國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

OMP2OCL 轉換器：可自動轉換 OpenMP 程式到 OpenCL 程式的轉換器

OMP2OCL Translator: A Translator for Automatic Translation of OpenMP Programs into OpenCL Programs

研 究 生：蔡宗展

指導教授：單智君 博士

中 華 民 國 九 十 九 年 十 一 月

OMP2OCL Translator：可自動轉換 OpenMP 程式到 OpenCL 程式的轉換器

OMP2OCL Translator: A Translator for Automatic Translation of OpenMP Programs into OpenCL Programs

研 究 生：蔡宗展	Student: Tsung-Chan Tsai
指導教授：單智君 博士	Advisor: Dr. Jyh-Jiun Shann

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

November 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年十一月

OMP2OCL Translator：可自動轉換 OpenMP 程式到 OpenCL 程式的轉換器

學生：蔡宗展　　　　　　　　　　　　　　　　指導教授：單智君 博士

國立交通大學資訊科學與工程研究所碩士班

摘要

由於對高效能系統的需求，異質多處理器平台成為了一個趨勢。然而，跟同質多處理器平台相比，異質多處理器平台難以程式化且多樣化。Knronos Group 近年來釋出了 OpenCL 標準，它改善了異質多處理器平台程式的可攜性，然而使用 OpenCL 撰寫異質多處理器平台程式依然複雜且易出錯。在這篇論文內，我們設計且實作名為 OMP2OCL translator 的轉換器，它自動轉換 OpenMP 程式到 OpenCL 程式。在 OMP2OCL translator 內，我們延用且修改相關研究內的優化，來優化轉換出來的 OpenCL 程式。OMP2OCL translator 鑒定出 OpenMP 程式內的核心區域，改變該核心區域成為 OpenCL 核心函式，且使用優化來改善轉換出來的 OpenCL 程式。此外，我們比較了由相關研究轉換出來的 CUDA 程式和由 OMP2OCL translator 轉換出來的 OpenCL 程式。雖然與針對 NVIDIA GPU 的相關研究相比，OMP2OCL translator 有效能損失，但 OMP2OCL translator 還是很有用，因為轉換出來的 OpenCL 程式並不限定使用 NVIDIA GPUs 為計算設備。

OMP2OCL Translator: A Translator for Automatic Translation of OpenMP Programs into OpenCL Programs

Student: Tsung-Chan Tsai Advisor: Dr. Jyh-Jiun Shann

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Heterogeneous multiprocessor platforms have become trends due to the need for high-performance systems. However, it is harder to program them than homogeneous multiprocessor platforms and there are various heterogeneous multiprocessor platforms in the world. Although newly-released OpenCL (Open Computing Language) standard from Khronos Group offers improves portability among heterogeneous multiprocessor platforms, programming such platforms using OpenCL is still complex and error-prone. In this thesis, we have designed and implemented a translator, called OMP2OCL Translator, which is for automatic source-to-source translation of OpenMP programs into OpenCL programs, in addition, we have reused and modified if necessary the optimizations from the related work to improve the OpenCL programs output from OMP2OCL translator. The translator identifies kernel regions of OpenMP programs, transforms and outlines the regions into OpenCL kernel functions, and does some optimization to improve the performance of the translated OpenCL programs. Moreover, we have compared the CUDA programs and the OpenCL programs output from the related work and OMP2OCL translator, respectively. Although there are performance losses for OMP2OCL translator compared to the related work designed dedicatedly for NVIDIA GPUs, it is still promising that the translated OpenCL programs can use other devices as compute devices other than NVIDIA GPUs.

致謝

首先感謝我的指導老師 單智君教授，在這兩年當中不論是正式報告，亦或是平日小組討論，老師對於學生的諄諄教誨，細心指導與勉勵，使我學習到如何面對問題，以及如何克服問題，並培養獨立研究的能力。有幸跟著老師做研究，觀察老師對於一件事情的執著與細膩，耳濡目染，並從中學習，最後完成了研究與碩士學位。同時，也感謝口試委員，鍾葉青教授、楊武教授與游逸平教授，由於教授們的指導與建議，才使得此篇論文更加完整與充實。另外，也謝謝實驗室的另一位老師，鍾崇斌教授，在一次次的報告之中給予學生指導與建議。

裕生學長、奕緯學長，感謝兩位學長帶領我進入 JVM 與 Compiler 領域，不僅僅只是給予我研究上的建議與討論，同時也常是鼓勵我前進的動力，才使得我學習到相關知識並完成此研究。同時，實驗室的學長姐、同儕以及學弟妹，在這一起渡過的時光中，你們不僅僅是我記憶中的好夥伴，更是人生道路上的貴人們，謝謝。

最後，對於我的家人以及總是在我低潮時給予鼓勵並且陪伴著我的親友，宗展也在此獻上最誠摯的謝意。

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Multiprocessor platforms are ubiquitous, ranging from embedded devices, mobile devices to workstations and high-end servers. Chip multiprocessors [1], as known as multi-core processors, are one kind of homogeneous multiprocessor platforms and have become the important trend in contemporary computer architectures by major microprocessor vendors, as these vendors could not overcome some problems related to physical constraints (e.g., power, thermal, and signaling) and limited instruction-level parallelism. In addition, heterogeneous multiprocessor platforms play an important role in the computer industry, and they could be found in mobile devices and workstations with CPU plus GPU or CPU plus DSP, for instance, iPhone 4 is powered by the Apple A4 chip, which is a system-on-a-chip and composed of a Cortex-A8 CPU integrated with a PowerVR SGX 535 GPU.

1.1 Research Motivation

Heterogeneous multiprocessor platforms especially have much potential regarding performance gain than homogeneous multiprocessor platforms, by partitioning programs into different properties of tasks and scheduling the tasks on suitable processors; hence, they are widely used in some dedicated systems. Nevertheless, heterogeneous multiprocessor platforms are harder to program, and there are various heterogeneous multiprocessor platforms in the world. To alleviate these issues, many parallel programming standards have been proposed, and one of them is OpenCL (Open Computing Language) [2].

OpenCL is an open industry standard from the Khronos group supported by AMD, IBM, Intel, NVIDA and others. It models heterogeneous multiprocessor platforms, consisting of a host and one or several devices, which can be CPU, GPU, DSP, Cell/B.E. processor, etc. Targeting for various devices from vendors makes OpenCL programs portable. In other words, once an OpenCL program is written, it can be compiled if necessary and run on any heterogeneous multiprocessor platform with OpenCL environments.

Despite of being portable, OpenCL is more complex to use than OpenMP [3]. OpenMP is yet another open industry standard maintained by the OpenMP Architecture Review Board, mainly targeting for homogeneous multiprocessor platforms with shared memory [4]. By using its directives to annotate structured blocks that could be paralleled and to insert some auxiliary annotations, among others, OpenMP is pretty simple to use.

1.2 Research Objective

In order to program heterogeneous multiprocessor platforms with the benefits of both OpenMP and OpenCL, which are ease-of-use and portability respectively, we design and implement a translator for automatic translation of OpenMP programs into OpenCL programs. The translation roughly consists of the following four steps:

i. Analyzing the OpenMP program under the models of OpenCL.

ii. Identifying kernel regions, i.e., regions executed on OpenCL devices.

iii. Transforming and outlining kernel regions into OpenCL kernel functions.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 describes OpenMP, OpenCL, CUDA, and the related work; Chapter 3 presents the design and implementation of OMP2OCL translator; Chapter 4 discusses and compares the performance of the OpenCL programs output by OMP2OCL translator and that of the CUDA programs output by the related work's translator; finally, Chapter 5 dedicates to the conclusions we draw and the future work plan.

Chapter 2 Background and Related Work

This chapter first introduces several parallel programming models including OpenMP, OpenCL, and CUDA. OpenMP and OpenCL are different in nature, from the models behind the scenes to the language constructs that developers use commonly. In contrast, OpenCL and CUDA are similar to each other, with just some names of components changed, some components removed, and some components added. This chapter then presents “OpenMP to GPGPU: a compiler framework for automatic translation and optimization” as a related work, the authors of which designed and implemented an automatic source-to-source translator of OpenMP programs into CUDA programs.

2.1 OpenMP

OpenMP is an open industry standard maintained by the OpenMP Architecture Review Board, mainly targeting for homogeneous multiprocessor platforms with shared memory [4].

The execution model of OpenMP is a fork-join model of parallel execution as shown in Figure 2-1. In Figure 2-1, an OpenMP program has only one thread initially, called the initial thread. When the initial thread encounters a *parallel* directive, it forks a team of itself and zero or more additional threads, and becomes the master of the new team. After finishing the *parallel* region, the team of threads joins together, and, as a result, only the master thread which is the initial thread continues execution. Furthermore, there are implicit synchronizations before and after the *parallel* regions in order to maintain program correctness.

The memory model of OpenMP is shared memory with each thread having its own private memory as shown in Figure 2-2. Data can be shared or private, shared data are accessible by all threads; in contrast, private data can only be accessed by the thread owning it.

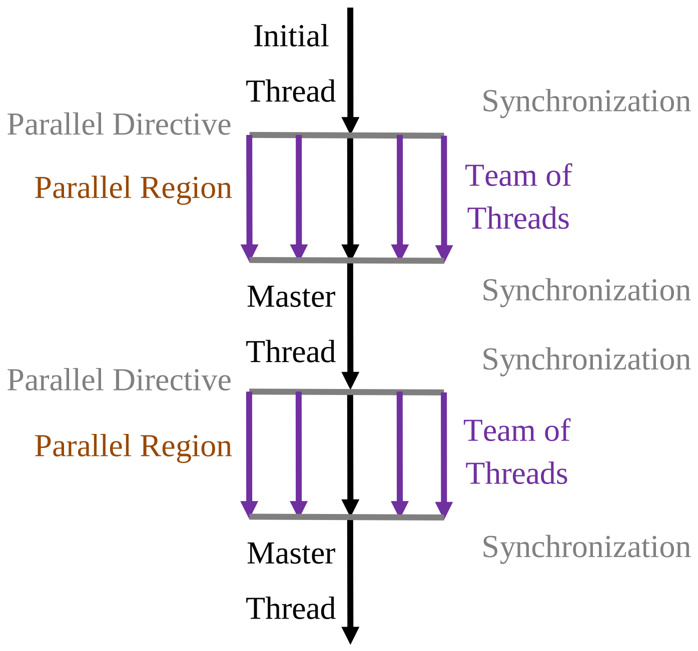


Figure 2-1 The Execution Model of OpenMP

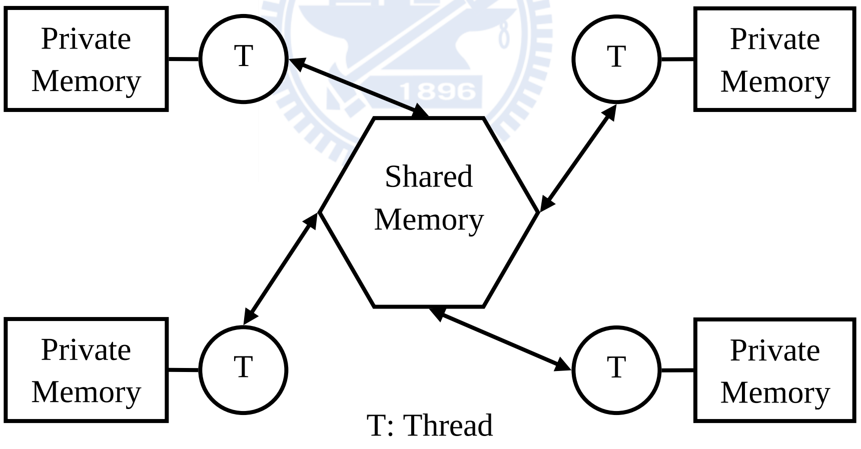


Figure 2-2 The Memory Model of OpenMP

Core elements of OpenMP standard consist of directives, runtime library routines, and environment variables, and directives are at the core of OpenMP standard. *Parallel* directive is used to annotate structured blocks that could be paralleled, and work-sharing directive is used to annotate works which could be divided and distributed to threads. In C/C++, OpenMP directives are specified by using the #pragma mechanism provided by the C/C++ standards. A simple example of an OpenMP program example is shown in Figure 2-3, and it uses both a *parallel* directive and a work-sharing directive. In line 10 of Figure 2-3, the *parallel* directive is used to indicate that the following structured block is fit for being paralleled; likewise, in line 12, the *for* directive, which belongs to work-sharing directives, indicate that the iterations of the following *for* loop should be divided and distributed to threads.

```
 1 #include <stdio.h>
 2 #define N 1024
 3
 4 int a[N][N], b;
 5
 6 int main()
 7 {
 8       int i, j;
 9
10       #pragma omp parallel
11       {
12             #pragma omp for private(j)
13             for (i = 0; i < N; i++) {
14                   for (j = 0; j < N; j++) {
15                         a[i][j] = b + 1;
16                   }
17             }
18
19             #pragma omp single
20             {
21                   puts("XD");
22             }
23       }
24
25       return 0;
26 }
```

Figure 2-3 A Simple Example of an OpenMP Program

2.2 OpenCL

OpenCL is an open industry standard from the Khronos group supported by AMD, IBM, Intel, NVIDA and others [5]. Thus it is vendor-neutral, and it models heterogeneous multiprocessor platforms, consisting of a host and one or more devices, which can be CPU, GPU, DSP, Cell/B.E. processor, etc. OpenCL specifies four models as its architecture: platform model, execution model, memory model, and programming model [6].

2.2.1 Platform Model

In the platform model, one host connects to one or more OpenCL devices (compute devices) on a platform as shown in Figure 2-4 [6]. One or more processing elements (PEs) compose a compute unit (CU), and one or more compute units compose an OpenCL device. Processing elements within a device are the workers doing computations.

An OpenCL program runs on a host in compliance with the native models of the host platform, and sends commands from the host to processing elements within a device for executing computations. The processing elements within a compute unit execute an instruction stream as SIMD units (each PE has its own data and a shared program counter) or as SPMD units (each PE has its own data and its own program counter).

Figure 2-5 shows a platform example with an Intel Core 2 as the host and a NVIDIA's OpenCL platform. The platform has only one compute device, which is NVIDIA GeForce GTX 280 in this example. The device has totally 240 shader processors grouped into 30 sets, each set having 8 shader processors. As a consequence, when mapped to the OpenCL platform model, the device has 30 compute units, each of which composes 8 processing elements.

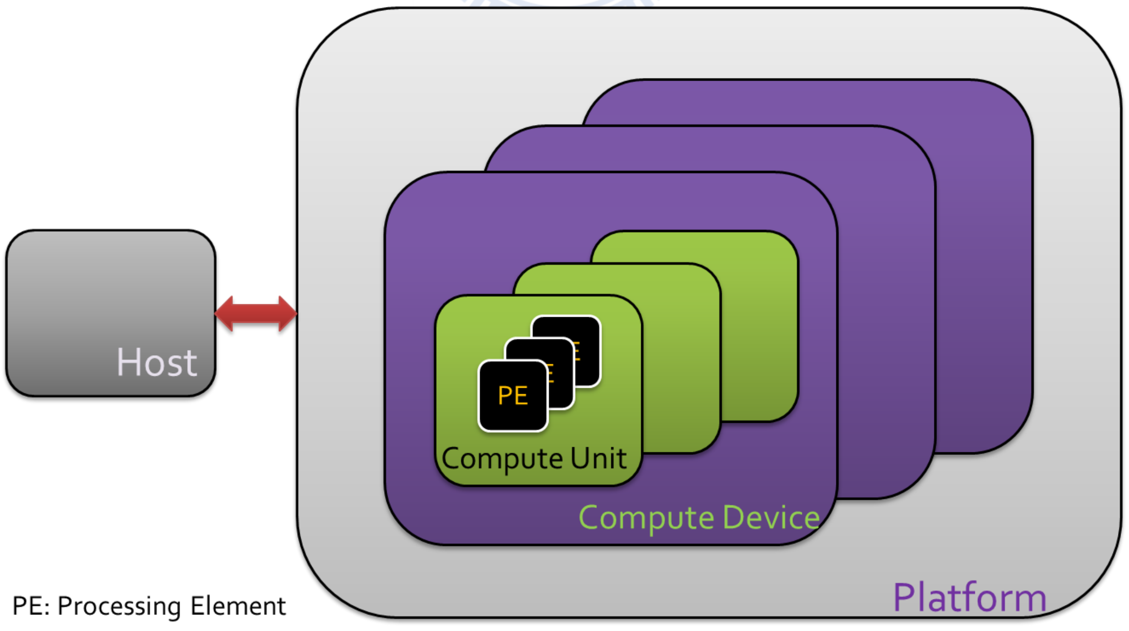


Figure 2-4 The Platform Model of OpenCL

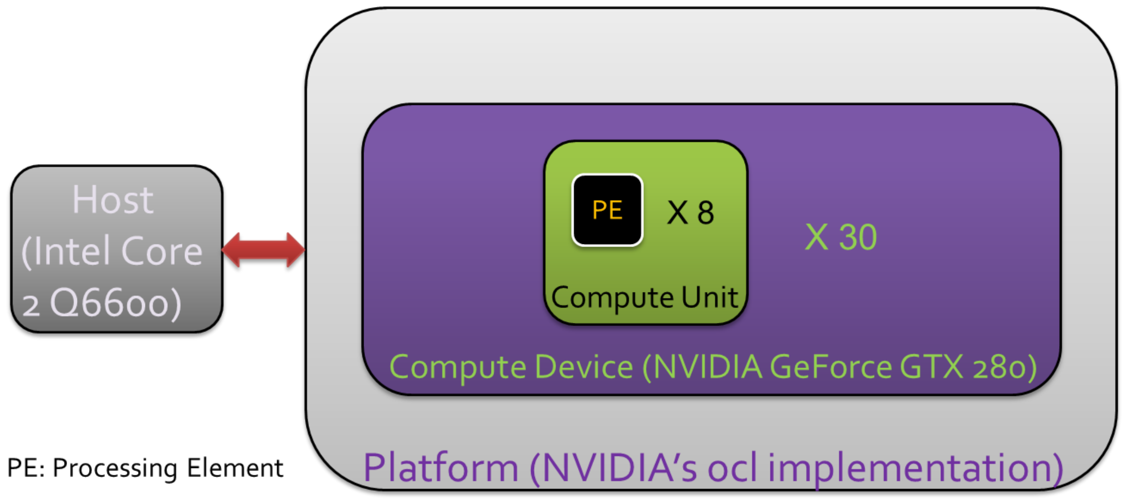

Host
(Intel Core
2 Q6600)
PE
X 8
Compute Unit
X 30
Compute Device (NVIDIA GeForce GTX 280)
PE: Processing Element
Platform (NVIDIA's ocl implementation)

Figure 2-5 The Platform Example of OpenCL

2.2.2 Execution Model

An OpenCL program is comprised of two parts: a host program executing on host and kernels executing on one or more OpenCL devices [6]. The host program defines the context for the kernels and manages their execution.

The way how a kernel should be executed on an OpenCL device is the heart of the OpenCL execution model as shown in Figure 2-6. When a kernel is executed on an OpenCL device, an index space is formed, and each point in the index space stand for a kernel instance, which is named as a work-item and identified by its position in the index space. Although each work-item executes the same code, it can have its own data and its own program counter; in other words, control flow through the code can vary per work-item. Work-items are further organized into work-groups, which provide a more coarse-grained decomposition of the index space; in addition, the work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

The index space within OpenCL is called an NDRange, which is an *N*-dimensional index space, where *N* is one, two or three. An NDRange is defined by an integer array of length *N* specifying the extent of the index space in each dimension starting at an offset index (zero by default).

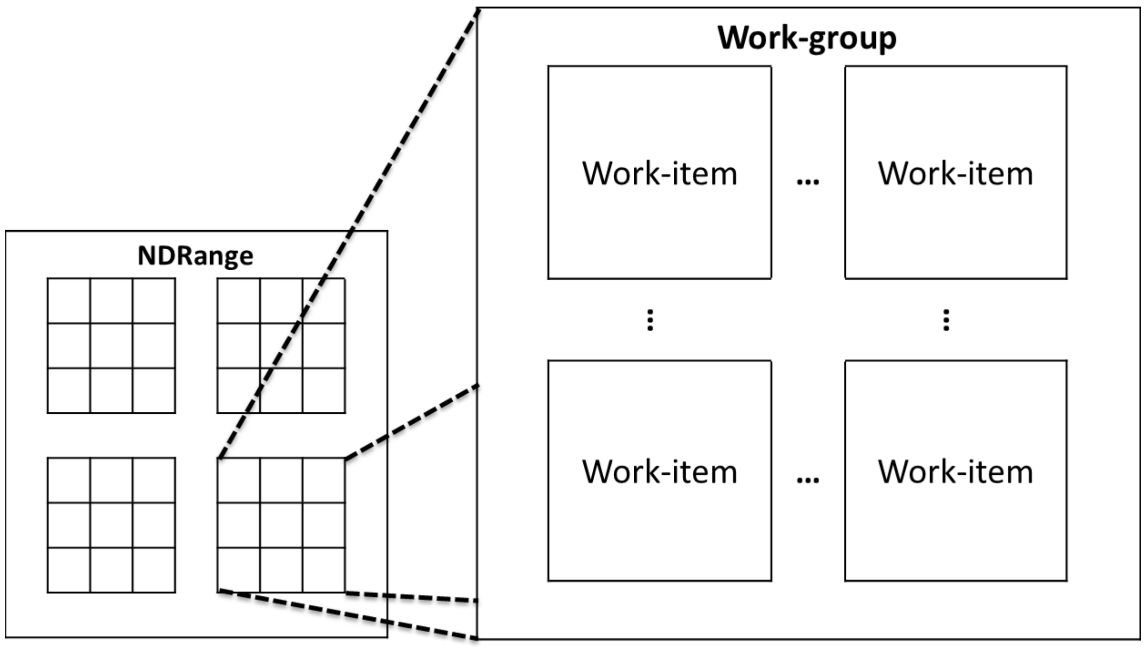


Figure 2-6 The Execution Model of OpenCL

2.2.3 Memory Model

Work-item(s) executing a kernel may access four distinct memory regions [6]:

- Global Memory:

  This memory region permits read/write access to all work-items in all work-groups, so work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.

- Constant Memory:

  This memory region is a region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

- Local Memory:

  This memory region is local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.

- Private Memory:

  This memory region is private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

Table 2-1 describes whether the host or the device can allocate from a memory region or not and the type of allocation (static, i.e., compile time vs. dynamic, i.e., runtime). Table 2-2 describes the type of access allowed, i.e., whether the host or the device can read and/or write to a memory region; moreover, Table 2-3 describes lifetime of variables defined in ach memory regions.

Table 2-1 Type of Allocation to Each Memory Regions for Hosts and Devices

Allocation	Global	Constant	Local	Private
Host	Dynamic	Dynamic	Dynamic	No
Device	No	Static	Static	Static

Table 2-2 Access Allowed to Each Memory Regions for Hosts and Devices

Access	Global	Constant	Local	Private
Host	Read/write	Read/write	No	No
Device	Read/write	Read-only	Read/write	Read/write

Table 2-3 Lifetime of Variables Defined in Each Memory Regions

	Global	Constant	Local	Private
Lifetime	From allocation to deallocation by host program	From allocation to deallocation by host program	As work-group's lifetime	As work-item's lifetime

The memory regions and how they relate to the platform model and the execution model are described in Figure 2-7.

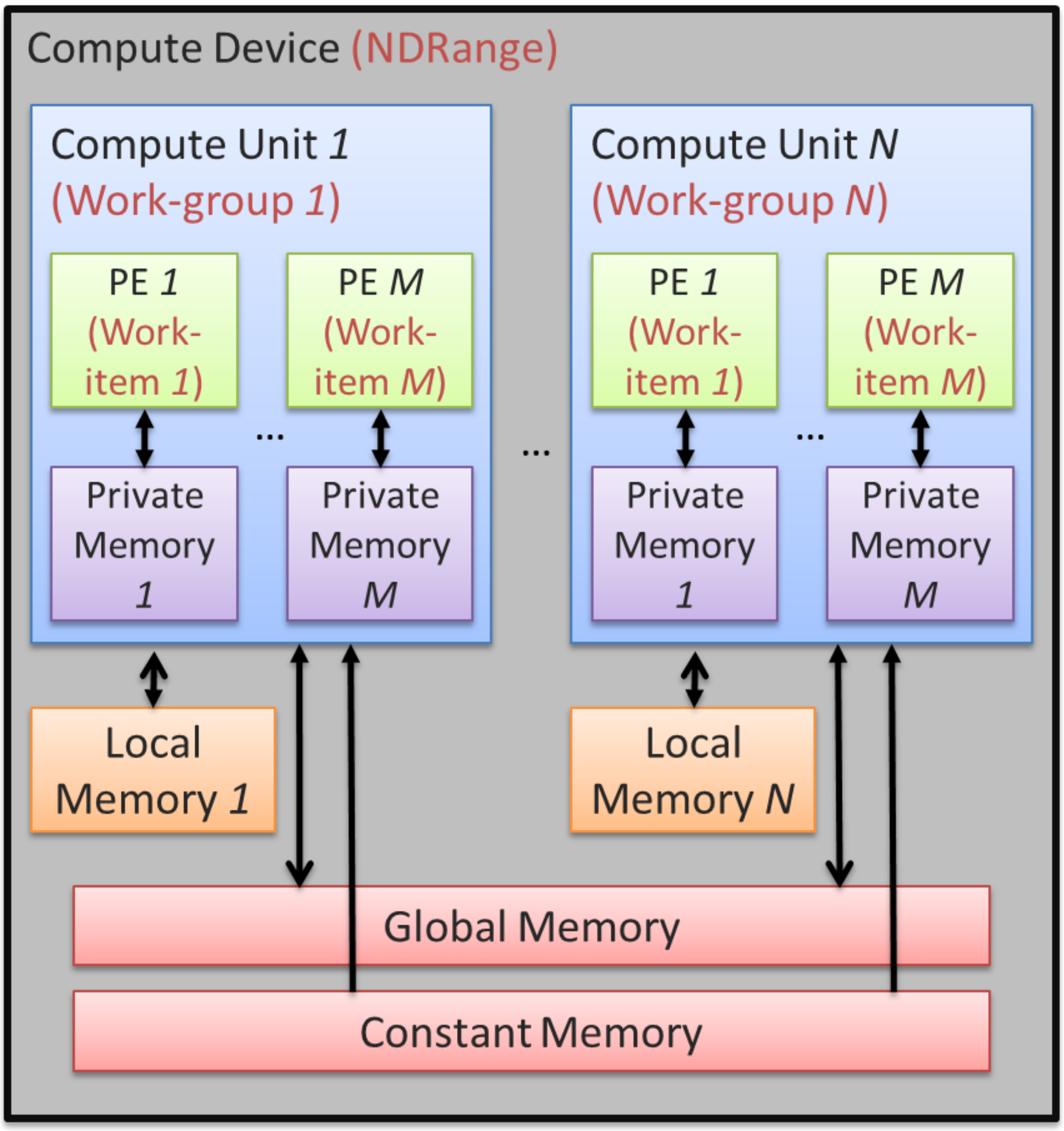


Figure 2-7 The Memory Model of OpenCL

2.2.4 Programming Model

The OpenCL execution model supports data parallel and task parallel programming models as well as the hybrids of these two models [6]. The primary model driving the design of OpenCL is data parallel programming model.

- Data Parallel Programming Model

  A data parallel programming model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items. In a strictly data parallel programming model, there is a one-to-one mapping between the work-item and the element in a memory object over which a kernel can be executed in parallel. However, OpenCL implements a relaxed version of the data parallel programming model where a strict one-to-one mapping is not a requirement.

- Task Parallel Programming Model

  The OpenCL task parallel programming model defines a model in which a single kernel instance is executed independently of any index space. It is logically equivalent to executing a kernel on a compute unit with a work-group containing a single work-item. Under this model, users may express parallelism by one of the following schemes.

  i. Using vector data types implemented by the device.

  ii. Enqueuing multiple kernels.

  iii. Enqueuing native kernels developed using a programming model other than that of OpenCL.

2.2.5 The OpenCL Framework

The framework inside OpenCL provides developers a way to develop an OpenCL program and manipulate the behavior of the program. The framework includes the following components:

- OpenCL Platform Layer

  The platform layer implements platform-specific features that allow programs to create contexts, manage contexts, query OpenCL devices, and query device configuration information.

- OpenCL Runtime

  The runtime supports numerous API calls that manage OpenCL objects such as command-queues, memory objects, program objects, and kernel objects for __kernel functions in a program object; it also supports API calls that allow you to enqueue commands to a command-queue for executing a kernel, and reading or writing a memory object.

- OpenCL Compiler

  The OpenCL standard defines a language for programming kernels as a subset of the ISO C99 language with extensions for parallelism. Therefore, it is necessary to have a compiler supporting such language, which creates program executables containing OpenCL kernels.

2.3 CUDA

NVIDIA's CUDA [7] is a parallel programming standard released in 2007. It extends the standard C/C++ programming language with some restrictions for parallel GPU computing. Via CUDA APIs, developers can program GPUs without tricky programming techniques which are required when developers programs GPUs with graphics APIs such as OpenGL and DirectX. Logically, CUDA could be viewed as a specialization of OpenCL for NVIDIA's GPUs in some extent because of many similarities between them. The similarities will be discussed in the following Subsections.

2.3.1 Platform Model

The platform model of CUDA, i.e., the CUDA-capable GPU architecture, is shown in Figure 2-8. One or more CUDA cores (CCs) compose a multiprocessor, and one or more multiprocessors compose a GPU which is connected to a CPU via the system bus. A CUDA core is a basic computation unit as a processing element in OpenCL, and a multiprocessor is analogous to a compute unit in OpenCL that both play the same role in CUDA and OpenCL respectively.

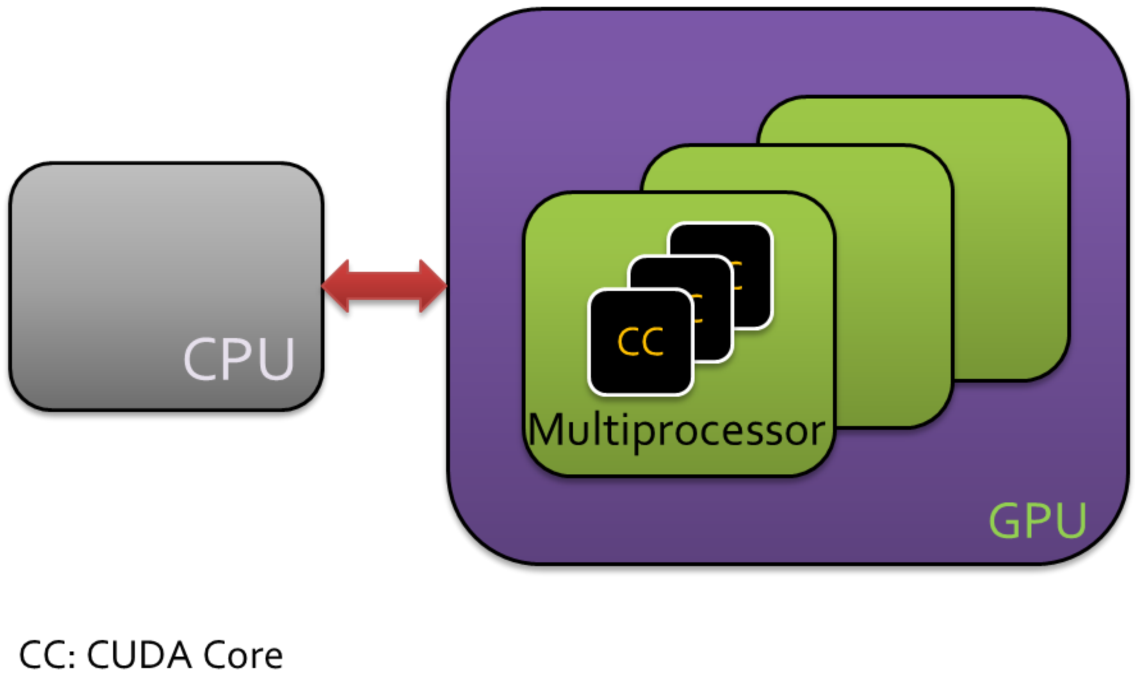


Figure 2-8 The CUDA-capable GPU architecture

2.3.2 Execution Model

The execution model or the thread hierarchy of CUDA as shown in Figure 2-9 is a grid of thread blocks, each of which contains a batch of threads. A grid in CUDA and an NDRange in OpenCL are alike that both of them are index space of basic computation units. However, a grid is two-dimensional, whereas an NDRange is three-dimensional. As expected, a thread block is just like a work-group in OpenCL, and a thread in CUDA plays the same role as a work-item in OpenCL

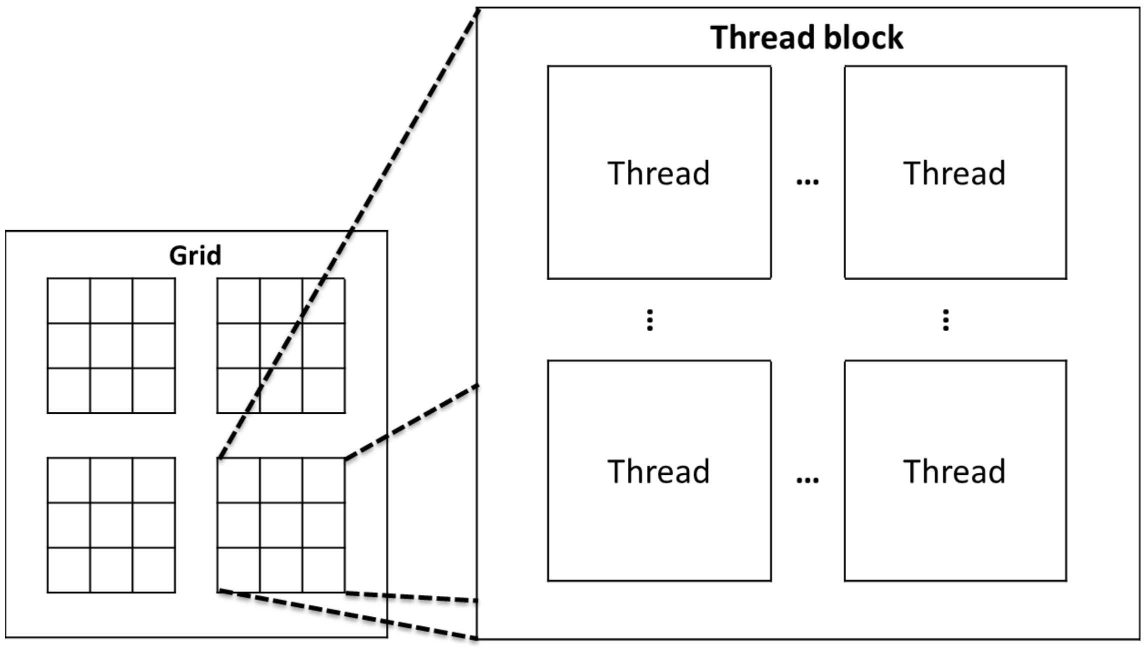


Figure 2-9 The Thread Hierarchy of CUDA

2.3.3 Memory Model

As mentioned in the first paragraph of Section 2.3 that CUDA could be viewed as a specialization of OpenCL, in contrast, OpenCL could be viewed as a generalization of CUDA. Thus, registers and local memory in CUDA are generalized as private memory in OpenCL which is a memory region per work-item. Shared memory in CUDA is generalized as local memory in OpenCL which is a memory region shared by a work-group. Likewise, global memory both in CUDA and OpenCL function alike. However, texture memory in CUDA which is a particular memory region for graphics processing is missing from OpenCL.

2.4 Related Work

Lee *et al* designed and implemented a translator for automatic translation of OpenMP programs into CUDA programs [8], which was released in Cetus [9] 1.2 on May 21, 2010. The translator consists of two phases as shown in Figure 2-10. Phase 1 contains an OpenMP optimizer. Phase 2 contains an O_2C baseline translator and a CUDA optimizer.

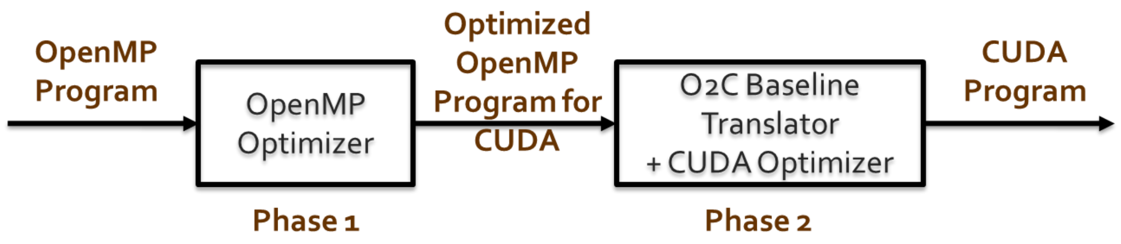

OpenMP Program
OpenMP Optimizer
Phase 1
Optimized OpenMP Program for CUDA
O2C Baseline Translator + CUDA Optimizer
Phase 2
CUDA Program

Figure 2-10 The Framework of The Related Work

First, the OpenMP optimizer optimizes the input OpenMP programs for CUDA. OpenMP mainly targets for homogeneous multiprocessor platforms with shared memory, such as a platform with single-core CPUs or multi-core CPU(s) sharing memory regions and having the same ISA. In contrast, CUDA targets for heterogeneous multiprocessor platforms that now can only be one or more CPUs plus one or more NVIDIA GPUs. Because of these architectural differences, the optimizer is used to alleviate the effect of the differences and to make the resulted OpenMP programs more suitable for executing on heterogeneous multiprocessor platforms that CUDA targets.

Second, the O_2C baseline translator translates the optimized OpenMP programs of the OpenMP optimizer into CUDA programs directly. The translation mainly consists of the following three steps:

i. Analyzing the OpenMP program under the models of CUDA.

ii. Identifying kernel regions, i.e., regions executed on GPUs.

iii. Transforming and outlining kernel regions into CUDA kernel functions

Finally, since the OpenMP optimizer only makes use of the constructs of the C programming language, which is the selected base language of the OpenMP programs, the CUDA optimizer optimizes the resulted CUDA programs of the O_2C baseline translator for performance by utilizing CUDA-specific features such as low-latency memory regions, coalesced global memory accesses, and persistent data in global memory.

Chapter 3 Design and Implementation of OMP2OCL Translator

Based on the source code of the related work, we designed and implemented OMP2OCL translator, which is a translator for automatic translation of OpenMP programs into OpenCL programs, according to the syntax, the semantics and the APIs of OpenCL. By the way, both of OpenMP programs and OpenCL programs are written in C programming language. The translator consists of two phases as shown in Figure 3-1. Phase 1 contains an OpenMP optimizer. Phase 2 contains an O2O baseline translator and an OpenCL optimizer. The details of these components will be discussed in the following sections.

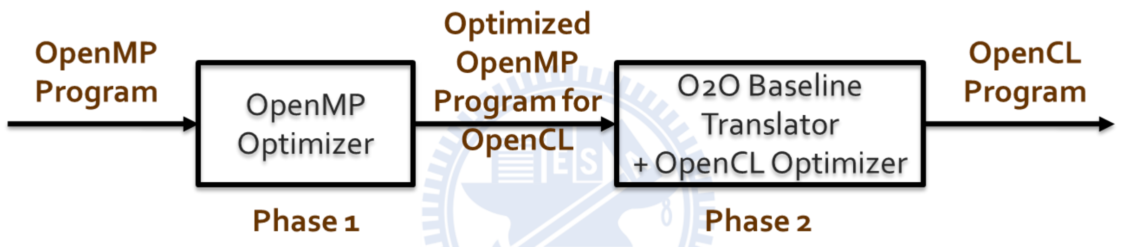

OpenMP Program
OpenMP Optimizer
Phase 1
Optimized OpenMP Program for OpenCL
O2O Baseline Translator + OpenCL Optimizer
Phase 2
OpenCL Program

Figure 3-1 The Framework of OMP2OCL translator

The example OpenMP program as shown in Figure 3-2 is used to demonstrate the concept of OMP2OCL.

```c
#include <stdio.h>
#define N 1024

int a[N][N], b;

int main()
{
      int i, j;

      #pragma omp parallel
      {
            #pragma omp for private(j)
            for (i = 0; i < N; i++) {
                  for (j = 0; j < N; j++) {
                        a[i][j] = b + 1;
                  }
            }

            #pragma omp single
            {
                  puts("XD");
            }
      }

      return 0;
}
```

Figure 3-2 The Example OpenMP Program

3.1 OpenMP Optimizer

The OpenMP optimizer optimizes the input OpenMP programs for OpenCL. With the reason that OpenMP and OpenCL target for different multiprocessor platforms, which is similar to the reason mentioned in the second paragraph of Section 2.4, the optimizer is used to alleviate the effect of the architectural differences and to make the resulted OpenMP programs more suitable for executing on heterogeneous multiprocessor platforms that OpenCL supports.

Concerning spatial data locality, CPUs, target devices of OpenMP, prefer intra-thread locality to inter-thread locality; whereas GPUs, devices supported by OpenCL, prefer both localities. In CPUs, intra-thread locality increases cache utilization, thus reducing memory access latency. However, inter-thread locality among threads on different CPUs of homogeneous multiprocessor platforms with shared memory may encounter false-sharing and hence increase memory access latency. Therefore, iterations of *omp for* loops in OpenMP programs are usually block-distributed rather than cycle-distributed. In GPUs, intra-thread locality within a thread also increases cache utilization. But most importantly, inter-thread locality among GPU threads has great effect on global memory access latency; that is, if adjacent threads concurrently access to adjacent data in global memory region, these accesses can be coalesced into fewer global memory accesses, and consequently, the memory access latencies are reduced.

This optimizer provides one optimization, Parallel Loop-Swap, to enhance inter-thread locality of OpenMP programs for executing on heterogeneous multiprocessor platforms with GPUs as OpenCL devices. The optimization can be turn on and turn off by the command line options of OMP2OCL translator.

3.1.1 Parallel Loop-Swap

Parallel Loop-Swap is an optimization to enhance the inter-thread locality of OpenMP programs with regular data accesses in loop nests. For each perfect loop nest that has multiple loops and contains no function call, the optimization rearranges the loops within the loop nest using following strategies:

i. For each loop with index variable which has less effect on values of subscript expressions of array accesses within the loop body, move it to outer level as far as possible.

ii. For each loop with index variable which has larger effect on values of subscript expressions of array accesses within the loop body, move it to inner level as far as possible.

iii. The rearrangement of loops within a loop nest should not break the dependencies among initial statements, condition expressions and step expressions of loops.

iv. For points i and ii, when there are ties among loops which have the same effect on values of subscript expressions of array accesses within the loop body, the loop with more iterations has higher priority to be rearranged.

A simple loop nest without applying Parallel Loop-Swap is shown in Figure 3-3. After the loop nest with the associated *omp parallel* is translated into a kernel function, a region of the kernel function as shown in Figure 3-4 contains an *if* statement, which in turn contains a singly nested *for* loop. The *if* statement is derived from L_i, the loop with index variable `i` in

Figure 3-3, and the *for* loop comes from L_j, the loop with index variable `j` in Figure 3-3. In Figure 3-4, each GPU thread accesses a row of each array via the array accesses `a[wiid][j]` and `b[wiid][j]`, and, as a consequence, these memory accesses create uncoalesced global memory accesses, and thus increase memory access latencies.

Array accesses `a[i][j]` and `b[i][j]` could be expressed as `a[i * N + j]` and `b[i * N + j]`, respectively. For expression `i * N + j`, increment of `i` will increases the value of the expression more than that of `j`. For this reason, in Figure 3-3, the index variable of L_j, j, has less effect on the values of the subscript expressions of the array accesses `a[i][j]` and `b[i][j]` than the index variable of L_i, i. Therefore, L_j is moved up to the outer level of the loop nest in Figure 3-5 after applying Parallel Loop-Swap. In Figure 3-6, each GPU thread accesses a column of each array via the array accesses `a[i][wiid]` and `b[i][wiid]`, and at every iteration of L_i, the accesses of all GPU threads fall within a row of each array. Accordingly, these memory accesses can be coalesced into fewer global memory accesses, and the memory access latencies are reduced.

```
	#pragma omp parallel
	{
		#pragma omp for private(j)
		for (i = 0; i < N; i++) {
			for (j = 0; j < N; j++) {
				a[i][j] = b + 1;
			}
		}
```

Figure 3-3 A Simple Loop Nest Without Applying Parallel Loop-Swap

```
if (wiid < N) {
	for (j = 0; j < N; j++) {
		a[wiid][j] = b[wiid][j] + 1;
	}
}
```

Figure 3-4 A Region of the Kernel Function Transformed from Figure 3-3

```
        #pragma omp parallel
        {
                #pragma omp for private(j)
                for (j = 0; j < N; j++) {
                        for (i = 0; i < N; i++) {
                                a[i][j] = b + 1;
                        }
                }
```

Figure 3-5 The Simple Loop Nest After Applying Parallel Loop-Swap

```
if (wiid < N) {
        for (i = 0; i < N; i++) {
                a[i][wiid] = b[i][wiid] + 1;
        }
}
```

Figure 3-6 The Region of the Kernel Function Transformed from Figure 3-5

OpenMP is the source parallel programming standard of OMP2OCL translator, and Parallel Loop-Swap transforms the input OpenMP program into another OpenMP program. With the O2O baseline translator in Phase 2, the optimized loop nest with the associated *omp parallel* will be translated into an OpenCL kernel function.

3.2 O2O Baseline Translator

The O2O baseline translator translates the possibly optimized OpenMP programs by the OpenMP optimizer in Phase 1 into OpenCL programs directly. The translation mainly consists of the following three steps:

i. Analyzing the OpenMP program under the models of OpenCL.

ii. Identifying kernel regions, i.e., regions executed on OpenCL devices.

iii. Transforming and outlining kernel regions into OpenCL kernel functions.

The execution model of OpenMP is a fork-join model, and the translated OpenCL program has two parts: the host code and the kernels for the device. The serial region of OpenMP is mapped to the host code execution, and the parallel regions of OpenMP are mapped to the kernel executions. In addition, the memory model of OpenMP is shared memory with each thread having its own private memory. Accordingly, *omp shared* data are mapped into global memory region of OpenCL, whereas *omp private* data are mapped into private memory regions of OpenCL.

3.2.1 OpenMP Program Analysis

As mentioned in Section 2.1, directives are at the core of OpenMP standard. So in this Subsection, the discussion is focused on the interpretation of the directives and the data-sharing attribute clauses.

For *parallel* directive, since it is used to annotate structured blocks that could be paralleled, the translator identifies these structured blocks as candidate kernel regions, outlines them from the original functions, transforms them into OpenCL kernel functions, and inserts kernel execution command calls into the position where the structured blocks reside originally.

With respect to work-sharing directives, which are used to annotate works that could be divided and distributed to threads, there are *for* directive, *sections* directive, and *single* directive specified in OpenMP specification [10]. Each iteration of an *omp for* loop is mapped to an OpenCL work-item, and accesses to the index variable of the *omp for* loop are replaced by accesses to the work-item id variable. Each section of an *omp sections* is assigned to an OpenCL work-item. *Omp single* structured block is left on the host and executed by one host thread in current implementation.

Concerning combined *parallel* work-sharing directives, they could be expressed as a *parallel* directive which annotates a structured block where a work-sharing directive and its associated region are contained. As a result, they can be mapped by the principles mentioned above.

About synchronization directives, they create split points where a candidate kernel region must be split into two sub-regions. If the resulted two sub-regions become kernel regions, and after the two kernel regions are outlined and transformed, the generated two kernel execution command calls ensure global synchronization among work-items. The split is necessary, because there is no global synchronization mechanism among work-items executing in an OpenCL device.

Finally, with regard to data-sharing attribute clauses and threadprivate directive, they are used to map data into OpenCL memory regions. OpenMP *shared* data are shared by all threads, they are copied to variables in global memory region for being used by all work-items after an OpenMP program is translated into an OpenCL program. OpenMP *private* data are private to each thread, and, therefore, they are mapped to variables in private memory regions. *Threadprivate* data specified by threadprivate directives are also private to each thread with static lifetime, that is, *threadprivate* data live across *parallel* regions, so they are expanded in variables in global memory region. For variables not explicitly determined, which are not listed in data-sharing attribute clauses, the translator appends them into data-sharing attribute clauses of appropriate directives. For instance, as shown in Figure 3-7, a simple loop nest has only one data-sharing attribute clause in line 12, and variables `a`, `b`, and `i` are not explicitly determined. For ease of translation, the translator analyzes the program and appends the three variables into data-sharing attribute clauses of the appropriate directives as shown in Figure 3-8. Variables `a` and `b` are static and in file scope so that they are *shared* in the *parallel* region. Variable `i` is the index variable of the loop Li, and, accordingly, it is appended in the *private* clause in line 12. Since the *parallel* directive in line 10 annotates the structured block where only one *for* directive and its associated *for* loop are contained, the data-sharing attribute clauses of the *parallel* directive include those of the *for* directive; thus, `private(i, j)` is included in both directives.

```
 4 int a[N][N], b;
 5
 6 int main()
 7 {
 8         int i, j;
 9
10         #pragma omp parallel
11         {
12                 #pragma omp for private(j)
13                 for (i = 0; i < N; i++) {
14                         for (j = 0; j < N; j++) {
15                                 a[i][j] = b + 1;
16                         }
17                 }
```

Figure 3-7 A Simple Example with A Simple Loop Nest

```c
#pragma omp parallel shared(a, b) private(i, j)
{
        #pragma omp for private(i, j)
        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        a[i][j] = b + 1;
                }
        }
```

Figure 3-8 The Simple Loop Nest with Additional Data-Sharing Attribute Clauses

This step analyzes the semantics of the input OpenMP program. With kernel regions transformation and outlining of the O2O baseline translator, for instance, *shared* data analyzed in this step will be copied into an OpenCL buffer object for the device before a kernel execution by OpenCL API. After the kernel execution, the shared data will be copied from the OpenCL buffer object back into main memory by the API.

3.2.2 Kernel Regions Identification

The translator targets OpenMP *parallel* regions as candidate kernel regions. As mentioned in the fifth paragraph of Subsection 3.2.1, synchronization directives result in split points, where a candidate kernel region must be split into two sub-regions in order to ensure global synchronization. In addition, at entry to and exit from a *parallel* region, and at exit from a work-sharing region, there are implicit *flush* synchronization directives. Moreover, most OpenMP directives annotate on a structured block which is an executable statement and possibly compound, with a single entry at the top and a single exit at the bottom. The translator must consider that the splits should not break the control flow semantics of OpenMP; in other words, the split points should not locate in the middle of a structured block to prevent creating unstructured blocks.

The overall algorithm derived from the related work [8] for identifying kernel regions is shown in Figure 3-9. This top-down split approach is for splitting candidate kernel regions as less as possible.

```
Input: R        /* a set of OpenMP parallel regions */

Output: KR      /* a set of identified GPU kernel regions */


Foreach R(i) in R

      Foreach split point in R(i)

            Divide R(i) into two sub-regions at the split point

      Build CFG, a control flow graph for R(i)

      Foreach sub-region SR(j) in R(i)

            Foreach entry/exit points other than the one at the
            top/bottom of SR(j)

                  Divide SR(j) into two sub-sub-regions at such
                  entry/exit points

            Foreach sub-sub-region SSR(k) in SR(j)

                  If SSR(k) contains an OpenMP work-sharing
                  directive
```

Figure 3-9 The Algorithm for Identifying Kernel Regions

The *parallel* region as shown in Figure 3-10 has four split points, which are shown by dash lines. The first and the last split points are at entry to and exit from the *parallel* region respectively, the second and the third split points are at exit from work-sharing regions. After splitting, the *parallel* region is split into two *parallel* regions as shown in Figure 3-11. The first region will be translated into an OpenCL kernel function, whereas the second region will not due to lack of global synchronization mechanism among work-items in OpenCL.

```
-----------------------------------------------
#pragma omp parallel shared(a, b) private(i, j)
{
        #pragma omp for private(i, j)
        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        a[i][j] = b + 1;
                }
        }
-----------------------------------------------

        #pragma omp single
        {
                puts("XD");
        }
-----------------------------------------------
}
-----------------------------------------------
```

Figure 3-10 The *Parallel* Region with Split Points Shown by Dash Lines

```
#pragma omp parallel shared(a, b) private(i, j)
{
        #pragma omp for private(i, j)
        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        a[i][j] = b + 1;
                }
        }
}


#pragma omp parallel
{
        #pragma omp single
        {
                puts("XD");
        }
}
```

Figure 3-11 The *Parallel* Regions After Splitting

3.2.3 Kernel Regions Transformation and Outlining

The kernel regions identified should be transformed before outlined into OpenCL kernel functions. The transformation includes two stages: work partitioning and data mapping. Firstly, for work-partitioning, each iteration of an *omp for* loop is mapped to an OpenCL work-item, and each section of an *omp sections* is assigned to an OpenCL work-item. As a consequence, the total number of work-items could be calculated by the number of iterations for an *omp for* loop or the number of sections for an *omp sections*. Secondly, for data mapping, using the interpretation results from Subsection 3.2.1 that all variables referenced in a kernel region are listed in data-sharing attribute clauses, the translator inserts memory transfer command calls to read data from or write data to OpenCL buffer objects for *shared* and *threadprivate* data (*threadprivate* data are transferred only when the corresponding variables are listed in *copyin* clauses of *parallel* directives). A basic strategy is to copy data used by kernel functions from main memory into memory objects and copy modified data from memory objects back to main memory. However, not all memory transfer command calls are necessary, since the modified data might not be used by the host and data in global memory region are persistent across kernel executions. The optimization for reducing memory transfer command calls will be discussed in Subsection 3.3.2.

After being transformed, the kernel regions are outlined and replaced with kernel execution command calls.

When the first *parallel* region in Figure 3-11 is transformed and outlined, the resulted OpenCL kernel function is shown in Figure 3-12. Variables `a` and `b` are *shared* so that they are mapped to OpenCL global memory region, and before the kernel execution, they are copied into an OpenCL buffer object which is passed as an argument to the kernel function. Variables `i` and `j` are *private* so that they are mapped to OpenCL private memory regions, and, accordingly, they are defined as local variables in the kernel function. For line 7 to line 11 in Figure 3-12, each work-item does an iteration of the *omp for* loop L_i in Figure 3-11.

```
     1 __kernel void main_kernel0(__global int a[N][N], __global int
*b)
     2 {
     3        int i;
     4        int j;
     5        i = wiid;
     6        ...;
     7        ...;
     8        if (i < N) {
     9              for (j = 0; j < N; j++) {
    10                    a[i][j] = *b + 1;
    11              }
    12        }
    13        ...;
    14        ...;
    15 }
```

Figure 3-12 The OpenCL Kernel Function Translated From the First *Parallel* Region in Figure 3-11

API and the extended C language for kernel functions of CUDA are different from those of OpenCL with some similarities, so some changes are necessary. For instance, OpenCL kernel functions must be in a string to be compiled by *clBulidProgram*, so the transformed and outlined kernel functions must be appended into a string. In addition, the OpenCL kernel functions in the string cannot access information such as macros in the host code, and, consequently, the necessary information must be also included in the string.

3.3 OpenCL Optimizer

Since the OpenMP optimizer in Phase 1 only makes use of the constructs of the C programming language, which is the selected base language of the OpenMP programs, the OpenCL optimizer optimizes the resulted OpenCL programs of O2O baseline translator for performance by utilizing features of OpenCL and some OpenCL devices like low-latency memory regions and persistent data in global memory.

This optimizer provides three optimizations, which are caching frequently accessed global data and memory transfer reduction. The optimizations can be turn on and turn off by the command line options of OMP2OCL translator.

3.3.1 Caching Frequently Accessed Global Data

In some OpenCL devices such as GPUs, global memory region is mapped to an off-chip memory which might not has cache depending on the capabilities of the OpenCL device, and other memory regions such as private memory region, local memory region, and constant memory region are either mapped to an on-chip memory or equipped with caches. In addition, an on-chip memory usually has higher bandwidth and lower latency than an off-chip memory for the same device. Hence, in order to reduce memory access latency, frequently accessed data in global memory region should be cached on low-latency memory regions.

Concerning temporal data locality, the hardware automatically exploits intra-thread locality and inter-thread locality on CPUs. Whereas using OpenCL on GPUs, the software must exploit both intra-thread locality and inter-thread locality (or intra-work-item and inter-work-item) by itself for global memory region.

This optimization performs data flow analysis to identify temporal locality of global data, and inserts necessary caching codes. The caching strategies for global data with different attributes are shown in Table 3-1. The O2O baseline translator maps *shared* data into global memory region; however, this optimization caches global data into different memory regions according to their attributes.

Table 3-1 The Caching Strategies for Global Data with Different Attributes

	Temporal Locality	
	Intra-work-item	**Inter-work-item**
R/O scalar	Private, Local	Local
R/W scalar	Private, Local	Local
R/O array	Local	Local
R/W array	Local	Local

In Figure 3-12, the data pointed by the variable `b` belongs to read-only scalar global data, and work-items share the data. Therefore, variable `b` will be cached in the local memory regions as shown in Figure 3-13. A variable declaration locates in line 5 with `__local` specifier, which is used to indicate that the variable is resided in the local memory regions. In line 7, the data pointed by the variable `b` is cached into the local memory regions. In line 12, the access to the data in the local memory regions substitutes the access to the global data.

```
__kernel void main_kernel0(__global int a[N][N], __global int *b)
{
        int i;
        int j;
        __local int lo__b;
        i = wiid;
        lo__b = *b;
        ...;
        ...;
        if (i < N) {
              for (j = 0; j < N; j++) {
                    a[i][j] = lo__b + 1;
              }
        }
        ...;
        ...;
}
```

Figure 3-13 The OpenCL Kernel Function with Variable `b` Cached in the Local Memory Regions

As mentioned in Subsection 2.2.3, shared memory in CUDA is generalized as local memory in OpenCL which is shared by a work-group. In CUDA, passing the data as an argument to a kernel function has an effect of caching the data into shared memory regions. Nevertheless, in OpenCL, to cache the data into local memory regions, it is necessary to insert a variable declaration and an assignment statement as shown in Figure 3-13 since parameters of a kernel function with specifier `__local` cannot be initialed from the host.

3.3.2 Memory Transfer Reduction

An important stage of transforming kernel regions into OpenCL kernel functions is the insertion of memory transfer command call. The O2O baseline translator inserts memory transfer calls for all *shared* and *threadprivate* data. However, not all memory transfer command calls are necessary since the modified data might not be used by the host and since data in global memory region are persistent across kernel executions.

To remove the unnecessary memory transfer command calls, this optimization performs *data flow analysis* in four steps:

i. Finding a set of *shared* data read in the kernel region as *UseSet*.

ii. Finding a set of *shared* data written in the kernel region as *DefSet*.

iii. For each variable in *UseSet*, if its reaching definition locates outside the kernel region, then the variable should be transferred from the host before the kernel execution.

iv. For each variable in *DefSet*, if it is used outside the kernel region, then the variable should be transferred back to the host after the kernel execution.

In Figure 3-12, after the kernel execution, the host transfers the data pointed by variable `a` and `b` back into the main memory. With this optimization, these memory transfers are removed because the data will be useless after the kernel execution.

3.4 Limitations of OMP2OCL Translator

Some limitations of OMP2OCL translator are listed as follows:

- OMP2OCL translator assumes that there is only one device in the platform and only one kernel executes at a time.
- OMP2OCL translator can only translate simple OpenMP programs now. For instance, it does not support *omp task*.
- OMP2OCL translator has three optimizations now. For the performances of the translated OpenCL programs, it is necessary to develop more optimizations for OMP2OCL translator.

Chapter 4 Experiment

The experiment platform has one Intel Core 2 Quad Q6600 as the host and one NVIDIA GeForce 9800 GT as the device. Using oclBandwidthTest from NVIDIA GPU Computing SDK, the bandwidth from the host to the device is around 2330 MB/s, and the bandwidth from the device to the host is about 1900 MB/s.

Our experiment is based on two important kernels (JACOBI and SPMUL) and one NAS OpenMP Parallel Benchmark (EP). Table 4-1 describes these three benchmarks, sorted from top to down according to the increasing order of the numbers of source lines. The following sections present the experiment results with respect to each benchmark.

Table 4-1 Descriptions of the Benchmarks

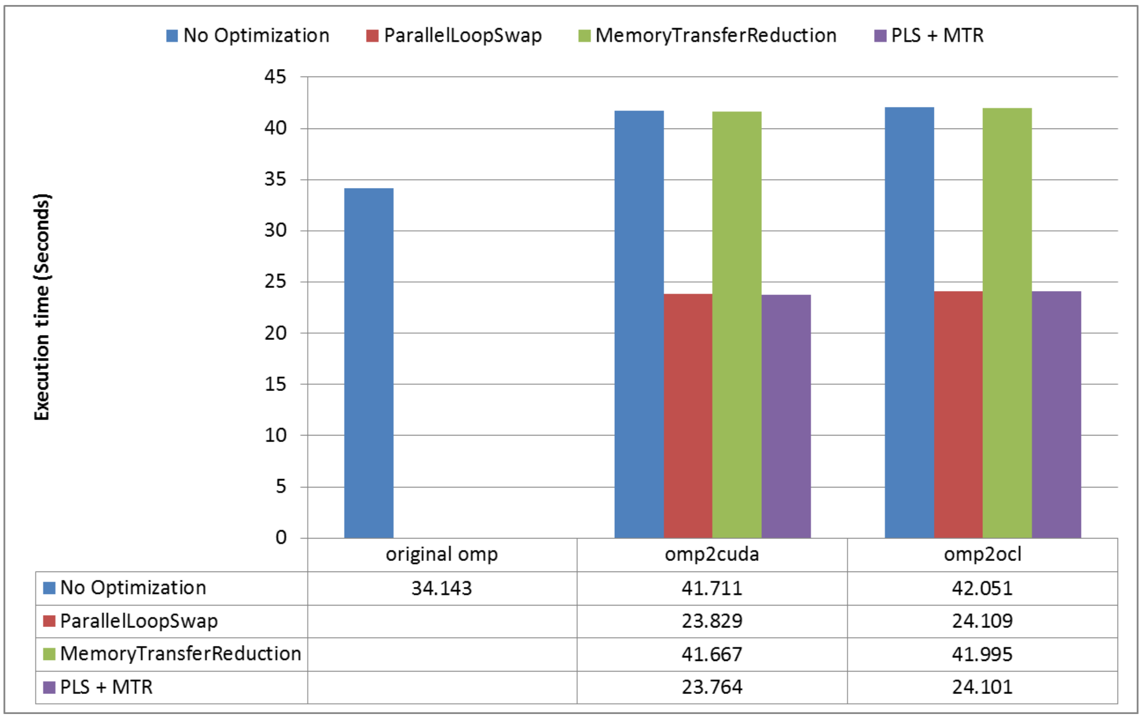
Benchmark	Description	# of Source Lines	# of Kernel Lines	# of Omp *Parallel*
JACOBI	An algorithm for determining the solutions of a system of linear equations with largest absolute values in each row and column dominated by the diagonal elements	92	14	2
SPMUL	Sparse matrix multiplication, which is an algorithm for sparse matrices, where most elements of the matrices are zero	250	44	1
EP	Embarrassingly parallel, which is an algorithm for generating independent Gaussian random variates using the Marsaglia polar method	614	79	1

Table 4-2 describes five optimizations implemented by the related work and the three optimizations included in OMP2OCL translator. As a result, three optimizations, Parallel Loop-Swap (PLS), Caching of Frequently Accessed Global Data (Caching), and Memory Transfer Reduction (MTR) are conducted in the experiments.

Table 4-2 Optimizations Implemented by the Related Work and OMP2OCL Translator

	The Related Work	OMP2OCL Translator
Parallel Loop-Swap (PLS)	√	√
Loop Collapsing (LC)	√	
Caching of Frequently Accessed Global Data (Caching)	√	√
Matrix Transpose of *Threadprivate* Array (MT)	√	
Memory Transfer Reduction (MTR)	√	√

4.1 JACOBI



	original omp	omp2cuda	omp2ocl
🟦 No Optimization	34.143	41.711	42.051
🟥 ParallelLoopSwap		23.829	24.109
🟩 MemoryTransferReduction		41.667	41.995
🟪 PLS + MTR		23.764	24.101

Figure 4-1 Experiment Results for JACOBI

Without any optimization, both the CUDA program and the OpenCL program output by the related work and OMP2OCL translator, respectively, perform worse than the original OpenMP program because of the uncoalesced global memory accesses in the kernel functions. The uncoalesced global memory accesses are resulted from the fact that each work-item accesses arrays in row-wise scheme. Accordingly, Parallel Loop-Swap in Phase 1 can be used to overcome these situations.

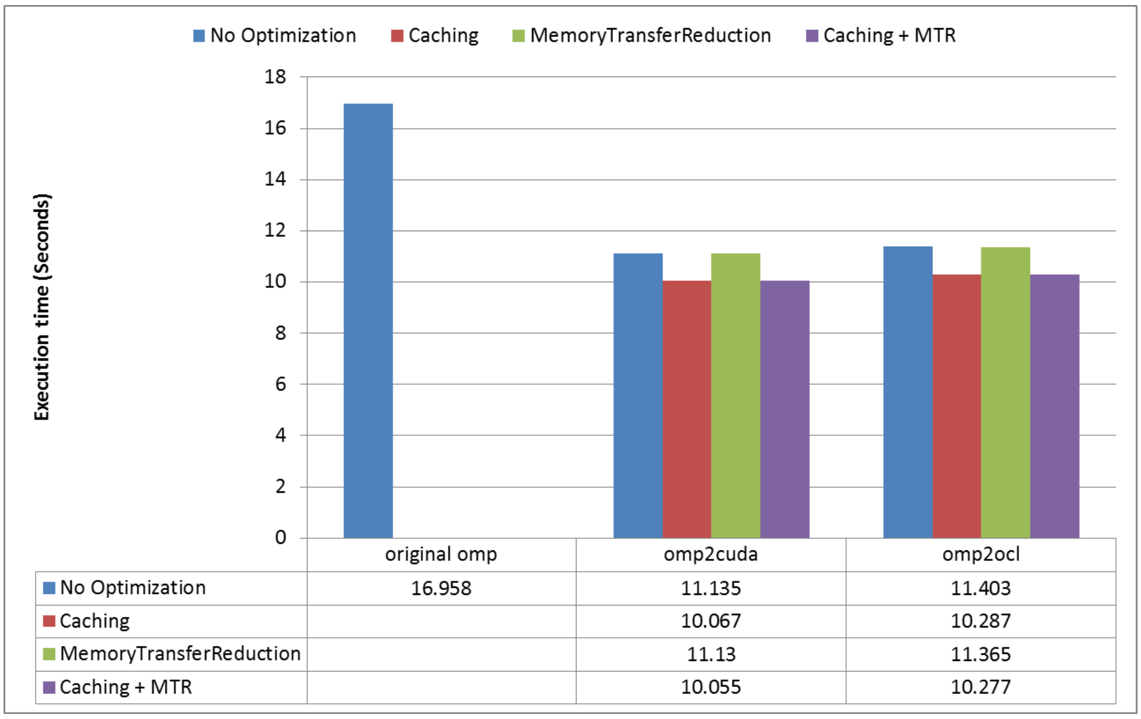
Using Parallel Loop-Swap in Phase 1, the performances of both the CUDA program and the OpenCL program increase substantially and their execution time are around 57.13% and 57.33% compared to the OpenMP program and the OpenCL program with no optimization respectively.

Using Memory Transfer Reduction in Phase 2, the performances of both the CUDA program and the OpenCL program improve just very little due to the small amount of memory transfers between the host and the device. Before optimization, the sizes of memory transfers are around 32 MB for each direction. After optimization, the size of memory transfers is reduced to around 16 MB for the direction from the host to the device, whereas the size of memory transfer is remained the same for the direction from the device to the host.

Using both Parallel Loop-Swap in Phase 1 and Memory Transfer Reduction in Phase 2, the combined effect of the optimizations are as expected that the execution time of both the CUDA program and the OpenCL program are the shortest in programs output by the translator of the related work and OMP2OCL translator respectively.

For JACOBI, Caching Frequently Accessed Global Data in Phase 2 has no effect on both the CUDA program and the OpenCL program, since there are no temporal data locality regarding to global memory accesses .Accordingly, the execution times after applying this optimization are not showed in Figure 4-1.

4.2 SPMUL



	original omp	omp2cuda	omp2ocl
🟦 No Optimization	16.958	11.135	11.403
🟥 Caching		10.067	10.287
🟩 MemoryTransferReduction		11.13	11.365
🟪 Caching + MTR		10.055	10.277

Figure 4-2 Experiment Results for SPMUL

Without any optimization, both the CUDA program and the OpenCL program output by the related work and OMP2OCL translator, respectively, perform better than the original OpenMP program because of the little uncoalesced global memory accesses and the large amount of computation.

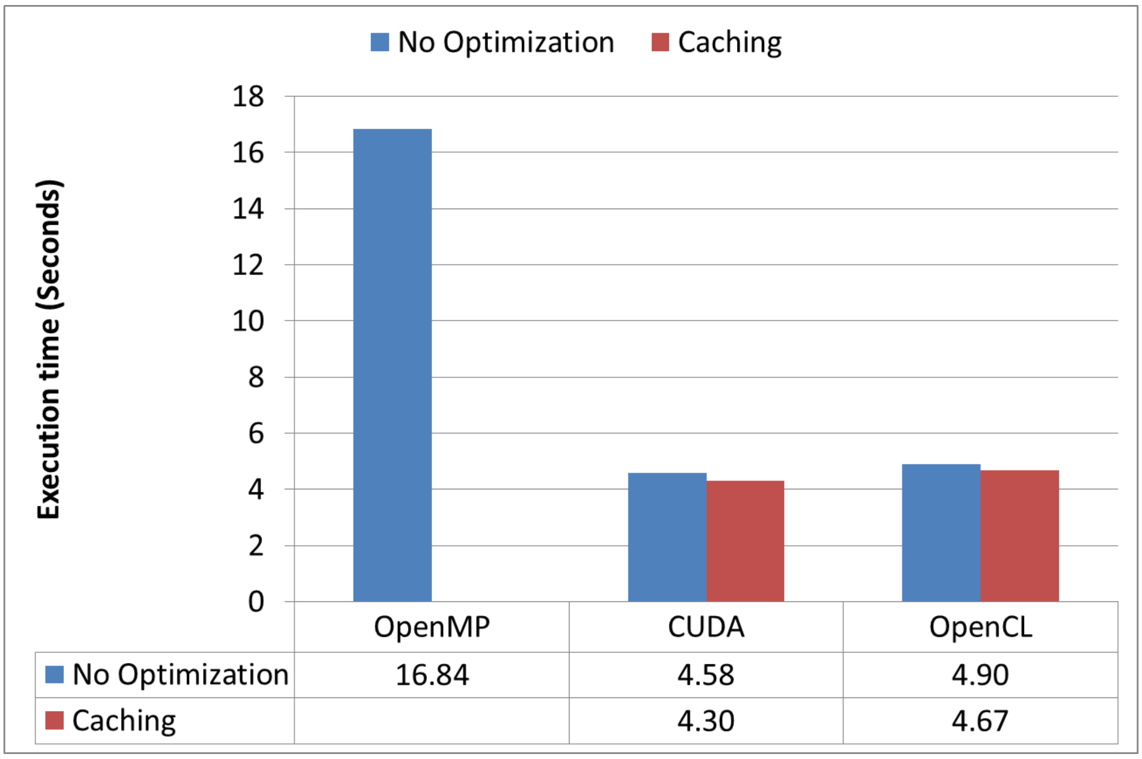
By Caching Frequently Accessed Global Data in Phase 2, some array elements are cached in registers and private memories of the CUDA program and the OpenCL program, respectively. As a consequence, the memory accesses to global memory region are reduced, and, thus the performances of both the CUDA program and the OpenCL program increase a few.

Using Memory Transfer Reduction in Phase 2, the performances of both the CUDA program and the OpenCL program just increase very little due to the small amount of memory transfer reduction. Before optimization, the sizes of memory transfers are around 85.64 MB from the host to the device and about 7.87 MB from the device to the host. After optimization, the size of memory transfers is reduced to around 77.77 MB from the host to the device, whereas the size of memory transfer is remained the same from the device to the host.

Using both Caching Frequently Accessed Global Data in Phase 2 and Memory Transfer Reduction in Phase 2, the combined effect of the optimizations are as expected that the execution time of both the CUDA program and the OpenCL program are the shortest in programs output by the related work and OMP2OCL translator respectively.

For SPMUL, Parallel Loop-Swap in Phase 1 has no effect because suitable loop nest described in Subsection 3.1.1 for optimization does not exist, and, as a consequence, the execution times after applying this optimization are not showed in Figure 4-2.

4.3 EP



	OpenMP	CUDA	OpenCL
🟦 No Optimization	16.84	4.58	4.90
🟥 Caching		4.30	4.67

Figure 4-3 Experiment Results for EP

Without any optimization, both the CUDA program and the OpenCL program output by the related work and OMP2OCL translator, respectively, perform better than the original OpenMP program because of the characteristics of EP, which has little uncoalesced global memory accesses and large amount of parallelizable computation.

By Caching Frequently Accessed Global Data in Phase 2, some array elements are cached in shared memories and local memories of the CUDA program and the OpenCL program, respectively. As a consequence, the memory accesses to global memory region are reduced, and, thus the performances of both the CUDA program and the OpenCL program increase a few.

For EP, Parallel Loop-Swap in Phase 1 and Memory Transfer Reduction in Phase 2 have no effect on both the CUDA program and the OpenCL program; in other words, there is no suitable loop nest for Parallel Loop-Swap and no suitable global data for Memory Transfer Reduction. Consequently, the execution times after applying this optimization are not showed in Figure 4-3.

4.4 Discussion

Parallel Loop-Swap in Phase 1 resolves uncoalesced global memory accesses in a loop nest with regular data accesses. Coalesced global memory accesses are significant in both CUDA programs and OpenCL programs with GPUs as the devices. In our experiment platform, there is a GPU. Therefore, for JACOBI, both the CUDA program and the OpenCL program while applying this optimization perform well compared to those without this optimization and the original OpenMP program. However, with devices rather than GPUs such as CPUs, this optimization might decrease the performance instead.

In the experiment platform, caching frequently accessed global data in private memories and local memories of OpenCL has benefits to the performance of both the CUDA program and the OpenCL program, because the latencies to private memories and local memories are shorter than that to global memory. Nevertheless, with devices rather than GPUs such as CPUs, the latency reduction by Caching might not be significant.

Memory Transfer Reduction in Phase 2 removes unnecessary memory transfers. If the memory transfers locate in a loop nest with many iterations, then the performance improvement could be substantial. However, this situation is not present in the experiments. Thus, for JACOBI and SPMUL, the performances of both the CUDA program and the OpenCL program while applying this optimization improve a little compared to those without applying this optimization.

Chapter 5 Conclusion and Future Work

We designed and implemented a translator for automatic translation of OpenMP programs into OpenCL programs. Although there are performance losses for OMP2OCL translator compared to the related work designed dedicatedly for NVIDIA GPUs, it is still promising that the translated OpenCL programs can use other devices as compute devices other than NVIDIA GPUs. In addition, we have constructed an infrastructure for automatic translation of OpenMP programs into OpenCL programs. Researchers can use this infrastructure as a basis to develop more device-independent and device-dependent optimizations. Moreover, programmers can use this translator to translate existed or newly-developed OpenMP programs into OpenCL programs, in order to make the OpenMP programs as programs of heterogeneous multiprocessor platforms.

The future work listed in the following might be considered to extend our OMP2OCL translator:

- Currently, our OMP2OCL translator assumes that there is only one device in the platform and only one kernel executes at a time. Whenever there are multiple devices in the platform and multiple kernels in the OpenCL program, these kernels have the potential to execute simultaneously. However, there are some conditions for the simultaneous execution, i.e., two kernels executed simultaneously should not have data and control dependencies between each other; otherwise, the translated OpenCL program would have different behaviors from the original OpenMP program. We plan to do necessary dependency analysis on kernels and let independent kernels execute on multiple devices in the future.

- Our OMP2OCL translator can translate only simple OpenMP programs now. For instance, it does not support *omp task*. We plan to map *omp task* into task parallel programming model of OpenCL in the future.

- OMP2OCL translator has three optimizations now. For the performances of the translated OpenCL programs, it is necessary to develop more optimizations for OMP2OCL translator. We plan to analyze and port Loop Collapsing and Matrix Transpose of the related work into OMP2OCL translator in the future.

References

[1] L. Hammond, B. A. Nayfeh, and K. Olukotun, “A Single-Chip Multiprocessor,” *Computer*, vol. 30, no. 9, pp. 79-85, 1997.

[2] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, S. Miki, and S. Tagawa, *The OpenCL Programming Book*, 1st ed. Fixstars Corporation, 2010.

[3] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46-55, 1998.

[4] W. Michael, “OpenMP on Accelerators—A Position Paper.” [Online]. Available: http://www.pgroup.com/lit/articles/insider/v2n2a5.htm.

[5] Khronos OpenCL Working Group, “OpenCL.” [Online]. Available: http://www.khronos.org/opencl/.

[6] Khronos OpenCL Working Group, *The OpenCL Specification 1.1*. 2010.

[7] NVIDIA Corporation, “CUDA Zone.” [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html.

[8] S. Lee, S. Min, and R. Eigenmann, “OpenMP to GPGPU: a compiler framework for automatic translation and optimization,” *SIGPLAN Not.*, vol. 44, no. 4, pp. 101-110, 2009.

[9] C. Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, R. Eigenmann, and S. Midkiff, “Cetus: A Source-to-Source Compiler Infrastructure for Multicores,” *Computer*, vol. 42, no. 12, pp. 36-42, 2009.

[10] OpenMP Architecture Review Board, *OpenMP 3.0 API Specifications - OpenMP Application Program Interface*. .