for the evaluation is described in Table 2. We use two Intel Xeon E5-2650 octa-core CPUs with 128GB DDR3 SDRAM. The GPU used is NVIDIA GeForce GTX Titan (labeled Titan from now on). Since NVIDIA supports only OpenCL 1.1 currently, we use OpenCL 1.1 and CUDA compute capability 3.5 for the evaluation. To see the portability of the translated OpenCL programs, we execute them on an AMD Radeon HD7970 (labeled HD7970 from now on).

Benchmark applications. Our evaluation is done with three benchmark suites: Rodinia 3.0[1], sample applications from NVIDIA CUDA Toolkit 4.2[16], and SNU NPB 1.0.3[18].

The Rodinia and NVIDIA Toolkit have both CUDA and OpenCL versions for each application, but SNU NPB does not have CUDA version. We chose sample applications from CUDA Toolkit 4.2 instead of the latest version because CUDA Toolkit 4.2 is the latest version having both CUDA and OpenCL sample applications. Since CUDA Toolkit 5.0, it does not provide any OpenCL sample applications.

6.2 OpenCL-to-CUDA Translation

Figure 7 shows the execution time difference between the original OpenCL version and the translated CUDA version for each application. We measure the total execution time of each program except the build time of the OpenCL application. CUDA does not require the on-line compilation of device code, but OpenCL does. Hence, the build time of



(a) CUDA to OpenCL (Rodinia)

(b) CUDA to OpenCL (NVIDIA Toolkit)

Figure 8: CUDA to OpenCL.

OpenCL should be excluded for a fair comparison. The time is normalized to the execution time of the original OpenCL version.

Rodinia. We successfully translated all 20 OpenCL applications in Rodinia to CUDA. The performance difference is about 3% on average for all applications.

The third bar in Figure 7 (a) indicates the performance of the CUDA code originally included in Rodinia for each application. For `hybridSort`, the performance difference is rather high (27%), compared to other applications. Others are less than 10%. This is attributed to the implementation difference between the CUDA version and the OpenCL version. The CUDA version of the application performs better since it requires less number of memory transfer between the host and the device. This implies that porting between two programming models is not always straightforward. If not carefully implemented, the ported program may lose performance.

SNU NPB. We successfully translated all 7 OpenCL applications in SNU NPB to CUDA. The performance difference is about 7% on average for all applications. In the case of FT, the resulting CUDA application takes only 57% of the execution time of the original OpenCL application. There are three kernels (`cffts1, cffts2`, and `cffts3`) whose execution time is significantly different across OpenCL and CUDA. They heavily use shared memory.

According to CUDA C Programming Guide[14], the shared memory provides two addressing modes, 32-bit and 64-bit, for CUDA compute capability 3.x. Note that the compute capability of Titan is 3.5. In the 32-bit mode, successive 32-bit (4-byte) words map to successive banks. In other words, a 64-bit (8-byte) word is stored in two successive memory banks. In 64-bit mode, successive 64-bit words map to successive memory banks. In other words, a 64-bit (8-byte) word is stored in a single bank. For example, in the 32-bit mode, a work-item accessing a variable of type `double` (8 bytes) has to bring data from two successive banks. In 64-bit mode, it has to bring data from a single bank. The number of banks in Titan is 32. It is the same as the number of work-items in a warp. We discover that OpenCL on Titan uses the 32-bit mode and CUDA on Titan uses the 64-bit mode.

The number of bank conflicts can be different for different addressing modes. If each work-item in a warp contiguously accesses an element from an array of `double` in the shared memory, it generates two-way bank conflicts in the 32-bit mode. It is because a work-item should be fed from two successive banks. In other words, a bank should provide data to two work-items. However, there is no bank conflict in the 64-bit mode because each work-item is fed by a different bank.

In FT, a work-item in the kernels copies an element of an array from the global memory to the shared memory. The element contains two variables of `double`. Then, it does some computation with the element in the shared memory. Finally, the result is copied back to the global memory. Because of the 32-bit addressing mode of the shared memory, the original OpenCL version generates two-way bank conflicts. This does not occur in the CUDA version. This is the reason why the original OpenCL version of FT is much slower than the translated CUDA version.

NVIDIA CUDA Toolkit. Our framework successfully translates all 27 OpenCL applications in the NVIDIA Toolkit samples. The performance difference is about 3% on average for all applications.

6.3 CUDA to OpenCL translation

Figure 8 shows the execution time difference between the original CUDA version and the translated OpenCL version for each application. We measure the total execution time of each application. The time is normalized to the execution time of the original CUDA version.

Rodinia. Figure 8 (a) shows the evaluation results of applications in Rodinia. Each application has four bars. Each bar indicates the original CUDA code executed on Titan, the target OpenCL code executed on Titan, the original OpenCL code executed on Titan, the target OpenCL code executed on HD7970, respectively. We use two different GPU architectures to demonstrate the portability of our translation framework. Note that HD7970 does not support CUDA.

Among 21 CUDA applications in Rodinia, all but seven applications are successfully translated to OpenCL. The seven applications contain untranslatable code that performs the following: passing pointers to a kernel function (`heartwall`), calling a built-in function for system parameters (*e.g.*, `cudaMemGetInfo()`) that cannot be implemented in OpenCL (`nn` and `mummgergpu`), using C++ classes in the device code (`dwt2d`), or using a one-dimensional texture whose size is larger than the maximum size of an 1D image in OpenCL (`kmeans, leukocyte,` and `hybridsort`).

For all applications, the performance difference between the original CUDA code and the translated OpenCL code

Table 3: Reasons of translation failures in NVIDIA Toolkit samples (CUDA to OpenCL)

Reason of translation failure	Applications
No corresponding functions	clock, concurrentKernels, simpleAssert, simpleAtomicIntrinsics, simpleVoteIntrinsics, FDTD3d
Unsupported libraries	convolutionFFT2D, lineOfSight, marchingCubes, particles, radixSortThrust
Unsupported language extensions	alignedTypes, convolutionTexture, dct8x8, dxtc, eigenvalues, Interval, mergeSort, MonteCarlo, MonteCarloMultiGPU, nbody, FunctionPointers, transpose, newdelete, reduction, simplePrintf, simpleTemplates, threadFenceReduction, HSOpticalFlow, simpleCubemapTexture
OpenGL binding	bilateralFilter, boxFilter, fluidsGL, imageDenoising, Mandelbrot, oceanFFT, postProcessGL, recursiveGaussian, simpleGL, simpleTexture3D, smokeParticles, SobelFilter, bicubicTexture, volumeRender, volumeFiltering
Use of PTX	matrixMulDrv, inlinePTX, ptxjit, matrixMulDynlinkJIT, simpleTextureDrv, threadMigration, vectorAddDrv
Use of unified virtual address space	simpleMultiCopy, simpleP2P, simpleStreams, simpleZeroCopy

on Titan is about 0.3% on average. Similarly, the performance difference between the translated OpenCL code and the original OpenCL code on Titan is about 0.2% on average for all applications. This implies two things. First, Rodinia provides original code that shows comparable performance between the OpenCL version and the CUDA version for each application. Second, the translated versions in OpenCL and CUDA also show comparable performance to the original OpenCL and CUDA versions, respectively. This indicates that our translator is good for rapid prototyping in transit between OpenCL and CUDA.

On the other hand, the target OpenCL applications on HD7970 show different performance behaviors because the underlying hardware and the OpenCL runtime are different from those of Titan. We emphasize that CUDA applications can run on HD7970 with our translation framework.

The application shows the largest performance difference in Rodinia is `cfd` (14%). It is because of the different occupancies of two versions (0.375 for CUDA and 0.469 for OpenCL). It is affected by the number of registers per work-item determined by the CUDA/OpenCL native compiler from NVIDIA.

NVIDIA CUDA Toolkit. 25/81 applications in NVIDIA CUDA Toolkit are successfully translated to OpenCL. NVIDIA CUDA Toolkit samples exploit many hardware-specific features of CUDA that are not supported by OpenCL. For example, while CUDA has built-in kernel functions, such as `assert`, `clock`, `__shfl`, and `__all`, which heavily depend on hardware features, OpenCL does not have any corresponding built-in functions or functionalities. There is no way to implement those functions in OpenCL without any hardware and native OpenCL compiler support. Specific reasons of translation failures are categorized in Table 3. All but unsupported libraries and the OpenGL binding come from the hardware-specific features in CUDA. All applications but four (`particles`, `Mandelbrot`, `nbody`, and `smokeParticles`) are failed due to multiple reasons. `particles` is failed due to unsupported libraries and OpenGL bindings while `Mandelbrot`, `nbody`, and `smokeParticles` are failed due to OpenGL bindings and C++ features.

The performance difference between CUDA and OpenCL on Titan is 0.2% on average for all applications. Two applications `deviceQuery` and `deviceQueryDrv` show significant performance degradation in the OpenCL version. They invoke CUDA API functions to get attributes of GPUs. These functions are implemented in wrappers that invoke OpenCL `clGetDeviceInfo()` many times to obtain those GPU attributes. For example, when an OpenCL wrapper `cudaGetDeviceProperties()` is called, the structure `cudaDevceProp` is filled by invoking `clGetDeviceInfo()` several times inside the wrapper.

Although previous studies[4, 8, 9, 10, 12] report that the translated OpenCL applications are slower than the original CUDA applications because of immaturity of NVIDIA's OpenCL framework, our evaluation shows comparable performance for most applications. Our evaluation indicates that NVIDIA's OpenCL framework has improved to the level of its CUDA framework.

Overall, the result shows that CUDA-to-OpenCL translation is relatively difficult compared to OpenCL-to-CUDA because CUDA applications have more hardware dependent features that cannot be implemented in OpenCL. The execution times of the source and target applications are comparable, on average, in both translation directions on the same hardware. We observe that the overhead of wrapper functions are negligible in our experiments.

7. CONCLUSION

In this paper, we analyze similarities and differences between OpenCL and CUDA and propose a bi-directional translation framework between them. Unlike previous translators that are solely based on static translation or partial manual translation, our translation framework is hybrid in the sense that it exploits both static translation and wrapper functions implemented in a runtime library. Our hybrid approach alleviates many difficulties of static translation, such as whole program analysis due to separate compilation. To the best of our knowledge, this paper is the first one that addresses handling textures and images in addition to the OpenCL-to-CUDA translation framework. We measure and compare the performance of original applications and translated applications from three benchmark suites: Rodinia, NVIDIA Toolkit samples, and SNU NPB. Most applications show comparable performance between the source and the target. The overhead of wrapper functions is negligible.

There are some inherent mismatches between OpenCL and CUDA. Hence, we cannot make a perfect automatic translation framework in both directions. However, we provide a much easier way to switch between the two programming models even if a programmer does not know one of them. We are planning to release the translation framework and its source to public. With our translator, we hope the separated CUDA and OpenCL communities can easily share their code base with each other.

8. REFERENCES

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

[2] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer*, 42(12):36–42, 2009.

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

[4] J. Fang, A. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225, 2011.

[5] M. Gardner, P. Sathre, W. chun Feng, and G. Martinez. Characterizing the challenges and evaluating the efficacy of a CUDA-to-OpenCL translator. *Parallel Computing*, 39(12):769 – 786, 2013. Programming models, systems software and tools for High-End Computing.

[6] K. Group. *OpenCL 2.0 Specification*. Khronos Group, November 2013. http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

[7] K. Group. *OpenCL 2.1 Specification*. Khronos Group, January 2015. https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf.

[8] M. Harvey and G. Fabritiisb. Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*, 182:1093–1099, April 2011.

[9] K. Karimi, N. G. Dickson, and F. Hamze. A Performance Comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, 2010.

[10] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[11] C. Lattner. clang: a C language family frontend for LLVM. Website, May 2007. http://clang.llvm.org.

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

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

[14] NVIDIA. *CUDA C Programming Guide*. NVIDIA, July 2013.

[15] NVIDIA. *CUDA Runtime API*. NVIDIA, July 2013.

[16] NVIDIA. *CUDA Samples Reference Manual*. NVIDIA, July 2013.

[17] P. Sathre, M. Gardner, and W. chun Feng. Lost in Translation: Challenges in Automating CUDA-to-OpenCL Translation. In *5th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, Pittsburgh, PA, September 2012.

[18] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148, 2011.

[19] J. A. Stratton, S. S. Stone, and W. mei W. Hwu. MCUDA: An Effective Implementation of CUDA Kernels for Multi-Core CPUs. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, pages 16–30, July 2008.

[20] top500.org. Top500 The List. Website, June 2015. http://top500.org/lists/.