國立臺灣大學電機資訊學院資訊工程學系
碩士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

移植計算語言於不同架構設備之開放計算語言與渲染腳本轉換器設計
O2render : A OpenCL-to-Renderscript Translator for Porting Across Various GPUs or CPUs

楊證諺
Yang Cheng-yan

指導教授：廖世偉 博士
Advisor: Liao Shih-wei, Ph.D.

中華民國 101 年 7 月
July, 2012

致謝

時光飛逝，碩士的兩年時間一轉眼就過去了。期間幫助過我的人太多了，謹此述以滿滿的感謝。

首先要感謝我的指導教授廖世偉老師。老師雖然平常工作繁重，但總是能從其中盡可能的撥出時間給予我們協助，並且適時地在我們遇到研究瓶頸的時候提供建議。除此之外，老師更是關心我們的生涯發展以及各種事務。對我而言，老師不只是指導教授，更像是工作上的夥伴、工作外的朋友。謝謝老師。

接著要感謝的是在碩一時給我許多幫助與指導的歐曜瑋及盧育龍兩位學長，在研究所學習起步的過程中，他們兩位不厭其煩的回答我問的問題，並且給予各種建議，讓我能在之後的研究過成中更加順利。

還要謝謝四位口試委員:楊佳玲教授、梁伯嵩博士、陳呈瑋博士與陳官辰博士能撥空參加我的口試，並且對於我的研究提出許多重要的建議。謝謝幾位委員及指導教授的指點，讓這篇論文更加完整。

最後感謝父母、家人的全力支持與鼓勵，讓我能在沒有其他生活壓力的情況下全力投注於研究之上；感謝實驗室的同學、學弟們，大家彼此之間的討論間學習到了很多東西，並且也讓課業外的研究所生活更加多采多姿；感謝各位好朋友，在我困頓時給我依靠、在我挫折時給我勉勵。

結束了十八年的學涯後，期許自己能為為我們的國家、社會貢獻一些微渺的力量。

中文摘要

現今世界中，數量極為龐大的Android設備可以說是一項最具影響力的系統。Google最近的Android版本中更是推出了Renderscript架構。

Renderscript為Android目前唯一官方的運算架構，提供了非常良好的效能與可攜性。但是如果要移植現有已經存在的其他運算架構例如OpenCL的程式到Android上，則必須要花費時間手動的改寫成Renderscript才可以享有相同的平行運算效果。

因此，我們提出了O2render系統讓現存的OpenCL程式可以自動的轉成Renderscript程式，藉此運行於Android系統之上。我們分析了OpenCL和Renderscript根本上的差異，並且藉由 low-level virtual machine(LLVM)的前端,Clang,來實作一個OpenCL到Renderscript的轉換器。

Abstract

400-million Android devices are arguably world‘s most impactful real-time multimedia systems. Google introduced Renderscript language and run-time in recent Android releases. Renderscript delivers performance and portability without losing usability. However, it is difficult to reuse software written in existing compute languages such as OpenCL. Thus, we develop the O2render system to enable OpenCL programs on Android devices. We analyze fundamental differences between OpenCL and Renderscript, and present our design of a translator between them using low-level virtual machine (LLVM). We extend LLVM’s frontend, Clang, and show that we achieve about the same performance in Renderscript with minimal translation overhead.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the development of *compute* languages and multicore processors becomes increasingly prevalent, software applications can achieve better performance via parallel computing. Parallelization on various types of computing units is referred to as heterogeneous computing. Open Computing Language (OpenCL) [7] is a heterogeneous computing framework that provides portability by defining an abstract execution model and a memory model. Apple proposed OpenCL and released its first implementation of OpenCL 1.0 in the Mac OS X Snow Leopard. OpenCL is now in version 1.2 and is popular among developers and vendors.

Smartphone-related technologies have advanced rapidly in recent years, partly thanking to the open-source Android project. Many developers and vendors have embraced the open-source Android platform and have contributed numerous software applications. However, Android did not have a heterogeneous computing framework such as OpenCL to allow developers to improve software performance until Google released Renderscript, a high-performance 3D rendering and heterogeneous computing framework for the Android platform. Because Renderscript [4] is Android's heterogeneous computing framework, if developers want to reuse existing source code that was originally programmed for other computing frameworks on Android, they must rewrite the code in Renderscript. Since Renderscript is relatively new then other heterogeneous computing framework, the number of programs that are written in it is currently small. Porting existing computing framework program to Renderscript is a way to speed up heterogeneous computing progressing on Android. Although OpenCL is a popular computing framework, porting an OpenCL application to a Renderscript one is difficult.

This thesis introduces O2render, an OpenCL to Renderscript translator that enables the porting of an OpenCL application to a Renderscript application. O2render automatically translates OpenCL kernels to a Renderscript script, or can create a skeleton of Renderscript script from an OpenCL kernel to enable developers to port OpenCL kernel more rapidly.

1.1 Contributions

This thesis makes the following contributions:

- We compare Renderscript and OpenCL and present insights into both computing languages.
- We develop a system to populate a new language's repertoire. Specifically, we enable code reuse and reduce the learning curve of a new computing language.
- We show that the resulting Renderscript code achieves about the same performance and the translation overhead is minimal.

1.2 Document organization

This thesis is organized as follows. Chapter 2 reviews previous research related to computing frameworks for the Android platform and translation between computing frameworks. Chapter 3 presents an overview of OpenCL and Renderscript and the differences between them. Chapter 4 considers issues related to translation from OpenCL to Renderscript. Chapter 5 explains the design and implementation of O2render. Chapter 6 evaluates O2render's perfromance. Finally, chapter 7 provides a conclusion and suggestions for future research.

Chapter 2

Related Works

Before Renderscript was developed, most Android developers used Android Native Development Tools (NDK) [6] to write performance-critical code. However, using NDK caused a high overhead for Java callback, had portability problems, and could not use parallel computing easily. Despite Android not supporting OpenCL, ZiiLABS enabled OpenCL on some ZiiLABS platforms and released the ZiiLABS OpenCL SDK [14].

There are three methodologies in translating between computing frameworks: (1) Add an abstract level library among frameworks; (2) Translate to a common intermediate representation (IR) and compile the IR during run-time; or (3) Translate a source code that uses a particular framework to a modified source code that uses a different framework. Below we will discuss each methodology in details through examples.

2.1 Add an abstract level among frameworks

Methodology (1) is employed by Swan [8], which is a tool that abstracts CUDA [12] and OpenCL using a higher-level runtime library. This library adapts to CUDA or OpenCL at lower level. Specifically, developers convert CUDA or OpenCL API calls into equivalent Swan calls, and the Swan library maps these calls to CUDA or OpenCL calls. In addition, Swan provides a Perl script that performs simple CUDA kernel code to OpenCL kernel code source-to-source translation using regular expression substitutions.

2.2 Translate to a common intermediate representation

An example of Methodology (2) is Ocelot [3]. The system enables CUDA applications to run on non-Nvidia GPU architecture. Ocelot operates in two main steps: It translates the CUDA Parallel Thread Execution (PTX) assembly to LLVM IR, and then applies an LLVM code generator to generate native code for the target platform. Ocelot allows CUDA programs to be executed at full speed on NVIDIA GPUs, AMD GPUs, and x86 CPUs without recompilation.

2.3 Translate the source code

Finally, CU2CL [11] is an example of a CUDA to OpenCL source-to-source translator, which follows Methodology (3). CU2CL is used as a plug-in for Clang. CU2CL begins translation using the preprocessor and parser of Clang [1] to obtain the abstract syntax tree (AST) of the input CUDA source code. Thereafter, CU2CL uses the "AST-Driven String-Based Translation" method to traverse the AST recursively. If an CU2CL node of interest is identified, CU2CL uses the Clang Rewriter library to rewrite the AST node. Finally, the translated OpenCL source code is generated based on the rewritten AST.

Chapter 3

Overview of OpenCL and Renderscript

OpenCL and Renderscript are designed to accelerate application performance by distributing computational workloads to multiple processing cores, such as CPU cores, GPU cores, and DSP cores, or a combination of these. However, OpenCL and Renderscript have several differences.

3.1 OpenCL

Before the release of OpenCL, Nvidia designed another computing (language), the Compute Unified Device Architecture (CUDA), for Nvidia GPUs, while supporting only Nvidia GPUs is the most notable drawback of CUDA. In order to design a computing (language) for various architectures, Apple Inc. in collaboration with Nvidia, AMD, IBM, and Intel initialized the OpenCL project and proposed it to Khronos Group. Although OpenCL was very similar to CUDA at the beginning because of the significant contribution from NVidia, it becomes more applicable to various architectures gradually. OpenCL defines an abstract layer specification for hardware related functions, which hardware vendors should follow by for implementing OpenCL supported hardware. Therefore, developers can speed-up their program performance by using OpenCL API without dealing with low-level hardware related function calls. Each OpenCL program comprises two components: kernel program and host program. The following will discuss them in more detail.

3.1.1 Kernel program

Developers have to write kernel code, in a variant of C99, to specify the detail execution of compute units. Fortunately, there are bunch of build-in functions which like integer handling, math and rational handling available for developer to use when writing kernel programs. But there are also kinds of more important functions that relates to ‘work-item’, the parallel unit of OpenCL. An OpenCL kernel, which is programmed in a variant of C99, specifies the functions to execute on OpenCL computing units.

3.1.2 Host program and memory model

An OpenCL host program is for setting execution environment for kernels. Figure 3.1 shows an example of typical execution flow of an OpenCL host program. In an OpenCL host program, developers have to set up the data to be processed first. After the the data are prepared, OpenCL kernels can then be loaded and compiled, and execute computational tasks. The communication between host and kernel device relies on OpenCL memory model. There are four kinds of memory region that can be access by kernels:

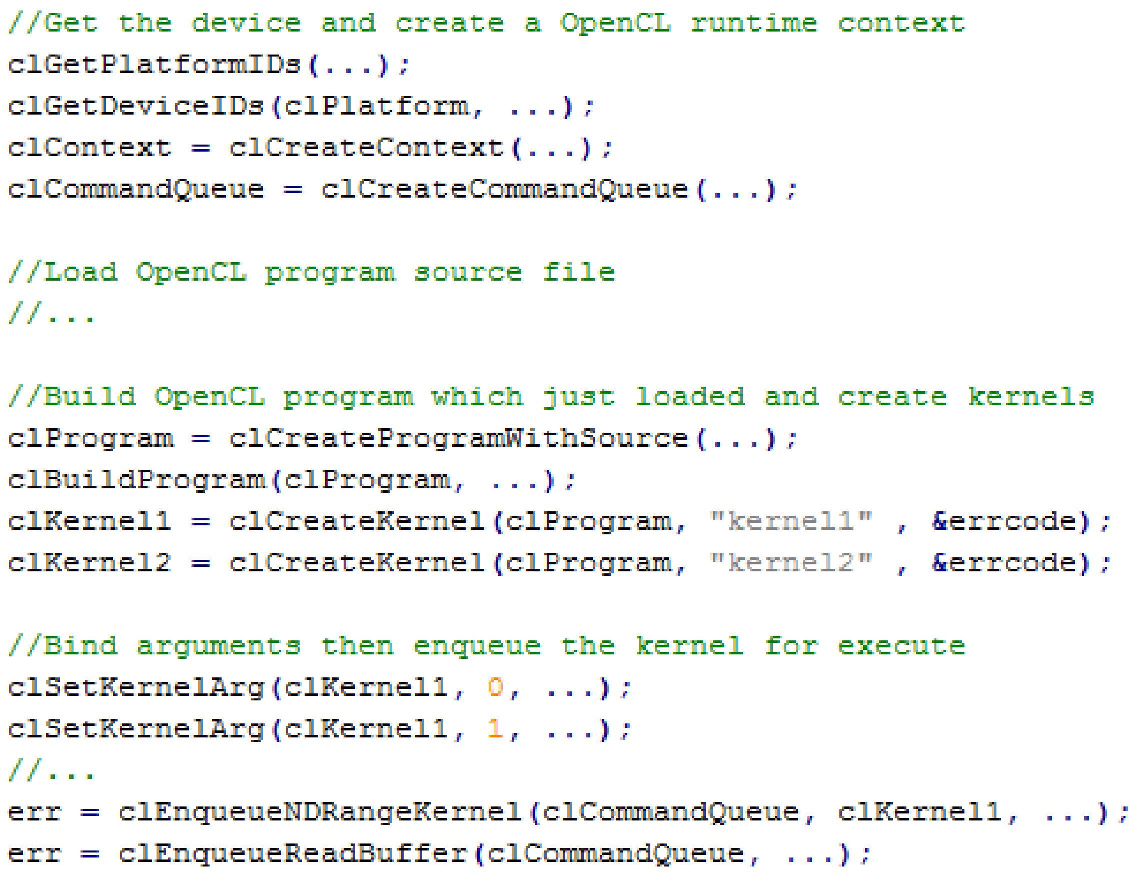
1. Global Memory : Read and write are permitted, all work-item in all work-groups are accessible.

2. Constant Memory : Only read is permitted, all work-item in all work-groups are accessible. Initialized and allocated in host program.

3. Local Memory : Read and write are permitted, only work-items in the same work-group are accessible.

4. Private Memory : Read and write are permitted, only can be accessed by the work-item which defines it.

OpenCL is designed to be portable across various devices. However, developers must often manually adjust parameters such as the size of the OpenCL work-group and the hardware-dependent factors in the OpenCL application to obtain better performance [9].



```
//Get the device and create a OpenCL runtime context
clGetPlatformIDs(...);
clGetDeviceIDs(clPlatform, ...);
clContext = clCreateContext(...);
clCommandQueue = clCreateCommandQueue(...);

//Load OpenCL program source file
//...

//Build OpenCL program which just loaded and create kernels
clProgram = clCreateProgramWithSource(...);
clBuildProgram(clProgram, ...);
clKernel1 = clCreateKernel(clProgram, "kernel1" , &errcode);
clKernel2 = clCreateKernel(clProgram, "kernel2" , &errcode);

//Bind arguments then enqueue the kernel for execute
clSetKernelArg(clKernel1, 0, ...);
clSetKernelArg(clKernel1, 1, ...);
//...
err = clEnqueueNDRangeKernel(clCommandQueue, clKernel1, ...);
err = clEnqueueReadBuffer(clCommandQueue, ...);
```

Figure 3.1: Example of typical host program execution flow

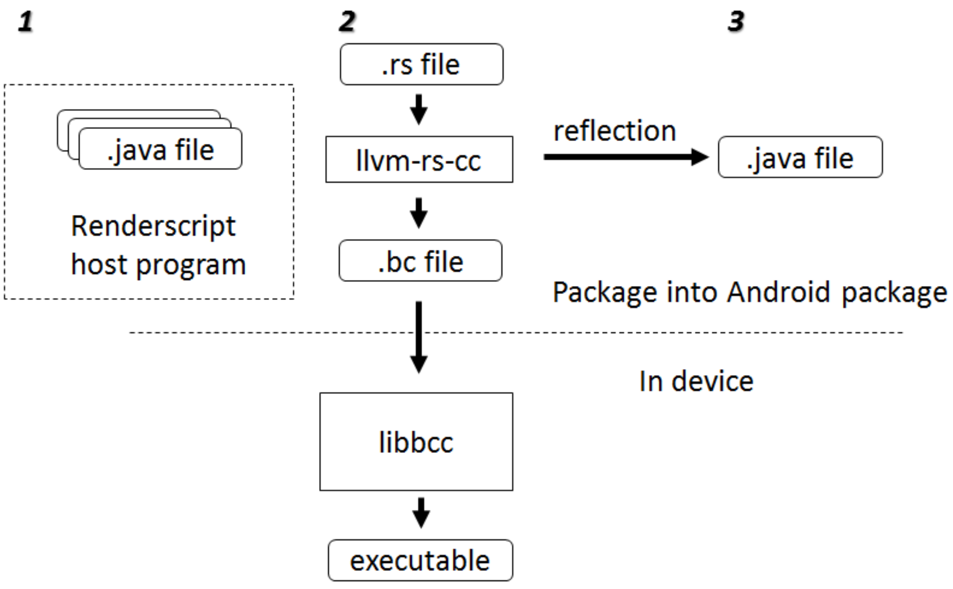


Figure 3.2: (a) 3 parts of a Renderscript application (b) Compilation flow of a Renderscript script

3.2 Renderscript

Google released Renderscript as an official computing framework of Android in 2011 [2]. Renderscript provides high performance and portability across hardware architectures to Renderscript applications. Although Renderscript provides API for both graphics and computation, this thesis focuses on computation.

A Renderscript application consists of three parts: 1) a Renderscript host program written in Dalvik, denoted as ".java file" in Figure 1, 2) one or more Renderscript kernels, and 3) reflected classes in Dalvik. These three parts are denoted as *1*, *2*, and *3* respectively in Figure 1. The Renderscript kernel, also written in a variant of C99, has a special function called root() that identifies the pieces of code for parallel execution in the same manner that an OpenCL kernel does. The Renderscript host program is similar to other Android applications but further initializes the use of the Renderscript kernel. The reflected Java classes contain functions enabling the Renderscript host program to communicate with the kernels during Renderscript runtime, allowing functions such as memory binding between the host program and the kernels.

Renderscript achieves portability using LLVM [10] technologies. Figure 3.2 shows the compilation flow for the Renderscript kernels. First, the Renderscript compiler "llvm-rs-cc", a derivative of the Clang project, compiles Renderscript kernels into LLVM-based bitcode (IR) files with a .bc filename extension. During compilation, llvm-rs-cc also generates the corresponding reflected Java classes of the kernels. Thereafter, the bitcode is packed into an Android application package (.apk file) of the Renderscript application, and the application is installed on an Android device. When the application is about to execute, the bitcode is compiled to the appropriate machine code for the device using libbcc, the back-end compiler of the Renderscript kernels.

3.3 Differences between OpenCL and Renderscript execution models

The differences between OpenCl and Renderscript is not only in API or code appearance. OpenCL tends to provide more low level controls to developers so that more information and operability are available. But on the other side, Renderscript is trying to offer a more intuitive design, which minimize learning curve and complexity of the program. Therefore compared to OpenCL, there are some restriction to Renderscript as the tradeoff. Consequently, the translation is accompanied by some issues.

One major difference between OpenCL and Renderscript is their respective execution models. Figure 3.3 shows how data are partitioned for parallel processing in OpenCL. A work-item is a single kernel execution with a set of data, a work-group is a set of work-items that access the same processing resource, and an NDRange describes the space of the work-items. Developers have to define how an OpenCL application partitions input data and executes work-items. In contrast, Renderscript data to be parallel-processed are first transferred to a Renderscript type "Allocation". When the special function "rsForEach()" is called in the kernel, or "ForEach_root()" is called in the Renderscript host program, the Renderscript engine can help partition the Allocation for each execution of the root(), distributing each execution to the processing cores.

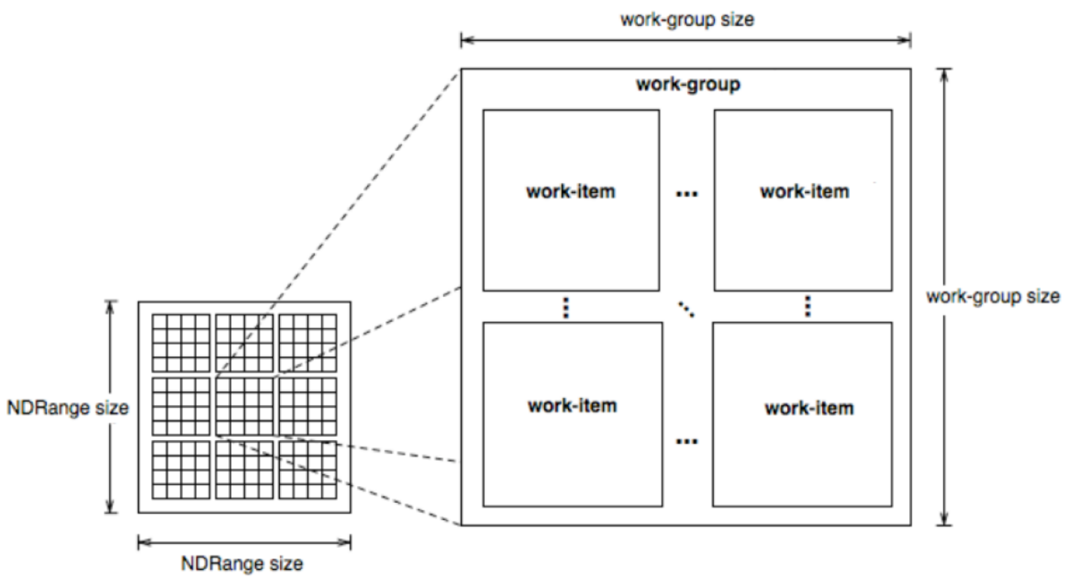


Figure 3.3: An example of OpenCL work-items and work-group

Chapter 4

Issues in translating from OpenCL to Renderscript

Although the languages of OpenCL and Renderscript are derived from C99, O2render cannot translate them using string substitution alone. This chapter describes how O2render achieves the translation from the aspect of implementation.

4.1 Removing unnecessary OpenCL-only qualifiers

The following qualifiers are not necessary to Renderscript kernels and O2render removes them:

1. Address space qualifiers, including __global/global, __local/local, __constant/constant, and __private/private.
2. Function qualifiers, including __kernel/kernel.
3. Access qualifiers, including __read_only/read_only, __write_only/write_only, and __read_write/read_write.

4.2 Generating a Renderscript kernel file for each OpenCL kernel

An OpenCL kernel script file may contain more than one kernel function for parallel execution, and each kernel function has a distinct function name. However, each Renderscript kernel script file should only have one function for parallel execution and the function name should be "root". Therefore, O2render generates a Renderscript script file for each OpenCL kernel function and renames the OpenCL kernel function as "root".

4.3 Translating OpenCL API calls into Renderscript ones

In addition, O2render translates OpenCL API calls in an input source code to corresponding Renderscript API calls. Although some APIs, such as mathematical ones, are similar in OpenCL and Renderscript, others are different. O2render translates these types of OpenCL API calls into Renderscript API calls that are semantically equivalent. Below we show a translation for accessing image pixel data. The program of reading image:

```
// OpenCL style of accessing image pixel data
read_imagef(srcImg, sampler, (int2)(x, y));
```

will correspond to the following translation:

```
// Renderscript style of accessing image pixel data
rsUnpackColore8888(*(uchar4 *)rsGetElementAt(srcImg, x, y));
```

4.4 Modifying the arguments of kernels

An OpenCL kernel function can have an arbitrary type and number of input arguments. However, arguments for a Renderscript root() should follow these rules: a. The first argument is optional and points to an input "Allocation" for the root() function; b. The next argument indicates an output "Allocation" that stores the result returned to the Renderscript host program; c. Several optional "uint32_t" type arguments for indexing can be placed after the output “Allocation” argument.

The number of input arguments in an OpenCL kernel is typically more than one. A Renderscript root() can have a maximum of one argument for input. O2render resolves these issues by translating every arguments of an OpenCL kernel to global variables in the translated Renderscript script. If an argument of an OpenCL kernel refers to a region of memory, O2render translate its data type to “rs_allocation”, or O2render keeps its data type in the translated Renderscript script. Thereafter, the Renderscript host program transfers input data to the root() through the reflected binding function of the global variables.

In OpenCL, developers typically use the “get_global_id” function to obtain the global work-item ID value. Because the total work-item size is specified in the host program and usually matches the input data array size, developers typically use this global work-item ID as a data array index for accessing target data.

Occasionally, developers may use the global work-item ID for another purpose, such as an integer for computation. Although Renderscript does not have a work-item ID, the usage of this work-item ID is similar to the index information that is passed into the root() function. Therefore, O2render maps the “get_global_id” function to the arguments for indexing, and uses the “rsGetElementAt()” function to access the data that the arguments point to in an “rs_allocation.”

A sample of the declaration of a translated root() is:

```
void root(float *result, uint32_t O2Rx, uint32_t O2Ry);
```

4.5 Translating OpenCL kernel function with multiple outputs

For most OpenCL kernel function, there are only one output parameter. So we just put it at the output parameter position in Renderscript root function after translation. Then we let Renderscript system to(DELETE THIS) distribute which data cell in the allocation is being handled. This method is intuitive and meets the design of root function.

But in the case of translating OpenCL kernel function of multiple output parameters, similar to the inputs, we have to generate additional global variables for them. Then developers can generate and bind the output memory allocation to these variables in Renderscript host program via reflected function. Compared to use the output parameter in root(), we have to specify which data cell of the output allocation is being assigned by index information that passed to the root().

4.6 Putting it altogether

We will use a kernel in the common GrayScale Filter to illustrate the task of O2render. Given the kernel *sample* below:

```
float3 gMonoMult = (float3)
                   (0.299f, 0.587f, 0.114f);
__kernel void sample(sampler_t sampler,
                     __read_only image2d_t srcImg,
                     __write_only image2d_t dstImg)
{
 float4 f4 = read_imagef(srcImg,
                         sampler,
                         (int2)
                         (get_global_id(0),
                         get_global_id(1)));
 float4 Mono = (float4){0.0f, 0.0f, 0.0f, 0.0f};
 Mono.xyz = dot(f4.xyz,gMonoMult);
 write_imagef(dstImg,
             (int2)
             (get_global_id(0), get_global_id(1)),
             Mono);
}
```

O2render should generate the following Renderscript code:

```
#pragma version(1)
#pragma rs java_package_name(...)
```

```
float3 gMonoMult = (float3){ 0.299, 0.587, 0.114 };
rs_allocation sampler;
rs_allocation srcImg;
void root(uchar4 *dstImg,
          uint32_t O2Rx,
          uint32_t O2Ry)
{
 float4 f4 =
 rsUnpackColor8888(*(uchar4 *)
                   rsGetElementAt(
                                  O2RenderIn->srcImg,
                                  O2Rx,
                                  O2Ry));
 float4 Mono = (float4){0.0f, 0.0f, 0.0f, 0.0f};
 Mono = dot(f4.xyz,gMonoMult);
 *dstImg = rsPackColorTo8888(Mono);
}
```

Chapter 5

Design and implementation of O2render

O2render is implemented as a Clang plugin. This method adds extra steps to the original Clang compilation. This chapter describes how O2render benefits from using Clang and provides the implementation details for O2render.

5.1 Implement O2render as a Clang plugin

O2render is a source-to-source compiler that operates at the AST level. Parsing input source code to generate AST is the most essential task of many compilers. Clang, a subproject of LLVM and a front-end compiler for the C language family, provides the necessary components for compiling source code to LLVM IR. The latest version of Clang supports compiling OpenCL code. Furthermore, Clang provides libraries for developers to add their own requirements easily in the compilation. Thus, we chose to leverage Clang to design and implement O2render. O2render uses Clang to parse and generate AST, focusing on the manipulation of the generated AST using a subclass of Clang "AST-Consumer." O2render also uses Clang Rewriter, a Clang library, to perform string-based rewrite that inserts, removes, and replaces text to specific ranges of the input source code. The modified result is stored in another buffer, so that the original source and the AST are not altered.

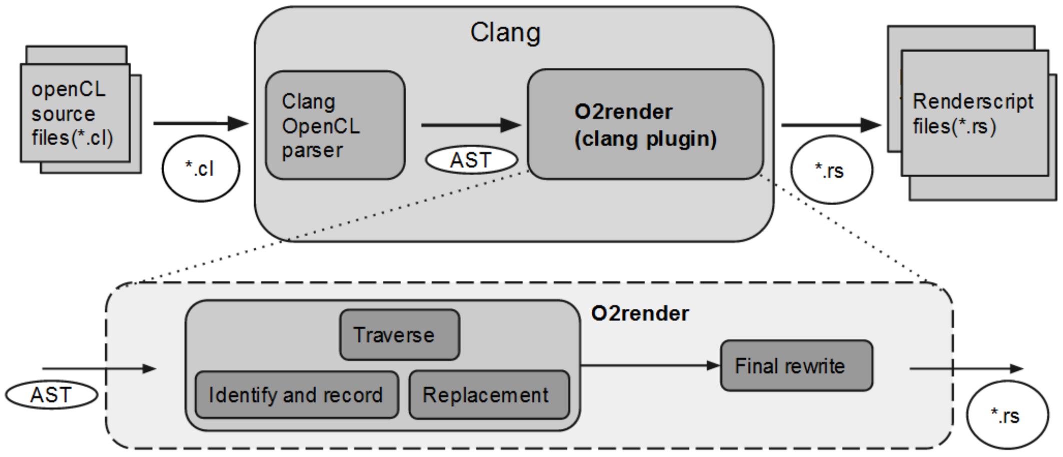


Figure 5.1: High level overview of O2render

5.2 Insight into O2render

Furthermore, O2render uses a two-phase-rewrite strategy. Specifically, O2render can rewrite during traversing or after traversing.

Figure 5.1 shows the high level overview of O2render. Here O2render takes AST which is generated by Clang OpenCL parser as input, then outputs Renderscript files. The flow of O2render has four steps: (1) traversing the AST; (2) identifying and recording AST nodes that O2render is interested in during traversing; (3) rewriting nodes that O2render is interested in; and (4) rewriting the whole script based on the recorded information. The first three steps are recursively executed until traversing of the AST is finished. We refer to these three steps as Phase 1. Thereafter, the fourth step, denoted as Phase 2, is executed. Figure 5.2 shows the driver for the first three steps algorithmically. Note that "IsInterest" in the figure checks whether the node is needed for further process or not. An example of such node is the kernel function declaration in OpenCL. "NeedRecord" checks if the node is need to be recorded for the rewrite in Phase 2.

Because O2render translates an OpenCL kernel function into a Renderscript script, a kernel function declaration in the AST is a basic translation unit. O2render begins translation by recording every argument in the kernel function declaration for the second phase rewrite. Thereafter, it begins recursively traversing the AST from the root of the kernel function declaration. When an AST node is found during recursive traversing, O2render examines its type and value to decide whether it is an interesting node. If so,

```
procedure RewriteExpression(expr)
  for all subexpr of expr do
    RewriteExpression(subexpr)
  end for
  if IsInterest(expr) then
    if NeedRecord(expr) then
      Record(expr)
    end if
    if IsPhaseOneRewriteExpr(expr) then
      source = GetRewrittenSourceCode(expr)
      newSource = DoPhaseOneRewrite(source)
      ReplaceSourceRange(expr, newSource)
    end if
  end if
end procedure
```

Figure 5.2: Algorithm of traversing AST in Phase 1

O2render determines whether the node is suitable for immediate rewrite or if O2render should record the node and leave the rewrite to be performed during the second phase after the rewrite has finished the recursive traversing.

After recursively traversing the AST, O2render executes the second-phase rewrite that alters the structure of the AST. The second-phase rewrite obtains the required information during traversing and performs the following tasks: (1) Adding the “pragma” directives required by Renderscript; (2) writing the global variables into the script; and (3) rewriting functions that require additional changes. The function-rewriting example is illustrated as follows.

```
// OpenCL
write_imagef(dstImg, (int2)(x,y), Mono);


// Renderscript
*dstImg = rsPackColorTo8888(Mono);
```

Table 5.1: An Example of O2render Table

OpenCL	Renderscript
read_imagef	rsUnpackColor8888(*(uchar4 *)rsGetElementAt(%s,x,y))
write_imagef	*%s = rsPackColorTo8888(%s)

5.3 Mapping table

O2render has two types of mapping tables for rewrite. The first type records an OpenCL token string to a Renderscript string pairs that the OpenCL token string can be simply replaced by the mapped Renderscript string when rewrite. The other type of table is used for the type of OpenCL token string that cannot be replaced, but instead must be translated into a semantically equivalent string in Renderscript. This latter type should record more information than the string-to-string method, including how arguments are adjusted. Table 5.1 shows an example of this type.

Chapter 6

Evaluation

Because Renderscript applications run only on Android, and Android does not support OpenCL, comparing the execution performance of an OpenCL application with a Renderscript application that had been translated using O2render is difficult.

Because computational photography is popular for mobile devices and can benefit from heterogeneous computing, this section examines the correctness and the execution performance of the translated Renderscript application for computational photography.

Table 6.1 shows the translation time of cases in this section. The experiment are on the desktop with Intel Core 2 Duo CPU E6550 2.23GHz and 3.4 GB Memory.

6.1 Grayscale Filter

We implement an OpenCL application (OpenCL Grayscale Filter) used to convert color images to grayscale images by calculating the scalar product of each pixel using a constant vector. Google provides a Renderscript application (RS Grayscale Filter) [5] that is a functional equivalent to the OpenCL Grayscale Filter. We translate this OpenCL Grayscale Filter into a Renderscript application using O2render and compare its perfor-

Table 6.1: Translation time

Program	Translation Time (millisecond)
Grayscale Filter	24
Saturation/Gamma Adjuster	38

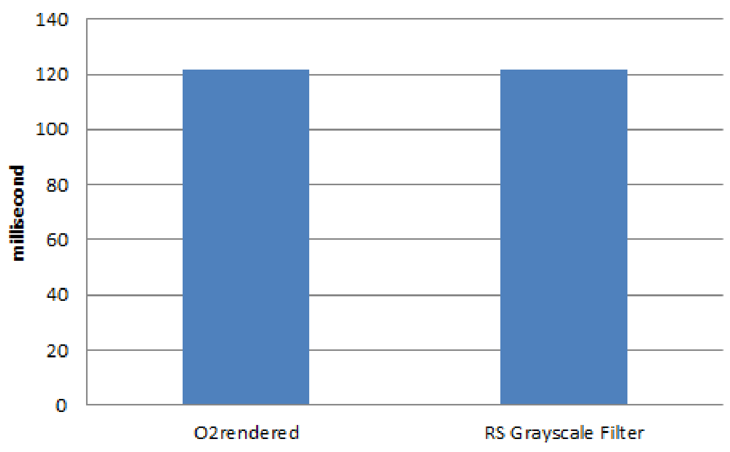


Figure 6.1: O2rendered Grayscale Filter vs. RS Grayscale Filter on Android devices

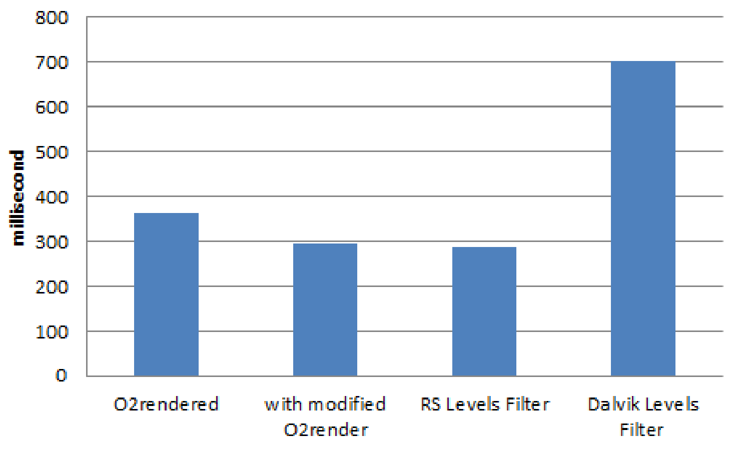


Figure 6.2: O2rendered Saturation/Gamma Adjuster vs. Using new O2render vs. RS Levels Filter vs. Dalvik Levels Filter

Figure 6.3: Input image for O2rendered Grayscale Filter

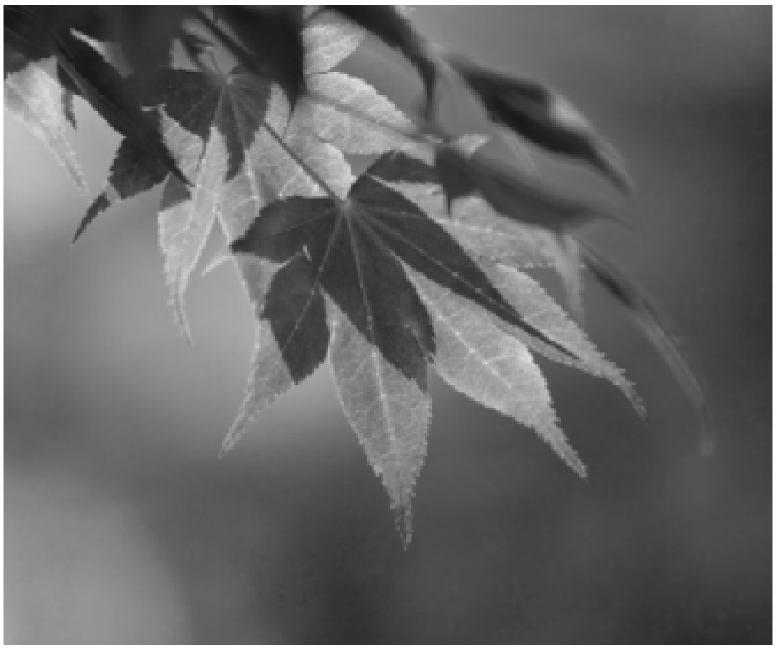


Figure 6.4: Output image for O2rendered Grayscale Filter

mance to RS Grayscale Filter.

The performance of the translated OpenCL Grayscale Filter is similar to that of RS Grayscale Filter, despite their coding being different. In addition, we execute both applications using the same input image, measure the execution time, and calculate the averages in milliseconds. Figure 6.1 shows that the execution performance of both applications is similar. Figure 6.3 and Figure 6.4 show the input and output images from O2rendered Grayscale Filter. Note that the translation time for an 18-line kernel in Grayscale Filter is less than one second, which attests to the efficiency of O2render.

6.2 Saturation/Gamma Adjuster

We implement an additional OpenCL application (OpenCL Saturation/Gamma Adjuster) that adjusts the saturation and gamma levels of an input image by multiplying each pixel using a color matrix and calculating a power value for each sub-pixel. Google has several similar applications, including ones that use Renderscript and Android Java (Renderscript Levels and Dalvik Levels, respectively) [13]. We translate the OpenCL Saturation/Gamma Adjuster to a Renderscript application using O2render and compared its performance to Renderscript Levels and Dalvik Levels.Figure 6.2 shows the average ex-

Figure 6.5: Input image for Renderscript Levels

ecution performance of the three applications, as well as a manually modified and translated OpenCL Saturation/Gamma Adjuster. The translated OpenCL Saturation/Gamma Adjuster is approximately 60 ms slower than is Renderscript Levels. This is because Renderscript provides better-performing APIs than does OpenCL, and the old O2render cannot map the lines of this type of OpenCL code into this type of Renderscript API. Specifically, the OpenCL Saturation/Gamma Adjuster uses slower read or write calls to convert images, but the Renderscript Levels calls the "convert_float4" or "convert_uchar3" (dedicated Renderscript APIs that are hardware accelerated) to achieve a better performance. However, the translated OpenCL Saturation/Gamma Adjuster still performs better than does the Dalvik Levels.

Even when O2render does not generate the highest performing Renderscript code, developers can use the translated code as a template and can optimize it manually. These results also shows that with less than 10 lines of modification, the new version of the translated OpenCL Saturation/Gamma Adjuster performed as well as the Renderscript Levels. We have enhanced O2render to leverage dedicated Renderscript APIs using improved O2render table. As a result, Figure 6.2 shows that this new O2render delivers about the same performance as Renderscript Levels does. Figure 6.5 and Figure 6.6 present the input and output images from Saturation/Gamma Adjuster. Both the old and new O2render translate the Adjuster within a second. We find that the efficiency is about producing every two lines of Renderscript code in one milli-second.

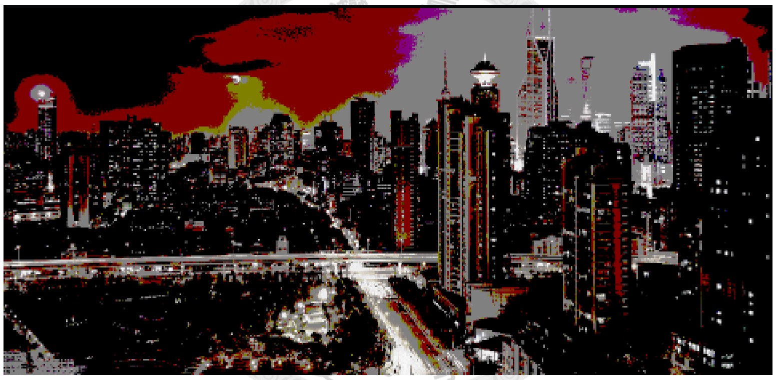


Figure 6.6: Output image for Renderscript Levels

Chapter 7

Conclusion and Future Works

7.1 Conclusion

This thesis begins by discussing the need to port an existing OpenCL application to a Renderscript one and the differences between the two frameworks. We create and implement O2render, an OpenCL to Renderscript translator that leverages the Clang compiler to simplify porting from OpenCL to Renderscript.

Almost all Android devices are equipped with cameras today, and photography application is a popular category of mobile application. Using heterogeneous computing technologies is an excellent method of improving the performance of computational photography. By using O2render to leverage existing OpenCL-based computational photography application, developers can create high-performance mobile photography applications more easily.

7.2 Future works

In the future, we will add more specific translation and mapping abilities to O2render. An ongoing project is to enable O2render to translate from Renderscript script back to OpenCL kernel, making it a bidirectional translator between OpenCL and Renderscript. To achieve such goal, O2render will leverage our defined API mapping tables and the “llvm-rs-cc” program to translate Renderscript to OpenCL. Furthermore, O2render has the potential to support other heterogeneous computing frameworks such as CUDA, because it is similar to OpenCL and is experimentally supported by Clang already.

Bibliography

[1] Clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[2] Renderscript is the official compute API for Android. `https://groups.google.com/forum/#!topic/android-platform/AhYDoDjMlb4`.

[3] G. Diamos. The design and implementation ocelot ' s dynamic binary translator from ptx to multi-core x86. *Computer Engineering*, 9:22–39, 2009.

[4] Google. Renderscript. `http://developer.android.com/guide/topics/renderscript/index.html`.

[5] Google. Renderscript Compute. `http://developer.android.com/guide/topics/renderscript/compute.html`.

[6] Google. Android ndk document, 2012.

[7] K. Group. OpenCL. `http://www.khronos.org/opencl/`.

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

[9] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating performance and portability of opencl programs. *Science And Technology*, 2:52, 2010.

[10] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization 2004 CGO 2004*, 53706(c):75–86, 2004.

[11] 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, December 2011.

[12] NVIDIA. CUDA Toolkit. `http://developer.nvidia.com/cuda-toolkit`.

[13] R. J. Sams. Levels in Renderscript. http://android-developers.blogspot.tw/2012/01/levels-in-renderscript.html.

[14] ZiiLABS. ZiiLABS OpenCL. http://www.ziilabs.com/products/software/opencl.php.