

國立臺灣大學電機資訊學院資訊工程學系

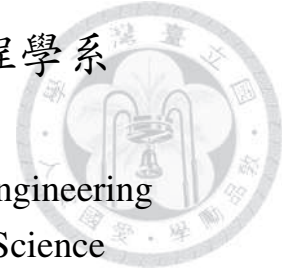
碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis



智慧型行動裝置OpenCL運算移轉框架

OpenCL Computation Offloading Framework on Mobile
Device

洪偉書

Wei-Shu Hung

指導教授：劉邦鋒 博士

Advisor: Pangfeng Liu, Ph.D.

共同指導教授：吳真貞 博士

Co-Advisor: Jan-Jan Wu, Ph.D.

中華民國103年 6月

June 2014



誌謝

感謝劉邦鋒老師及吳真貞老師的指導，在碩士這兩年，老師所傳授之專業知識、研究態度及研究方法都讓我受益良多，對我一輩子有著莫大地助益。

感謝實驗室的學長們，從碩一進來一路在學長領導的計畫下學習，這些研究經驗，對我完成這份論文幫助非常大。也感謝實驗室的同學們，能在研究上遇到困難時，不吝教導我各種解決方法。

最後，感謝家人一直以來的照顧，以及女友的陪伴，謝謝你們的支持、鼓勵，讓我能夠順利完成碩士學業。



摘要

現今，行動裝置仍然受限於有算的運算資源。爲了增進行動裝置的性能，許多研究將行動裝置上的運算移轉到運算能力更強的電腦上，然而，大多數的方法只能將運算移轉至CPU上。近來，GPU得到了科學界的重視，事實上，GPU出色的平行計算能力能夠爲許多不同種類的應用帶來加速。

在這篇論文中，我們爲行動裝置提出了一個基於OpenCL的運算移轉框架，這個框架能夠在使用者不知情的情況下將行動裝置上的OpenCL 運算移轉到可使用且OpenCL相容的計算設備上，我們將系統框架實作於真實的機器上，並且以矩陣計算、影像處理及AMD的標竿分析程式來測試我們的系統。透過將運算移轉至遠端機器，程式最多可達到50.3倍的加速。

關鍵字 OpenCL、GPU、GPGPU、行動雲端運算、普及運算、運算移轉



Abstract

Nowadays, mobile devices are suffering from limited computational resource. To increase capabilities of mobile devices, many efforts have been made to offload computation from mobile devices to resourceful servers. However, most of the approaches are only capable of offloading computation to CPUs. Recently, GPUs have received a lot of attention from the scientific community. Indeed, the exceptional parallel computing capabilities of GPUs can be used to accelerate different types of applications.

In this thesis, we propose a computation offloading framework based on OpenCL – a standard for GPU computing. Our framework transparently offloads OpenCL workloads from mobile devices to an available OpenCL compatible device. We deployed our framework on real machines and conducted evaluation experiments using various OpenCL programs including basic matrix computations, an image processing Android app and benchmarks from AMD. The program achieves up to 50.3X speedup by remote offloading compared to the local execution using CPU.

Keywords OpenCL, GPU, GPGPU, Mobile Cloud Computing, Pervasive Computing, Computation Offloading



Contents

Certificate	i
Acknowledgement	ii
Chinese Abstract	iii
Abstract	iv
1 Introduction	1
2 Related Work	3
2.1 Computation Offloading Research for Mobile Device	3
2.2 OpenCL	5
2.2.1 Platform Model	5
2.2.2 Execution Model	5
3 System Architecture	7
3.1 System Components	8
3.1.1 OpenCL Client	8
3.1.2 OpenCL Server	9
3.1.3 OpenCL Manager	9
3.2 System Flow	11
4 Implementation	12
4.1 OpenCL Client	12
4.2 OpenCL Server	14

4.3	OpenCL Manager	14
5	Experiments	16
5.1	Experimental Environment	16
5.2	Experiment Result	17
6	Conclusion	21
	Bibliography	22





List of Figures

2.1	The OpenCL platform model [16]	6
3.1	OpenCL computation offloading framework	7
3.2	System flow	11
5.1	Matrix multiplication execution time	18
5.2	Matrix transpose execution time	18
5.3	Bilateral filter execution time	19
5.4	Black scholes execution time	19
5.5	Bitonic sort execution time	20



List of Tables

3.1	The historical performance record	10
5.1	Server setup	16
5.2	Network status	16



List of Listings

3.1 The vecAdd kernel source	10
--	----



Chapter 1

Introduction

Mobile devices such as smartphone and tablets have become increasingly popular. Recent study [5] shows that smartphone penetration rate reached 27% in 2013, and will exceed 60% in 2019. Developers worldwide have built different types of mobile apps which enables mobile devices to provide wide range of functionalities and enriches users' mobile computing experience. The constant evolution of mobile devices results in an increasing user expectation and a growing demand for more sophisticated apps such as 3D video gaming, real-time media processing and augmented reality. However, most mobile devices suffer from limited resource (e.g., battery capacity, processor capability and memory size), which makes them perform poorly on some resource-intensive applications.

To address this issue, considerable research has been conducted on methods to offload computation from mobile device to resourceful computers (e.g., cloud server or nearby computer). These approaches increases mobile devices' capabilities and improves the performance of mobile apps. Nevertheless, this emerging approach faces several challenges.

Nowadays, computers equipped with graphics processing unit (GPU) are very common. However, most of the approaches [13, 14, 18, 20] are only capable of offloading computation to CPU. GPUs are a high performance computational hardware that are originally designed for graphics rendering. Recently, developers have used GPU to accelerate general-purpose applications such as game physics, bioinformatics, molecular dynamics, and computational finance [19, 21]. Che et al. [12]'s work shows the GPU program achieves 72X speedup compared with CPU implementation. In addition, manufacturers have developed programming model to assist development of GPU program (e.g., Nvidia CUDA [4]). Amazon EC2 [11] also has begun to provide high-performance computing (HPC) service using GPUs. As GPU technology advances, how to offload workloads to GPU has become an important issue.

To address the issues, our work utilizes OpenCL [24] – a parallel programming standard for writing programs that execute across heterogeneous compute devices including CPUs, GPUs, and other processors. With OpenCL, developers can build mobile applications and accelerate them

on GPU. Moreover, OpenCL shrinks hardware heterogeneity [22] by providing a unified programming environment including a set of APIs to control heterogeneous compute devices and a runtime system to execute OpenCL programs. Such unified environment enables developers to standardize their programs and to ensure portability on any computers that supports OpenCL. Our work utilized this “write once, run anywhere” functionality to offload workloads to computers with different hardware architectures (e.g., smartphones, laptops, desktop PCs and workstations).

In this thesis, we propose an OpenCL computation offloading framework for mobile device. It adopts a centralized mechanism that manages multiple OpenCL compatible devices and transparently offloads computation from mobile devices to them. This mechanism relies on a dynamic allocation method targeting to reduce the execution time of users’ programs. Also, when our framework detects remote offloading cannot bring benefits due to insufficient network speed or servers’ capabilities, it directly runs the workload on user’s mobile device. Additionally, our system is implemented by transparently replacing the original OpenCL API implementation. Therefore, developers simply use the standard OpenCL API to implement OpenCL programs that can be executed on our framework.

The rest of the paper is organized as follows. Chapter 2 presents the related works in mobile computation offloading and introduction of OpenCL. Chapter 3 gives an overview of our framework architecture. Chapter 4 describes the details of our system implementation. Chapter 5 discusses the results of experiments. Finally, we conclude in Chapter 6.



Chapter 2

Related Work

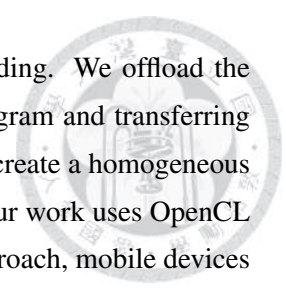
In this chapter, we describe related works in mobile computation offloading research. In addition, we introduce basic knowledge of OpenCL about its platform and execution model.

2.1 Computation Offloading Research for Mobile Device

The mobile computation offloading research has been conducted for more than a decade. In 2001, Satyanarayanan proposed an remote execution approach called “cyber foraging” [23] which dynamically augments the computing resources of a wireless mobile computer by exploiting wired hardware infrastructure. They believe, in the foreseeable future, stationary compute server or data staging server in public space will provide intensive computation service for mobile devices in vicinity. We adopt a similar approach. In our framework, we have a stationary computer that manages a set of OpenCL compute devices and allocates compute devices to users. The difference relays in the fact that, in our framework, this stationary computer does not provide computational service. Its function is to forward the workload to the compute devices it manages.

In recent years, the availability of high-capacity networks, low-cost computers and storage devices has bred a new research area called “cloud computing”. It offers an on-demand service that provides computing capabilities according to user’s needs. Thin or thick client platforms (e.g., mobile phones, tablets, and workstations) access the cloud service through network. Since cloud computing creates new opportunities for workloads offloading, many works study offloading approaches based on cloud to augment mobile devices’ capability. These techniques are categorized into Mobile Cloud Computing [17].

CloneCloud [13] is a cloud-based and thread-level offloading system. It adopts the offline static analysis on every new program binary to identify possible program partitions. Then it collects the metrics data on local and cloud under different program partitions, and finds out a partition to minimize the cost metric. For offloading, it clones the entire mobile platform into cloud virtual machine, and then migrates the partitioned program to the virtual machine. In our frame-



work, OpenCL programs don't need prior partition analysis before offloading. We offload the whole OpenCL program by remotely invoking the APIs written in the program and transferring corresponding data. Moreover, their work uses virtualization techniques to create a homogeneous execution environment for mobile devices and clouds. On the other hand, our work uses OpenCL standard to deal with the heterogeneity among computers. Through this approach, mobile devices only transfer required data to the cloud instead of entire platform. And the cloud uses OpenCL runtime system in order to provide a compatible execution environment.

MAUI [14] is a cloud-based and method level offloading system aiming to reduce energy consumption. It requires developers to annotate methods that should be considered for the workload offloading. Their system decides at runtime which methods should be remotely executed. Once an offloaded method terminates, MAUI gathers profiling information that is used to better predict whether future invocations should be offloaded. In this approach, programmers have to use annotation to inform the system whether a method can be offloaded. Therefore, existing code needs to be modified and reorganized, and developers need to pay attention to restrictions on annotating methods. For example, these methods should not interact with I/O that only make sense on local (e.g., GPS). Our work transparently replaces the original OpenCL API implementation. Therefore, existing OpenCL code works on our framework without modification. Developers simply use the standard OpenCL API to implement OpenCL programs that can be executed on our framework.

Furthermore, there are studies about offloading workload to nearby mobile devices. This approach is useful in several situations. For example, when a user seeks for a cloud service to accelerate some computations, however, is unable to access the Internet for some reasons. By utilizing nearby mobile resources, user can still enhance their mobile devices' capability without accessing the cloud.

Hyrax [20] ports Hadoop [25] to the Android platform. It applies MapReduce [15] programming model to process large data over multiple mobile devices. To avoid file sharing overhead, Hyrax assumes data is constant. Therefore, it is not suitable for applications whose data frequently changes (e.g., event-driven application). Their performance experiments show main overhead comes from Hadoop system. They noticed that since Android has 16MB application memory limit, Hadoop system has to swap data to disk many times in order to process the whole data, which degrades the performance. Their work uses MapReduce to run a program. MapReduce is designed to run programs on a cluster i.e. multiple computers works together to execute a program. Unlike their approach, OpenCL is for single computer. We offload an OpenCL program to single computer, so there are no cluster issues. (e.g., data transfer overhead among computing nodes)

Huerta-Canepa et al. [18] proposes a framework to connect nearby mobile devices together as a virtual cloud by ad hoc, and offload computation to them. Their framework intercepts a program

at loading time and modifies code of the program to add RPC (Remote Procedure Call) support for offloading. A program profiling is performed before offloading to determine the number of remote devices needed to compute this program. Locations of nearby devices are traced, and are used for peer-to-peer connection during offloading. During offloading, the partitioned application and data is transferred to devices within virtual cloud for execution. Once completes, the result is integrated. Similar with Hyrax, their work offloads a program to multiple devices. However, our approach offloads a program to single one server. Additionally, their work modifies program code to support offloading. But, in our framework, a program code does not require modification for offloading.

In contrast to the previous works, we adopt an offloading approach focusing on general-purpose computing on graphics processing unit (GPGPU). CUDA (Compute Unified Device Architecture) [4] and OpenCL (Open Computing Language) [24] are two major GPU programming models. Compared with CUDA that is only supported by Nvidia GPU, OpenCL is supported by GPU or CPU from various manufactures such as Nvidia [9], AMD [1], and Intel [6]. As computer hardware is become more and more heterogeneous, OpenCL provides better support for heterogeneous hardware. Therefore, our framework is based on OpenCL.

2.2 OpenCL

OpenCL is a framework for parallel programming. It includes a language, a set of APIs, and a runtime system to support applications development and execution. Developers use the framework to accelerate program across heterogeneous hardware including CPUs, GPUs, digital signal processors (DSPs), and field-programmable gate arrays (FPGAs).

We use the *platform model* and the *execution model* to introduce core ideas behind OpenCL.

2.2.1 Platform Model

The OpenCL platform model is depicted in Figure 2.1. The model consists of a *host* equipped with one or more *compute devices*. A compute device includes one or more compute units each with one or more processing elements.

The host submits *commands* from the host to compute device to manipulate the compute device and execute computations on processing elements.

2.2.2 Execution Model

An OpenCL program includes the *host program* and *kernels*. A host program runs on the host computer, and kernels runs on a compute device. The host program manages the execution of the

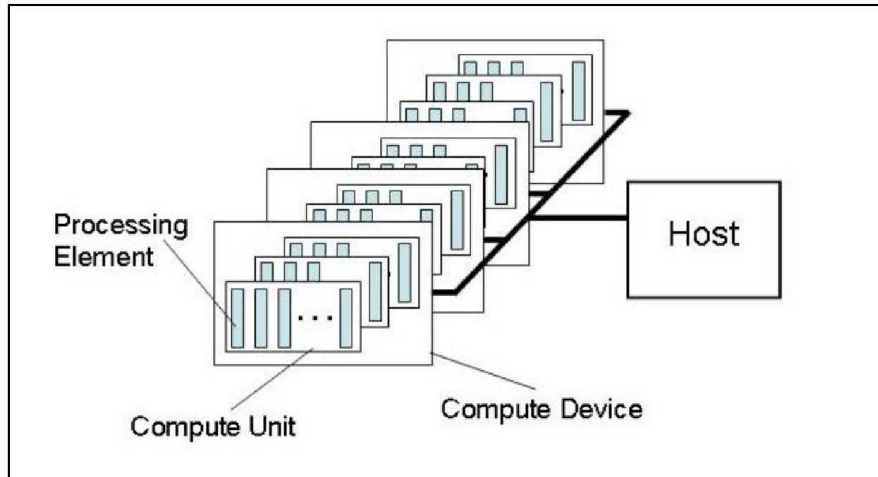


Figure 2.1: The OpenCL platform model [16]

kernels by invoking OpenCL APIs. These APIs create or manipulate the OpenCL objects. Core OpenCL objects are *platform*, *compute device*, *context*, *program*, *kernel*, *buffer* and *command queue*.

A *platform* is the abstraction of the platform model. It's an interface for host program to obtain *compute devices* within the host. A *context* encapsulates required OpenCL objects (command queue, program, kernel, buffer) to execute the kernels. A *program* is the source and executable that implements the kernels. A *kernel* is the abstraction of kernels. It encapsulates a program, and it's used for setting arguments of the kernels. A *buffer* is the abstraction of compute device memory. A *command queue* is used for host program to issue commands to compute devices.

An OpenCL program execution flow is described as follows. To begin with, the host program gets a platform and obtains the list of compute devices available on the platform. Then, it creates a context and adds a command queue in the context. Next, the host program compiles the kernel source code at runtime to create a program and a kernel. After that, the host program creates buffers to allocate device memory, and issues the write buffer command to upload the kernel input data. The host program then sets the arguments of the kernels and issues a command to execute the kernels. Finally, the host program issues the read buffer command to download the kernel output data from the compute device.



Chapter 3

System Architecture

Figure 3.1 shows the architecture of our OpenCL computation offloading framework. The architecture consists of a client called “*OpenCL client*”, a server called “*OpenCL server*” and a manager called “*OpenCL manager*”. A client offloads an OpenCL program on a mobile device. A server provides one or more compute devices for clients to execute OpenCL programs. A manager is a stationary computer provides centralized service for clients in vicinity. (e.g., the manager can be a public stationary computer in a working place.) The manager maintains compute devices in the network. The client request a compute device from the manager to execute their OpenCL program, and the manager decides whether the OpenCL program should run on a client’s compute device or offload to a server’s compute device. These components communicate with each other through a network.

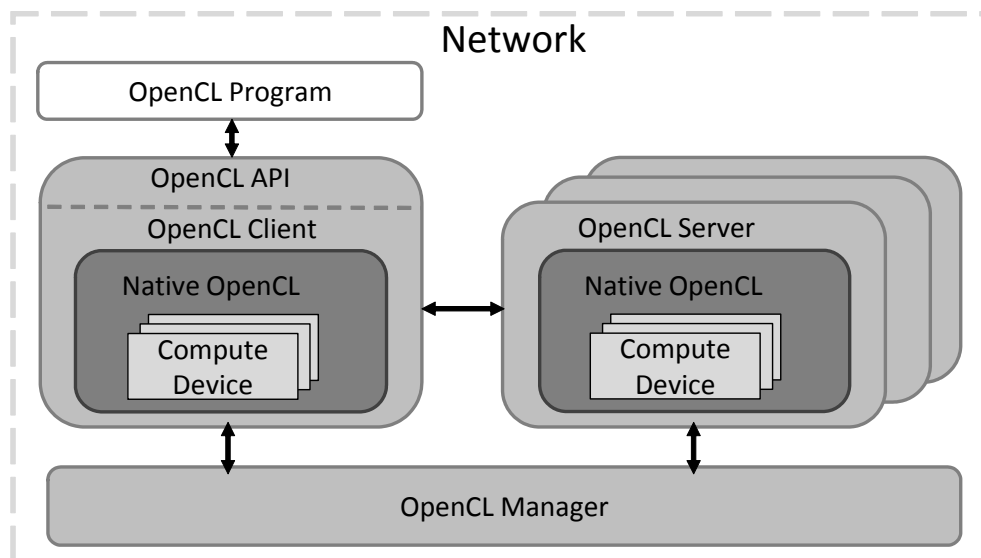
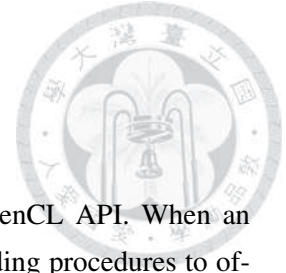


Figure 3.1: OpenCL computation offloading framework



3.1 System Components

3.1.1 OpenCL Client

The client is implemented by transparently replacing the original OpenCL API. When an OpenCL program invokes the OpenCL API, the client performs corresponding procedures to offload the program. These procedures includes *program data collection*, *device query procedure*, *remote offloading procedure*, *local offloading procedure*, and *read buffer procedure*.

When the host program invokes the APIs to create or manipulate OpenCL objects including *program*, *kernel*, and *buffer*, the client performs *program data collection*. This procedure stores API arguments such as kernel source, values of kernel arguments and buffer size. The client collects these data for two purposes. One is to send the program data to the manager for requesting a compute device to offload. Since program data describes the content of the OpenCL program, the manager uses it to find out the best device running the program. Another is to use these data as arguments for invoking native OpenCL APIs to create and manipulate the native OpenCL objects.

When the host program invokes the API to issue kernel execution command, the client performs *device query procedure* to request a compute device from the manager. If the compute device is on the remote server, the client performs *remote offloading procedure* to run the OpenCL program on the remote compute device. Otherwise, the client performs *local execution procedure* to directly run the OpenCL program on the local compute device. Both procedures use collected program data to create and manipulate the native OpenCL objects. What's different is that, in remote offloading procedure, the client has to transfer the program data to remote server. As a result, network operations involves in the remote offloading procedure, so the network status affects this procedure.

The offloading procedure terminates if the network or OpenCL execution fails. For example, the server may disconnect with the client, or the compute device may fail to run the kernels. In the case of network failures, the client performs device query procedure again, and the manager returns another available compute device. The client then starts offloading procedure with the new compute device. By doing so, our system is able to recover from network failures. In the case of an OpenCL failure, a standard OpenCL error code is returned in order to help developers debug for their OpenCL program.

Finally, when the host program invokes the API to issue read buffer command, the clients performs *read buffer procedure*. This procedures downloads the kernel output data from the compute device. The compute device can be on a remote server or the client itself. If it's on a remote server, the client downloads it from the server over the network. Otherwise, the client directly invokes the native OpenCL API to download the output data from the client's device.



3.1.2 OpenCL Server

A server continuously listens network connections from clients. Once accepts a client connection, it creates a service thread to serve the client. Initially, the service thread sets up a *platform* for serving this client. The purpose of this platform is to host every OpenCL programs offloaded by the client. The service thread listens client's offloading requests, and invokes corresponding native OpenCL APIs to execute the OpenCL program. The service thread creates a *context* for each OpenCL program. Each context encapsulates the OpenCL objects in the OpenCL program. Therefore, the objects from different OpenCL programs are isolated.

3.1.3 OpenCL Manager

The manager publishes its IP address, so clients in vicinity and servers can connect to it.

The manager maintains a list of available servers. When a server starts, the manager adds the server to the list. When a server stops its service or disconnects with the manager (e.g., leaving the reachable range of the manager), the manager removes it from the list.

The manager allocates compute devices to clients according the time spent on network operations and the time spent on executing the offloaded program. We define the following function to help illustrate our allocation method. Suppose a client offloads an OpenCL program p to a compute device d . $T_{network}(p, d)$ is the time spent on network operations to transfer p 's data to and from d . And $T_{exe}(p, d)$ is the time spent to execute p on d . $T_{total}(p, d)$ is the total execution time for data transferring and executing p on d . It's the sum of $T_{network}(p, d)$ and $T_{exe}(p, d)$, as in Equation 3.1.

$$T_{total}(p, d) = T_{network}(p, d) + T_{exe}(p, d) \quad (3.1)$$

When a client request a compute device from the manager to offload an OpenCL program p , the manager's objective is to select a compute device with minimum $T_{total}(p, d)$. To achieve this, for each compute device d , the manager estimates its $T_{network}(p, d)$ and $T_{exe}(p, d)$, and sums them up to get the estimated $T_{total}(p, d)$. Then the manager returns to the client the compute device with minimum estimated $T_{total}(p, d)$.

In our approach, the estimated $T_{network}(p, d)$ is the kernel data transfer time to and from the compute device d . Because data other than kernel data are usually smaller than 1 KB which take very short time to transfer under normal network speed (over 1 MB/s) [8], therefore, we assume overhead of transferring data other than kernel data are negligible. (e.g., we ignores time spent on transferring kernel source). Furthermore, if the compute device d is located in the client itself, the estimated $T_{network}(p, d)$ is 0 because the client does not have to transmit or receive the kernel data through the network. The manager monitors the network receive and transmit speed of each

server. This network speed is used to calculate the data transfer time. For example, the input and output data size of the kernels are 8 MB and 4 MB respectively, and the network receive and transmit speed of a server are both 4 MB per second. In this case, the server will spend 2 seconds to receive the kernel input data and 1 second to transmit the kernel output data. Therefore, the estimated $T_{network}(p, d)$ is 3 seconds.

In our approach, the estimated $T_{exe}(p, d)$ is the average time of executing p on d . We assume that OpenCL programs will be offloaded in the same network environment multiple times. We also assume the control flow of OpenCL programs is independent of program data. To achieve this, we relies on *historical performance records*. Each compute device has historical performance records. Each historical performance record stores data. We show an example of historical performance record in Table 3.1. In this example, the device owns the record has executed an OpenCL program for 22 times (values in *Count*) with the given kernel name and source and the data size. The average execution time of the OpenCL program is 387 ms. If the p and d are the case in the example, the estimated $T_{exe}(p, d)$ is 387 ms.

Table 3.1: The historical performance record

Kernel Name	Kernel Source	Kernel Input Data Size (MB)	Kernel Output Data Size (MB)	Average Execution Time (ms)	Count
vecAdd	reference to Listing 3.1	8	4	387	22

Listing 3.1: The vecAdd kernel source

```

1  __kernel void vecAdd(__global int *A,
2                          __global int *B,
3                          __global int *C,
4                          const unsigned int size)
5  {
6      int id = get_global_id(0);
7      if(id < size)
8          C[id] = A[id] + B[id];
9  }

```

In order to collect historical performance records of each compute device, whenever a compute device finishes an OpenCL program, the manager receives the performance record from the client or the server depending the location of the compute device. This performance record includes the kernel name and the kernel source of the OpenCL program, the input and output data size of the kernels, and the OpenCL program execution time. The manager uses this performance record to find out the corresponding historical performance record and update its *Average Execution Time* and *Count*.

3.2 System Flow

Figure 3.2 depicts the system flow of our framework. At the beginning, an OpenCL program starts executing on a mobile device. The client collects the program data. Then it sends required program data to the manager to request a compute device. The manager selects a compute device and returns it to the client. Next, if the device is on the client itself, the client directly execute the OpenCL program using native OpenCL. If the device is on a remote server, the client connects to the server, and transfer the program data over the network. The server then execute the OpenCL program. After that, the client request kernel output data from the server, and the server sends back the data to the client. Finally, the client or the server sends performance record of their compute device to the manager. The manager then updates the historical performance records.

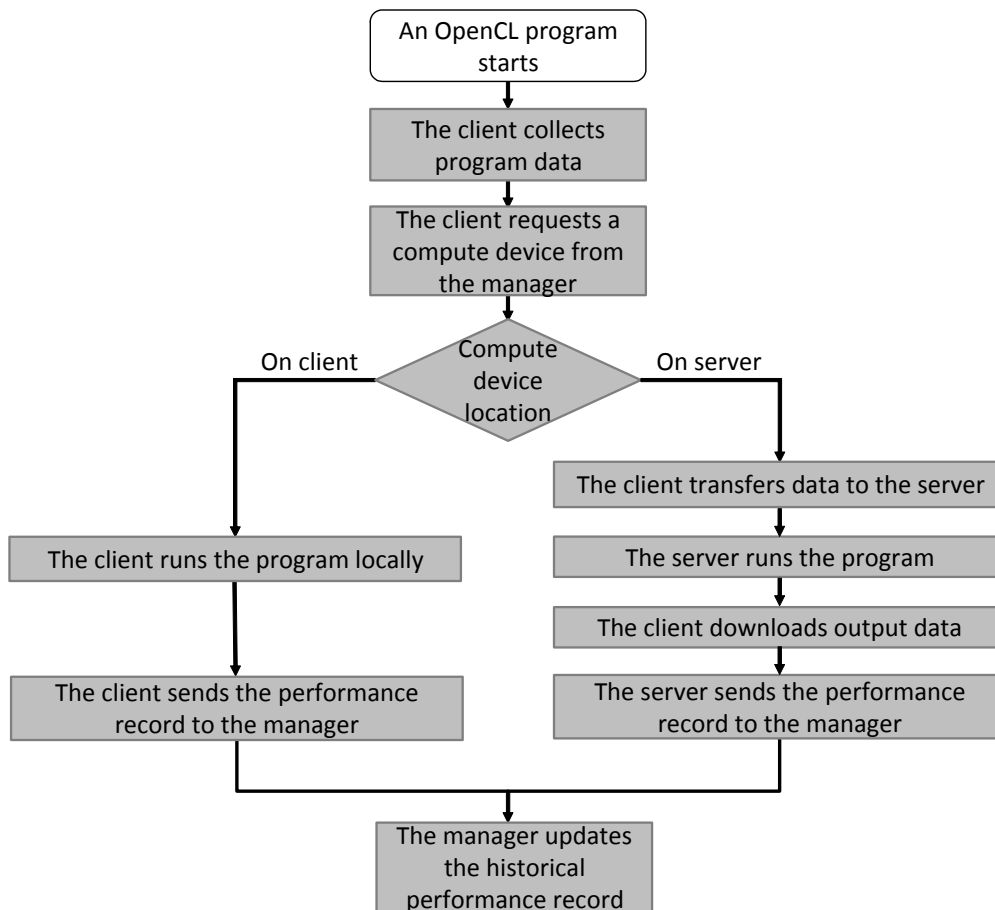


Figure 3.2: System flow



Chapter 4

Implementation

We implemented a prototype of our framework with C/C++. The components in the prototype use Berkeley sockets library to communicate with each other. The communication and data transferring use TCP/IP protocol to provide reliable delivery of packets. The prototype uses JSON data-interchange format [7] for OpenCL data serialization.

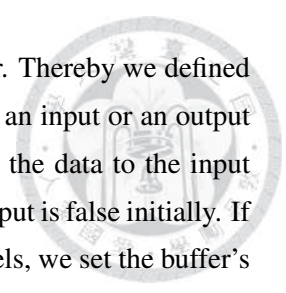
Our current prototype has a limitation caused by the abstracted nature of OpenCL. In our framework, we have to identify different compute devices on a computer, so as to assign programs to a specific compute device. However, in standard OpenCL, a compute device does not have a unique identifier. In our prototype, we simply use the device type (CPU or GPU) and the device name obtained by *clGetDeviceInfo* for device identification.

In our framework, the user needs to set up the manager IP address beforehand to start the OpenCL client or the OpenCL server. In our prototype, a client or a server owns a file which stores the manager IP address. When a client or a server starts, it reads this file from disk so as to connect with the manager.

4.1 OpenCL Client

We implemented the client prototype on Android OS 4.1 branch. OpenCL is a standard with C/C++ interface. Most OpenCL programs on Android use Java Native Interface (JNI) and are built with Android Native Development Kit (Android NDK) [3] to use the native C/C++ OpenCL interface. Thus, we modified this native C/C++ OpenCL interface, replacing original implementation with our OpenCL client functions.

In the prototype, the *clCreateProgramWithSource*, *clCreateKernel*, *clCreateBuffer*, *clEnqueueWriteBuffer* and *clSetKernelArg* store the API arguments as program data including name and source of kernels, flag, size and data of each buffer, and values of each kernel arguments. Additionally, several features specific to our framework have been added:



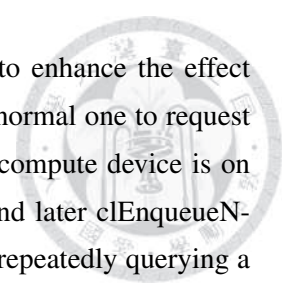
First, we need to specify whether a buffer is an input or an output buffer. Thereby we defined a Boolean attribute *isInput* for a buffer object to specify whether a buffer is an input or an output buffer. Standard OpenCL programs invoke *clEnqueueWriteBuffer* to write the data to the input buffer before executing kernels. Therefore, in our prototype, the value of *isInput* is false initially. If a buffer is accessed by the *clEnqueueWriteBuffer* before executing the kernels, we set the buffer's *isInput* to true.

Furthermore, our framework is based on the use of multiple buffers; that imply the need for the servers of identifying each of these buffers. Thus when a buffer is created by *clCreateBuffer*, a *bufferID* is given to the buffer. The *bufferID* is used as an identifier for a client and a remote server. At the beginning, an OpenCL program creates a buffer with a *bufferID*, and the client offloads the OpenCL program to a remote server. When the program finishes, the client wants to read the result in the buffer. It then sends the *bufferID* to request the data from the server. The server uses the *bufferID* to find out which buffer the client refers to, and returns the data of that buffer.

Then our framework needs to perform remote and local kernels execution. However, the standard *clEnqueueNDRangeKernel* OpenCL interface can proceed only locally. To solve this issue, the original implementation of this API has been replaced by our OpenCL client functions. When *clEnqueueNDRangeKernel* is invoked, the client first performs *device query procedure* to request a compute device from the manager. This procedure sends kernel name, kernel source, and *isInput* and size of each buffer to the manager. The manager selects the compute device and returns a message including and a device type, a device name and an IP address for network connection to the client. After that the client checks whether IP address is the same as itself. If it's the same, then the clients performs *local execution procedure* to execute the OpenCL program on client itself. Otherwise, it performs *remote offloading procedure* to execute the program on the remote server.

In our prototype, the remote offloading procedure including three stages. First stage is to complete building the OpenCL objects including compute device, context, program, kernel and command queue on a remote server. Second stage is to complete building all buffers on a remote server. Third stage is to complete issuing the kernels execution command on a remote server. In each stage, a client transfers required program data to a remote server. The server returns whether the stage was successfully perform. If this is the case, the client proceed to the next stage. If an OpenCL error happens on the server side, the server returns the standard OpenCL error code, so developers can debug for the OpenCL program. If a network error happens, Berkeley sockets library will return error. The client then terminates offloading procedure with the server. It performs device query procedure again to request another compute device, and starts offloading procedure with the new device.

Some OpenCL programs repeatedly invoke *clEnqueueNDRangeKernel* to execute kernels



multiple times. For example, an image filter kernels run multiple times to enhance the effect of the filter. In this case, the first `clEnqueueNDRangeKernel` operates like normal one to request a compute device from the manager, send program data to a server if the compute device is on the server and issue kernels execution command. After that, the second and later `clEnqueueNDRangeKernel` simply issues kernels execution command. This is to avoid repeatedly querying a device from manager and sending the same data set over network.

Lastly, we replaced the `clEnqueueReadBuffer`. This API originally downloads the data from the device memory. In our prototype, when a client executes an OpenCL program locally, the client invokes native `clEnqueueReadBuffer` directly. If a client offloads an OpenCL program to remote server's compute device, the client sends a read buffer request with buffer's `bufferID` to the server. The server then invoke `clEnqueueReadBuffer` to read the data from its compute device, and transfers the data to the client.

4.2 OpenCL Server

In the prototype, the server is implemented as a daemon process running in background. Thus, it won't interfere with any other processes. The server prototype uses Berkeley sockets to listen connection and requests from clients. A client's offloading request consists of a request name and corresponding program data. These requests including `buildKernel`, `buildBuffer`, `executeKernel` and `readBuffer`. In `buildKernel`, the server creates OpenCL objects including compute device, context, program, kernel and command queue. In `buildBuffer`, the server creates all buffers, receives kernel input data and writes data to these buffers. In `executeKernel`, the server issues kernel execution command to the compute device. In `readBuffer`, the server issues read buffer command to download the kernel output data from the compute device, and transmits the data to the client.

The server records the OpenCL program execution time that includes time spent on every native OpenCL API. When the client receives all the kernel output data, it disconnects with the server. The server then sends kernel name, kernel source, input data size, output data size, and OpenCL program execution time as performance record to the manager.

When the server transmits or receives the kernel data, it calculates the network transmit and receive speed. The network speed is sent with performance record, so that the manager knows the network speed of the server.

4.3 OpenCL Manager

The manager prototype maintains an IP address list of available servers.

In the prototype, when the manager starts, it loads the historical performance records from

files. Each record includes an IP address, a device name, a device type, a kernel name, a kernel source, an input data size, an output data size, an average execution time, and a count.

When the manager receives the device query request from a client, it selects the best device and returns device's IP address, device's type, and device's name to the client. When it receive the performance record from a server, it updates corresponding historical performance records. When the manager stops, it stores the historical performance records into files for future usage.



Chapter 5

Experiments

5.1 Experimental Environment

Our hardware setup includes a client, a manager and two servers. The OpenCL client was tested on a smartphone, Sony Xperia ZL with Quad-core 1.5 GHz CPU and 2 GB RAM running with Android 4.1.2. we set up the OpenCL manager on a desktop PC with Intel i5-2400 Quad-core 3.1 GHz CPU and 8 GB RAM running with Ubuntu 12.04. We tested the OpenCL server on an Android smartphone, Sony Xperia ZU and another desktop PC running with Ubuntu 12.04. Each provides different OpenCL compute devices as shown in Table 5.1.

Table 5.1: Server setup

OpenCL Server	Compute Device	Device Type	# of Cores	Clock Rate	Memory
Smartphone (Sony Xperia ZU)	Adreno 330	GPU	128	450 MHz	900MB
Desktop PC	Intel i7-4930K	CPU	6	3.9 GHz	16 GB
	AMD HD 7990	GPU	4096	1 GHz	6 GB

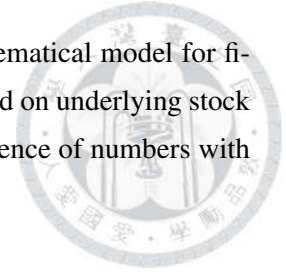
We conducted the experiments on a LAN (Local area network). Table 5.2 shows the network's average bandwidth and RTT (round trip time) between client and other components.

Table 5.2: Network status

	Avg. bandwidth (MB/s)	Avg. RTT (ms)
client and desktop PC server	4.86	8
client and smartphone server	2.44	115
client and manager	4.61	8

We used five OpenCL programs to evaluate our framework. We selected the *matrix multiplication* and the *matrix transpose* which are common and basic computations in mathematics, physics and engineering. We also utilized an Android sample app, *bilateral filter* from Sony [10] which takes an image as input and reduces noise in the image. Furthermore, we used *Black-Scholes*

and *bitonic sort* provided by AMD APPSDK [2]. Black-Scholes is a mathematical model for financial market. It determines an estimate price for a call or a put option based on underlying stock volatility, time to expiration, and others. Bitonic sort sorts an arbitrary sequence of numbers with the bitonic sorting algorithm.



5.2 Experiment Result

The terms shown in the experiment results are defined as follows. *Mobile GPU* denotes the smartphone server's GPU. *PC GPU* denotes the desktop PC server's GPU. *PC CPU* denotes the desktop PC server's CPU. The *Computation* is the OpenCL program execution time which includes time spent on executing every OpenCL function call. The *Kernel Data Transmission* is the kernel data transfer time including sending input data to the server and receiving output data from the server. The *Others* is the time other than computation or kernel data transmission. It includes operations such as creating network connection and exchanging messages between the system components.

We evaluate our framework using five OpenCL programs. For each program, we compare the program execution time on the local CPU and remote servers using different input data size.

Figure 5.1 shows the experiment result of the matrix multiplication. In most of the cases, the servers' execution time are faster than the local CPU since the computation time significantly reduces on these servers. In fact, the matrix multiplication can be effectively improved by offloading it from the local CPU to remote GPUs because the GPU provides better capabilities to process matrix elements in parallel. However, for matrix size 512×512 , the local CPU is slightly faster than mobile GPU. The reason is, in our setup, the smartphone server has lower network speed and takes longer time to create network connection than the PC server. Consequently, for mobile GPU, the performance gain in the computation time is eliminated by the network overhead.

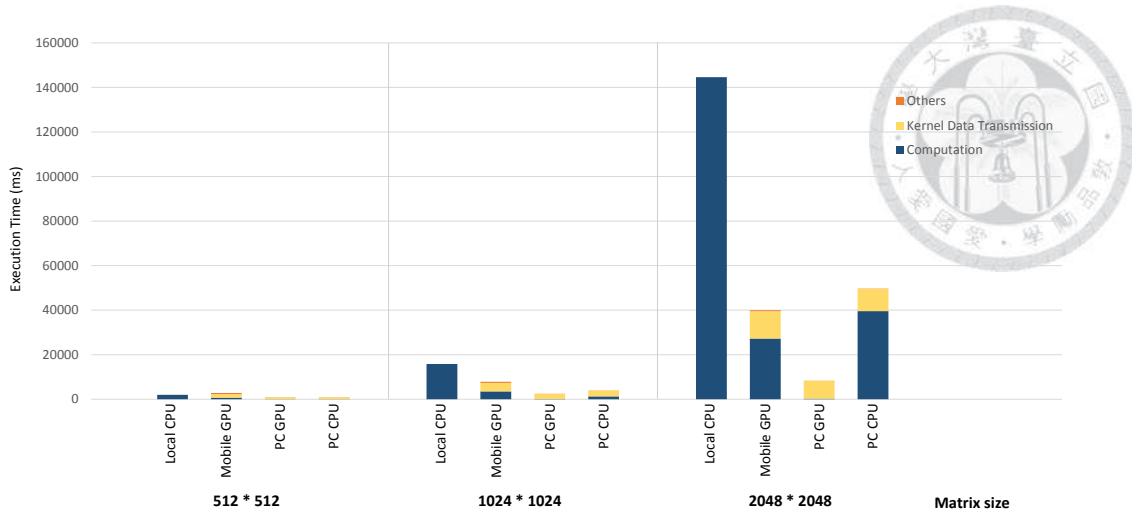


Figure 5.1: Matrix multiplication execution time

In the case of the matrix transpose, local execution is always faster than remote servers as shown in Figure 5.2. This is because the matrix transpose is not a resource-intensive program. It's able to run efficiently enough on the local CPU. Hence, the computation time only reduces very slightly when offloading the program to servers. Such little performance gain in computation is offset by the network overhead.

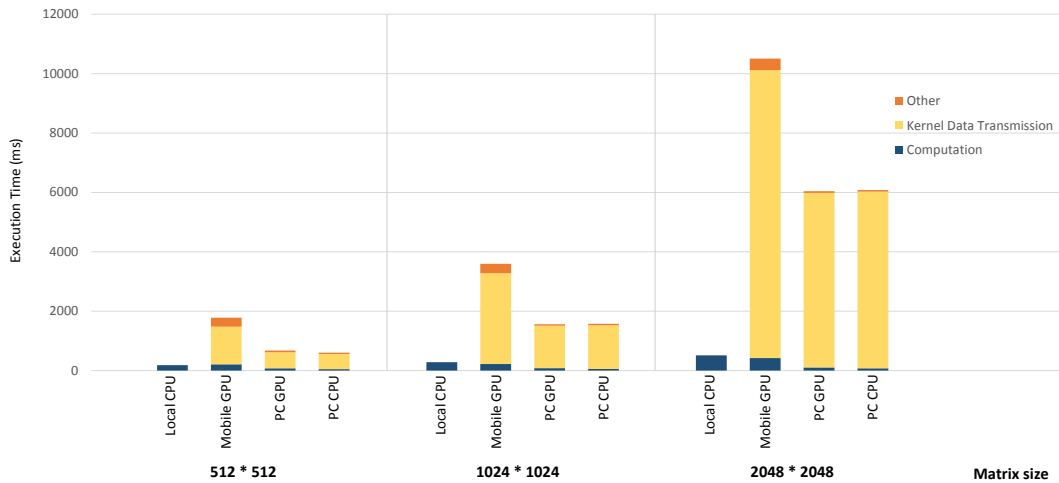


Figure 5.2: Matrix transpose execution time

For bilateral filter, in all cases, remote offloading is much faster than local execution as shown in Figure 5.3. The bilateral filter performs a heavy computation on each pixel of an input image. Since the program benefits from processing all pixels in parallel, when the program is offloaded from the local CPU to remote GPUs, the performance significantly improves. Our result shows 50.3X speedup on the PC server's GPU in image size 2048*2048.

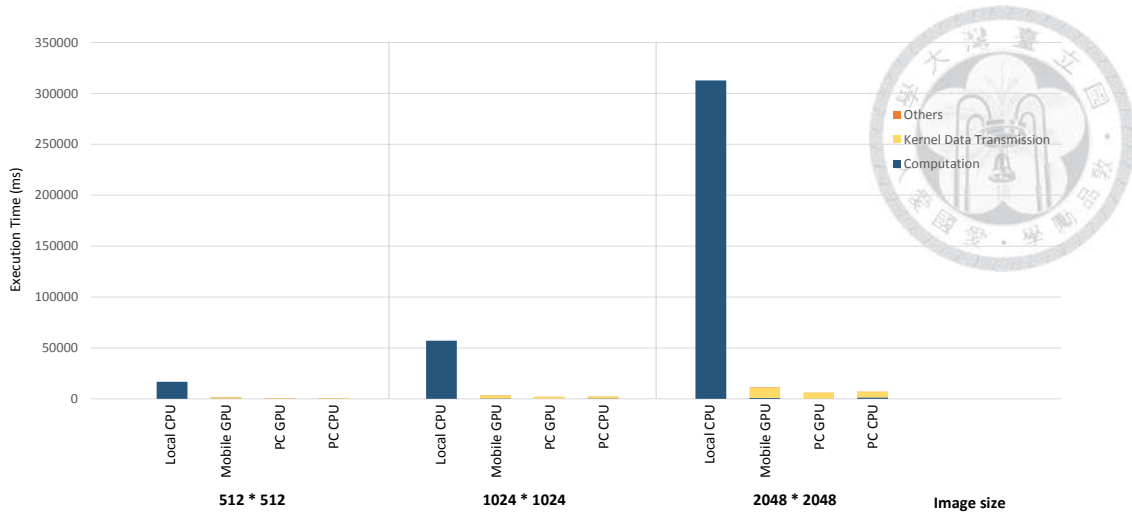


Figure 5.3: Bilateral filter execution time

Figure 5.4 shows the result of the Black-Scholes. For this program, remote servers are slower than local execution in all the cases. The Black-Scholes have obvious performance gain in computation on all servers. However, the data transmission overhead offset this performance gain.

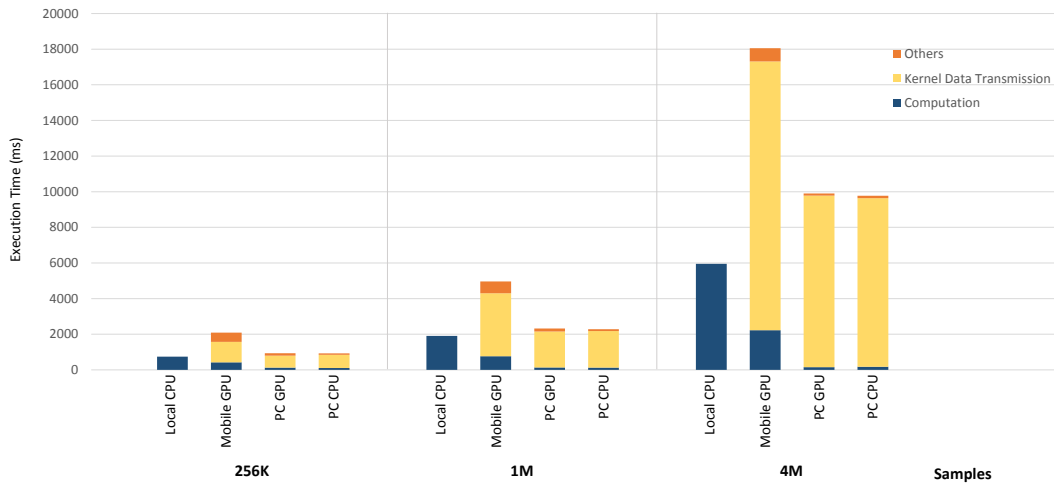


Figure 5.4: Black scholes execution time

Our experiment result for the bitonic sort is shown in Figure 5.5. This program runs faster on remote servers in most of the cases. Same as matrix multiplication and bilateral filter, bitonic sort is well-suited for parallel architectures, so it benefits from offloading to remote GPUs. In fact, the bitonic sort requires executing kernels multiple times to complete sorting the array. For example, in array with one million length, the program executes kernels 210 times. This situation results in more message transmission from the client to the server, so the bitonic sort has relatively more overhead in *Others* compared with other OpenCL programs. However, our result shows that, in

most of the cases, this overhead does not offset the performance gain in servers expect for mobile GPU in 256K array length.

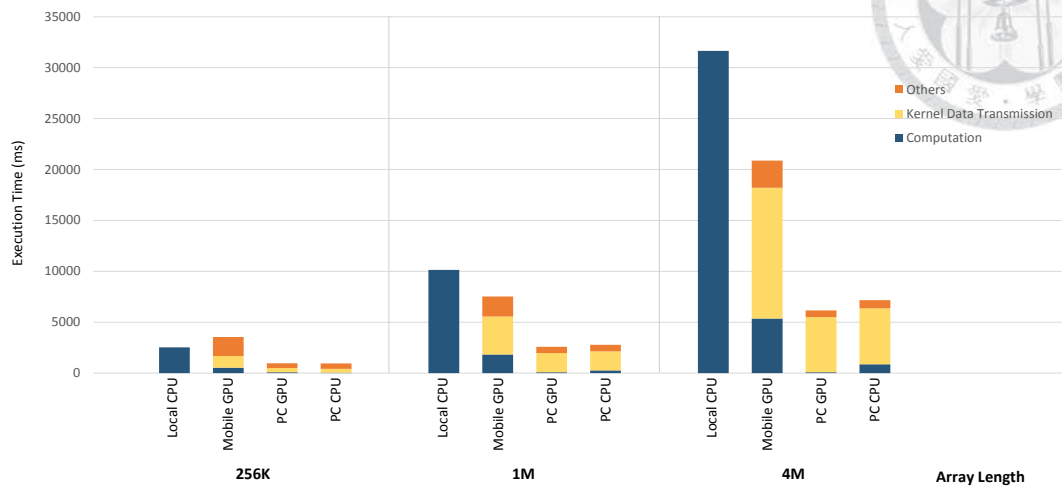


Figure 5.5: Bitonic sort execution time



Chapter 6

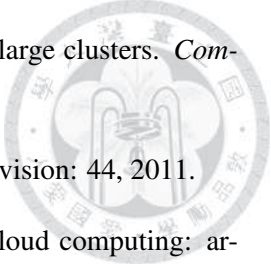
Conclusion

In this thesis, we propose an OpenCL computation offloading framework for mobile device. It adopts a centralized mechanism that manages multiple OpenCL compatible devices and transparently offloads computation from mobile devices to them. This mechanism relies on a dynamic allocation method targeting to reduce the execution time of users' programs being aware of the data transfer overhead and historical program execution data. We evaluated our framework on real hardware with various OpenCL programs. We showed that several programs can be speed up by offloading. For the bilateral filter, offloading shows up to 50.3X speedup.



Bibliography

- [1] Amd. <http://www.amd.com/>.
- [2] Amd app sdk. <http://developer.amd.com/tools-and-sdks/opencv-zone/opencv-tools-sdks/amd-accelerated-parallel-processing-app-sdk/>.
- [3] Android ndk. <https://developer.android.com/tools/sdk/ndk/index.html>.
- [4] Cuda. <http://www.nvidia.com/content/cuda/cuda-toolkit.html>.
- [5] Errison mobility report on the pulse of the networked society. <http://www.ericsson.com/mobility-report>.
- [6] Intel. <http://www.intel.com/>.
- [7] Json. <http://www.json.org/>.
- [8] Net index. <http://www.netindex.com/>.
- [9] Nvidia. <http://www.nvidia.com/>.
- [10] Sony android opencv sample. http://developer.sonymobile.com/knowledge-base/tutorials/android_tutorial/boost-the-performance-of-your-android-app-with-opencv/.
- [11] Amazon elastic compute cloud. <http://aws.amazon.com/ec2/>.
- [12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, Oct. 2008.
- [13] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.
- [14] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

- 
- [15] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
- [16] K. O. W. Group. The OpenCL Specification Version: 1.1 Document Revision: 44, 2011.
- [17] D. T. Hoang, C. Lee, D. Niyato, and P. Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611, 2013.
- [18] G. Huerta-Canepa and D. Lee. A virtual cloud computing provider for mobile devices. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, MCS '10, pages 6:1–6:5, New York, NY, USA, 2010. ACM.
- [19] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. Gpgpu: General-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.
- [20] E. E. Marinelli. Hyrax: cloud computing on mobile devices using mapreduce. Technical report, DTIC Document, 2009.
- [21] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
- [22] Z. Sanaei, S. Abolfazli, A. Gani, and R. Buyya. Heterogeneity in mobile cloud computing: Taxonomy and open challenges. *Communications Surveys Tutorials, IEEE*, 16(1):369–392, First 2014.
- [23] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, Aug 2001.
- [24] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.
- [25] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.