저작자표시-비영리-변경금지 2.0 대한민국

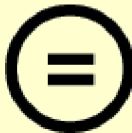
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사학위논문

Techniques for Ease of OpenCL Programming

OpenCL의 프로그래밍 용이성 향상 기법

2016년 2월

서울대학교 대학원

컴퓨터공학부

김 정 현

Abstract

OpenCL is one of the major programming models for heterogeneous systems. This thesis presents two limitations of OpenCL, the complicated nature of programming in OpenCL and the lack of support for a heterogeneous cluster, and proposes a solution for each of them for ease of programming.

The first limitation is that it is complicated to write a program using OpenCL. In order to lower this programming complexity, this thesis proposes a framework that translates a program written in a high-level language (OpenMP) to OpenCL at the source level. This thesis achieves both ease of programming and high performance by employing two techniques; data transfer minimization (DTM) and performance portability enhancement (PPE). This thesis shows the effectiveness of the proposed translation framework by evaluating benchmark applications and the practicality by comparing it with the commercial PGI compiler.

The second limitation of OpenCL is the lack of support for a heterogeneous cluster. In order to extend OpenCL to a heterogeneous cluster, this thesis proposes a framework called SnuCL-D that is able to execute a program written only in OpenCL on a heterogeneous cluster. Unlike previous approaches that apply a centralized approach, the proposed framework applies a decentralized approach, which gives a chance to reduce three kinds of overhead occurring in the execution path of commands. With the ability to analyze and reduce three kinds of overhead, the proposed framework shows good scalability for a large-scale cluster system. The proposed framework proves its effectiveness and practicality by compared to the representative centralized approach (SnuCL) and MPI with benchmark applications.

This thesis proposes solutions for the two limitations of OpenCL for ease of programming on heterogeneous clusters. It is expected that application developers will be able to easily execute not only an OpenMP program on various accelerators but also a program written only in OpenCL on a heterogeneous cluster.

Keywords : OpenMP, OpenCL, ease of programming, high performance, clusters, heterogeneous systems, programming model, accelerators, benchmarks

Student ID : 2009-30183

Contents

Abstract . . . . **i**

I. Introduction **1**

I.1 Motivation and Objectives 5

I.1.1 Programming Complexity 5

I.1.2 Lack of Support for a Heterogeneous Cluster . 8

I.2 Contributions 12

II. Background and Related Work **15**

II.1 Background 15

II.1.1 OpenCL 16

II.1.2 OpenMP 23

II.2 Related Work 26

II.2.1 Programming Complexity 26

II.2.2 Support for a Heterogeneous Cluster 29

III. Lowering the Programming Complexity **34**

III.1 Motivating Example 35

III.1.1 Device Constructs 35
III.1.2 Needs for Data Transfer Optimization 41
III.2 Mapping OpenMP to OpenCL 44
III.2.1 Architecture Model 44
III.2.2 Execution Model 45
III.3 Code Translation 46
III.3.1 Translation Process 46
III.3.2 Translating OpenMP to OpenCL 48
III.3.3 Example of Code Translation 50
III.3.4 Data Transfer Minimization (DTM) 62
III.3.5 Performance Portability Enhancement (PPE) . 66
III.4 Performance Evaluation 69
III.4.1 Evaluation Methodology 70
III.4.2 Effectiveness of Optimization Techniques . . . 74
III.4.3 Comparison with Other Implementations . . . 79

IV. Support for a Heterogeneous Cluster 90
IV.1 Problems of Previous Approaches 90
IV.2 The Approach of SnuCL-D 91
IV.2.1 Overhead Analysis 93
IV.2.2 Remote Device Virtualization 94
IV.2.3 Redundant Computation and Data Replication 95
IV.2.4 Memory-read Commands 97
IV.3 Consistency Management 98
IV.4 Deterministic Command Scheduling 100

IV.5 New API Function: clAttachBufferToDevice() 103
IV.6 Queueing Optimization 104
IV.7 Performance Evaluation 105
IV.7.1 Evaluation Methodology 105
IV.7.2 Evaluation with a Microbenchmark 109
IV.7.3 Evaluation on the Large-scale CPU Cluster . . 111
IV.7.4 Evaluation on the Medium-scale GPU Cluster . 123

V. Conclusion and Future Work 125

Bibliography 129

Korean Abstract 140

List of Figures

I.1. The overview of the translation framework. 7

I.2. Previous OpenCL frameworks for a cluster. 9

I.3. An overview of SnuCL-D. 10

II.1. The architecture model of OpenCL. 16

II.2. The execution model of OpenCL. 18

II.3. An example of an NDRange index space in OpenCL. 18

II.4. An example code computing `C = A + B`. 20

II.5. The architecture model of OpenMP. 23

II.6. The execution model of OpenMP. 24

II.7. An OpenMP example code computing `C = A + B`. . 25

III.1. An example code computing `c = Ax + b`. 36

III.2. Usages of `distribute` and `distribute parallel for`. 40

III.3. An example code in which the host modifies data within a device data environment. 43

III.4. The translation process. 46

III.5. An OpenMP program using the device constructs. . 47

III.6. The OpenCL program translated from the program in Figure III.2 (b). 50
III.7. Scheduling policies for CPUs and GPUs. 56
III.8. Reduction policies for CPUs and GPUs. 68
III.9. Reduction in the number of data transfers with DTM on the AMD GPU. 74
III.10. Effectiveness of optimization techniques. 75
III.11. Performance comparison to other implementations. . 80
IV.1. The organization of the SnuCL-D runtime. 92
IV.2. Remote device virtualization. 95
IV.3. Commands C1 and C2 modify the same memory object. 101
IV.4. Comparison of SnuCL and SnuCL-D using a microbenchmark. 111
IV.5. Comparison on the large-scale CPU cluster. 112
IV.6. Distribution of the kernel execution time of `BT`. . . . 120
IV.7. Comparison on the medium-scale heterogeneous cluster. 122

List of Tables

III.1. Description of four map-types. 38

III.2. Prototypes of runtime functions. 52

III.3. Formula that calculates the default number of teams. 60

III.4. Actions according to events to minimize data transfers. 61

III.5. Applications used. 71

III.6. Evaluation environment. 73

IV.1. Actions of a special command for consistency management in SnuCL-D. 99

IV.2. Conditions under which commands do not need to be enqueued. 105

IV.3. System configuration for the large-scale CPU cluster. 106

IV.4. System configuration for the medium-scale GPU cluster. 106

IV.5. Applications used. 107

IV.6. The number of commands executed on the large-scale CPU cluster. 114

IV.7. The number of commands for actual devices after queueing optimization. 118

IV.8. Speedup of SnuCL-D over SnuCL on the large-scale CPU cluster. 121

Chapter I

Introduction

A heterogeneous system is a system that contains different types of processors, such as general-purpose CPUs, GPUs, FPGAs, DSPs, and other types of accelerators. According to the TOP500 list[11], the number of heterogeneous supercomputers is continuously growing. As of November 2015, 104 supercomputers in the TOP500 list are equipped with accelerators. Based on this trend, it is easily expected that the number of heterogeneous high performance computing (HPC) systems will continue to grow. In addition to this, mobile devices are already heterogeneous systems. For example, smartphones, in addition to having a general purpose CPU, have a dedicated graphics processor to efficiently process images in real time. From mobile devices to supercomputers, heterogeneous systems are already everywhere in the world.

One of the most popular accelerators is the graphics processing unit (GPU), and several programming models have been proposed to use it efficiently. Among others, CUDA[53] and OpenCL[32] are used widely.

CUDA (Compute Unified Device Architecture) is proposed and maintained by NVIDIA. Since CUDA is designed for NVIDIA GPUs only, it exposes many hardware-specific details to programmers to achieve maximum performance on their GPUs.

OpenCL (Open Computing Language) is an open and royalty-free heterogeneous programming model for many different types of accelerators including general-purpose CPUs. Its specification is maintained by the Khronos group, and many hardware vendors, such as Altera, AMD, Apple, ARM, IBM, Imagination, Intel, MediaTek, NVIDIA, Qualcomm, Samsung, Xilinx, *etc.*, provide OpenCL implementations for their hardware. Since an OpenCL program should be able to execute on different types of processors, OpenCL only provides hardware-independent functions that are able to be performed by all processors that support OpenCL. In other words, an OpenCL program is portable across various types of hardware platforms.

In this thesis, two limitations of OpenCL are presented, and a solution for each of them is proposed. The first one is programming complexity of OpenCL. Since application developers should write coordination code as well as kernel code to use OpenCL, they should un-

derstand language extensions and various API functions in OpenCL. This increases maintenance cost and decreases productivity. In an effort to alleviate programming complexity from application developers, this thesis proposes a framework that allows application developers to write a program using OpenMP as a high-level, directive-based programming model and translates programs written in OpenMP to OpenCL.

OpenMP was originally proposed to provide ease of programming for multicore processors, and it has been extended to support accelerators since OpenMP 4.0. Using OpenMP, application developers can simply insert compiler directives to existing programs. The code annotated with the compiler directives is executed on an accelerator. By converting an OpenMP program to an OpenCL program, the OpenMP program can be executed on various hardware platforms that support OpenCL. This provides ease of programming on a heterogeneous system to application developers.

To lower programming complexity even more, a technique that automatically minimizes data transfers is proposed. Using this technique, data transfers between the host and an accelerator are automatically minimized by the runtime system without application developers' efforts to reduce data transfers. In addition, in order to enhance performance portability, the proposed framework applies different scheduling and reduction policies for different hardware platforms. This thesis also verifies the practicality and effectiveness of

the translation framework by evaluating it with various benchmark applications.

The second limitation of OpenCL is the lack of support for a heterogeneous cluster. OpenCL only focuses on being executed on a single operating system (OS) instance even though it is able to be extended to support a heterogeneous cluster transparently. To program a heterogeneous cluster, both OpenCL and a communication library such as MPI are required. This is difficult and error-prone as application developers use two programming models in a single program. Instead of using a mix of two programming models, this thesis proposes a framework, *SnuCL-D*, in order to execute a program written only in OpenCL on a heterogeneous cluster.

SnuCL-D gives application developers the illusion as if all OpenCL compute devices scattered in different nodes were in a single node. There are a lot of previous studies that extend OpenCL for heterogeneous clusters. Unlike previous studies that apply a centralized approach, the proposed framework applies a decentralized approach. Using a decentralized approach, this framework can obtain better scalability compared to the previous studies. Also, this thesis analyzes three kinds of overhead that occurs in the execution path of commands, and proposes techniques to reduce them.

The first source of overhead is *delivering overhead* which occurs because the host needs to send messages and data to other compute

nodes. This can be reduced by the decentralized approach. The second kind of overhead is *scheduling overhead* which occurs because the OpenCL runtime schedules commands and maintains memory consistency between devices. The overhead of maintaining memory consistency can be reduced by using the proposed new OpenCL API function. The third source of overhead is *enqueueing overhead* which occurs because unnecessary commands are enqueued to maintain memory consistency. These unnecessary commands occur because memory consistency is no longer required due to the proposed OpenCL API function. These can be reduced by not enqueueing unnecessary commands. While previous studies do not evaluate their scalability for a large-scale cluster, the solution of this thesis shows good scalability for a large-scale cluster.

I.1 Motivation and Objectives

I.1.1 Programming Complexity

Programming heterogeneous systems is usually difficult and complicated with current major programming models, such as CUDA[52] and OpenCL[33]. The code is much longer when a sequential program is converted to OpenCL even for a simple vector addition program as shown in Section II.1.1. OpenCL requires programmers to understand their language extensions and various API functions. For example, a

fully parallel loop in an application can be executed on a GPU to boost performance. To use OpenCL, application developers have to write coordination code as well as the kernel code that is equivalent to the parallel loop. Specifically, they have to allocate memory objects on the accelerator and explicitly copy the data from the host to the accelerator as needed. When the data is ready, the kernel is invoked. Then, the result is copied back from the accelerator to the host. Even though the coordination code does not relate to the core computation in the program, they should know how to write the coordination code in CUDA or OpenCL. This degrades productivity and increases maintenance costs.

To lower programming complexity, OpenACC[6] and OpenMP 4.0[7] have been introduced. OpenACC is an application program interface that helps application developers create an application for heterogeneous systems. By simply adding OpenACC compiler directives to an existing sequential program, the program can be executed on an accelerator. The representative compiler that supports OpenACC is the PGI compiler[5].

OpenMP was originally proposed for multicore processors with a shared memory model. Its support has expanded to programming heterogeneous systems since OpenMP 4.0. The latest version, OpenMP 4.5[9], was released in November 2015.

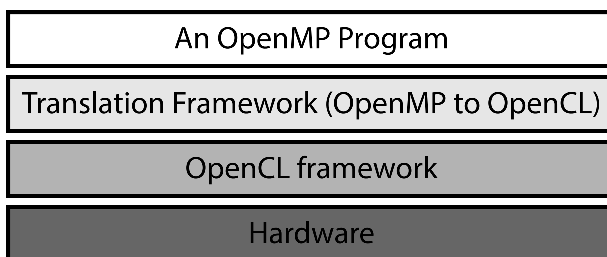


Figure I.1: The overview of the translation framework.

This thesis proposes techniques to translate OpenMP 4.0 accelerator directives to OpenCL. Other directives are not translated and remain the same. When the translated code is compiled by using other compilers that support OpenMP, such as gcc, the remaining OpenMP directives are also translated. In addition, this thesis currently targets programs written in C or C++. The same techniques can also be used to translate OpenMP 4.0 accelerator directives in Fortan.

Figure I.1 shows the overview of the translation framework. An OpenMP program is converted to an OpenCL program. Then, the translated program is executed on the OpenCL framework. If OpenMP accelerator directives are translated to OpenCL, the translated program can be executed on various hardware platforms that support OpenCL. This allows for greater portability, which is one of OpenCL's strengths. Currently, an OpenCL program can be executed on multi-core CPUs, GPUs, Intel Xeon Phi coprocessors, DSPs, and FPGAs. Using the proposed translation framework, application developers can simply insert OpenMP directives to an existing program in order to execute the program on various hardware platforms.

I.1.2 Lack of Support for a Heterogeneous Cluster

OpenCL and CUDA work only on a single operating system (OS) instance. Even if heterogeneous computing on a single OS instance continues to grow in popularity, more research is needed for heterogeneous computing with a cluster running multiple OS instances, one in each node. Since both CUDA and OpenCL were originally developed for a single OS instance, applications written solely in CUDA or OpenCL cannot be executed on the entire cluster. To develop OpenCL or CUDA applications for the entire cluster, a communication library, such as MPI[49], must be used to support communication between different nodes (*i.e.*, between different OS instances). As a result, programmers are forced to use a mix of two different programming models. However, it is cumbersome and error-prone for programmers to switch between two different programming models in different phases of an application.

Moreover, programmers have to distribute workload hierarchically in two levels: MPI and OpenCL. If a loop is parallelized by MPI and OpenCL, the iterations of the loop are distributed by MPI first. Then, the iterations assigned to a node are distributed to compute devices in the node using OpenCL.

Numerous studies have addressed this issue and proposed solutions[13, 16, 25, 40, 42, 45, 70, 73]. These approaches, albeit with different implementations, have similar goals and ideas. Figure I.2 il-

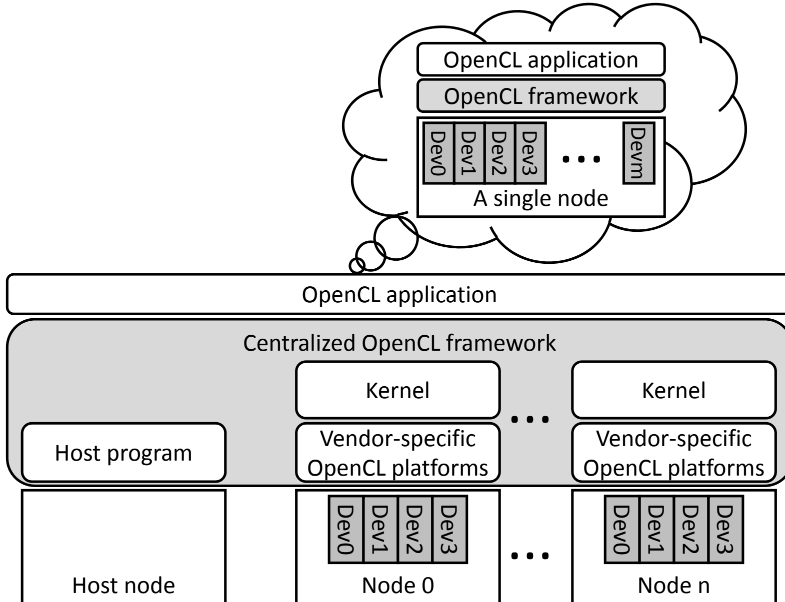


Figure I.2: Previous OpenCL frameworks for a cluster.

lustrates the idea. There is a centralized host node that executes the host program, and the other nodes perform kernel computation coordinated by the centralized host node. When the OpenCL host program enqueues commands to command-queues, the host node schedules the commands according to the execution order defined by the host program and delivers them to the proper target nodes. It is a natural and intuitive implementation because the OpenCL platform model contains a single host and multiple compute devices.

However, most of the previous approaches do not address the scalability issue in their solutions. The centralized mechanism may become a significant performance bottleneck for a large-scale cluster due to the centralized host. Most of them evaluate their OpenCL frame-

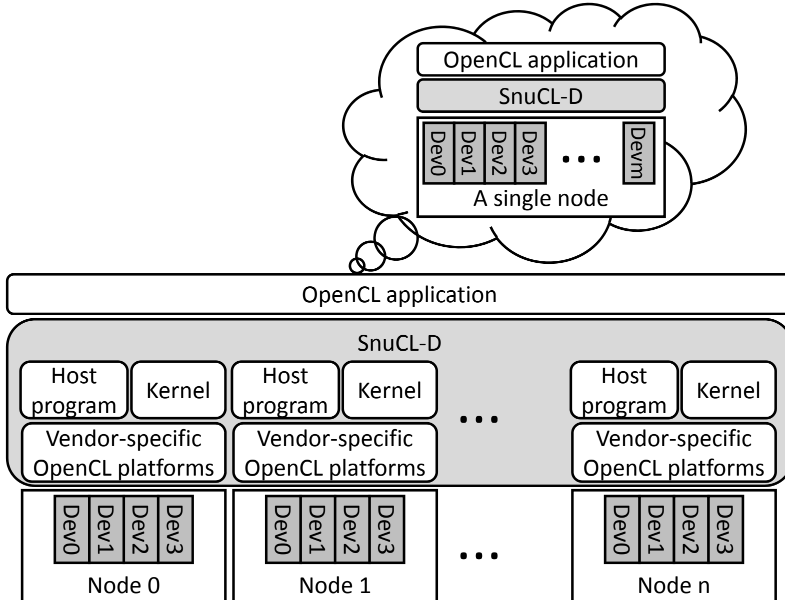


Figure I.3: An overview of SnuCL-D.

works only with a small-scale cluster. One exception is our previous SnuCL[42] that uses a large-scale cluster with 256 nodes for the evaluation. We reported that SnuCL did not scale well with more than 64 nodes on some applications due to its centralized approach. The main problem is the centralized host node. In this centralized approach, the host node delivers commands to compute nodes, then the compute nodes execute the commands. The host node may become a bottleneck if there are a large number of compute nodes.

To solve the scalability problem of the centralized approach, this thesis proposes a scalable and decentralized OpenCL framework for a large-scale heterogeneous cluster. It exploits redundant computation and data replication. This new OpenCL framework is called *SnuCL-*

D. In general, the OpenCL host program controls which command is executed on which compute device. In the centralized approach, the host program is executed only on the host node and delivers control messages and data to the compute nodes. Instead of doing this, SnuCL-D executes the host program in every node redundantly (*i.e., redundant computation*) as shown in Figure I.3. Because of the redundant computation, every node has the data it needs (*i.e., data replication*). Since the data is replicated in all nodes, data transfer incurred by the host node in the centralized approach is eliminated. Although it increases the overall memory footprint in the system, it also significantly improves performance. Note that this approach only increases the memory footprint in the host memory of each node; the memory space required in each compute device remains the same.

There are studies[12, 37, 43, 61, 72] that exploit redundant computation and data replication to improve performance on parallel systems even though they require more computing power and memory footprint.

As a result, SnuCL-D provides the OpenCL application with the view of a single OpenCL platform image. The OpenCL application uses any compute device in the cluster as if it were in the single OpenCL platform.

I.2 Contributions

This thesis presents two limitations of OpenCL and tackles them in two ways. First, to lower the programming complexity of OpenCL, a translation framework converting an OpenMP program to OpenCL is proposed. It helps application developers exploit the accelerator much more easily. Second, to support a heterogeneous cluster, this thesis proposes a decentralized and scalable OpenCL framework that transparently extends OpenCL to a large-scale heterogeneous cluster. This also helps application developers exploit a heterogeneous cluster easily while preserving scalability.

The contributions of lowering the programming complexity are the following:

- We present the techniques used in the translation framework that converts OpenMP 4.0 device constructs to OpenCL. It is the first practical implementation of the translation framework of OpenMP 4.0 device constructs.
- We propose an optimization technique that automatically minimizes data transfers between the host and the accelerators. This technique frees programmers from finding the optimal locations for data transfer code/directives in OpenMP 4.0 programs. This further lowers the programming complexity of OpenMP 4.0.
- We propose different scheduling and reduction policies in the

target OpenCL code for different hardware architectures. These policies enhances performance portability of the target code.

- To show the effectiveness of our translation framework, we evaluate our translation framework with the original OpenCL and OpenMP programs on three different devices: an multicore Intel CPU, an AMD GPU, and an NVIDIA GPU.
- To show the practicality of our translation framework, we also compare it with the commercial PGI compiler.

The contributions to support a large-scale heterogeneous cluster are the following:

- This thesis proposes a scalable decentralized mechanism of OpenCL frameworks for a cluster. This approach executes the host program in every node to eliminate performance bottlenecks caused by the centralized host. It especially exploits redundant computation and data replication. It also proposes remote device virtualization and deterministic command scheduling techniques.
- This thesis proposes a technique to avoid enqueueing unnecessary commands to command-queues of virtual devices. This technique is implemented in all OpenCL API functions that insert commands to command-queues. It alleviates the command queueing and scheduling overhead.
- This thesis proposes a new OpenCL API function **clAttachBufferToDevice()** to alleviate consistency management over-

head. Coupling it with the queueing optimization technique, SnuCL-D alleviates the consistency management and remote command scheduling overhead significantly.

- The effectiveness of SnuCL-D can be seen by the evaluation of eleven benchmark applications running on a large-scale CPU cluster with 512 nodes (4096 CPU cores in total) and a medium-scale GPU cluster with 36 nodes (144 GPU devices in total). Also, SnuCL-D is compared to MPI and SnuCL. It is shown that SnuCL-D is always faster than SnuCL, and in some occasions going up to 45 times faster than SnuCL. SnuCL-D is implemented by modifying SnuCL[42], our previous open-source OpenCL framework for heterogeneous clusters.

Chapter II

Background and Related Work

II.1 Background

This thesis mainly discusses OpenCL and OpenMP, and this section introduces the architecture model and the execution model of OpenCL and OpenMP.

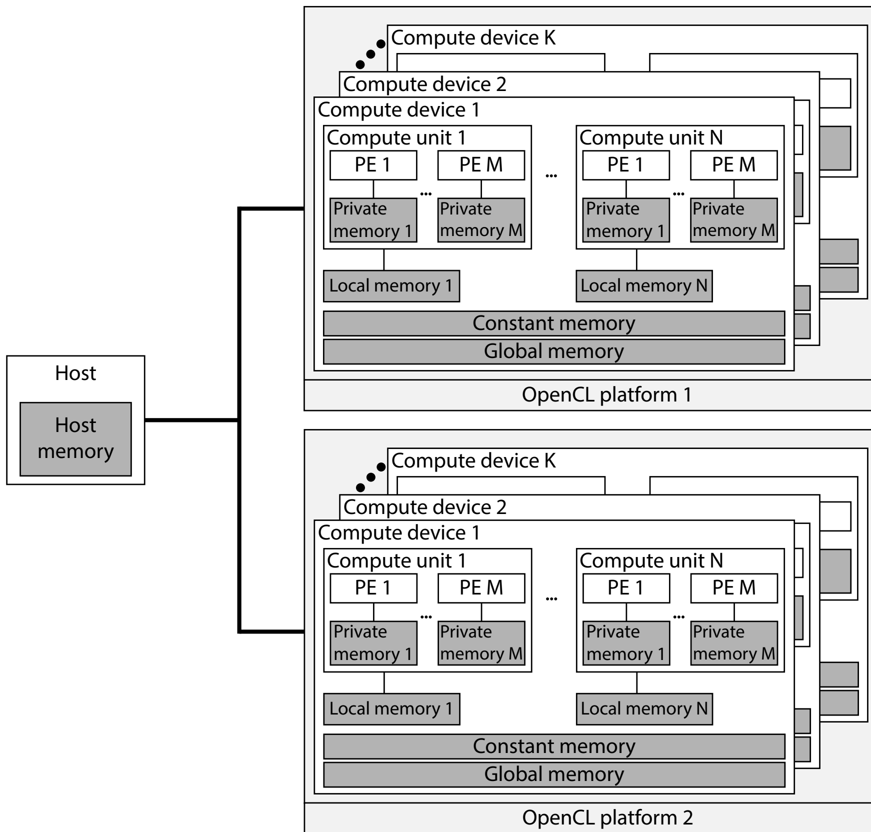


Figure II.1: The architecture model of OpenCL.

II.1.1 OpenCL

II.1.1.1 Architecture Model

Figure II.1 shows the architecture model of OpenCL. There is a single *host* and multiple *compute devices*. The host owns the distinct host memory that is the main memory of a typical heterogeneous system. A host program, which is written in the general C/C++ language,

is executed on the host, and an OpenCL kernel, which is written in the OpenCL C language, is executed on compute devices. Since multiple accelerators that support OpenCL can co-exist in a system, OpenCL provides a concept of OpenCL platforms. For example, a system can contain two AMD GPUs and two NVIDIA GPUs. Since AMD and NVIDIA provide their OpenCL implementations, there are two OpenCL implementations (platforms) in the system; one from AMD and one from NVIDIA. Programmers can select an OpenCL platform in a program.

A compute device has multiple *compute units* (CUs). A compute unit has multiple *processing elements* (PEs). A PE is a unit of execution of an OpenCL kernel. An OpenCL kernel is a description of what a PE should do. A compute device has four distinct memories: *private* memory, *local* memory, *constant* memory and *global* memory. Private memory is private to a single PE and cannot be accessed by other PEs. The local memory is shared by PEs in a CU. A CU's local memory cannot be accessed by PEs in other CUs. Constant memory is shared by all PEs in the compute device. Since this memory is read-only memory, it cannot be written by PEs. Global memory is shared by all PEs in the compute device. Unlike the constant memory, the global memory can be read or written by any PEs.

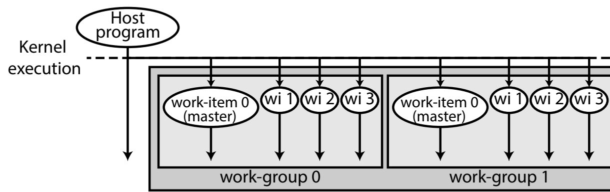


Figure II.2: The execution model of OpenCL.

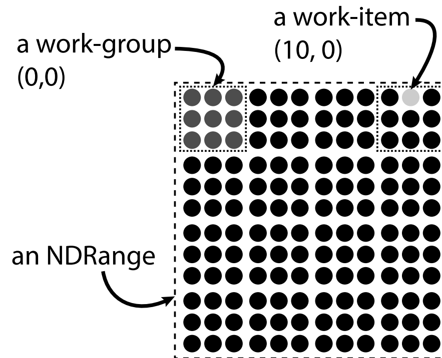


Figure II.3: An example of an NDRange index space in OpenCL.

II.1.1.2 Execution Model

Figure II.2 shows the execution model of OpenCL. At first, the host program is executed on the host. When a kernel is invoked, it is executed on the compute device. Before invoking a kernel, an index space (NDRange) of the kernel should be specified. Figure II.3 shows an example of an NDRange of an OpenCL kernel. This is specified by the programmer. The figure presents a two-dimensional index space, and each point is called a work-item. A work-item is an instance of a kernel, and it is executed on a PE. A work-item has a unique global and local ID in the index space. The global ID of a work-item is its posi-

tion in the NDRange. The local ID of a work-item is its position in its work-group. For example, the work-item whose global ID is (10, 0) is presented as the light gray circle in the figure. Its local ID is (1, 0). A work-group is a group of work-items, and it is assigned to a CU. Also, a work-group has a unique ID in the index space. Basically, the OpenCL runtime dispatches a work-group to a CU, and work-items in the work-group are executed concurrently on PEs in the CU.

II.1.1.3 Description of an OpenCL Program

Figure II.4 shows two programs that add two vectors ($\mathsf{C} = \mathsf{A} + \mathsf{B}$). Figure II.4 (a) shows a sequential C program while Figure II.4 (b) shows an OpenCL program. Even though the algorithm of the program is very simple, the OpenCL program requires many OpenCL API functions. This increases maintenance cost and decreases productivity.

In Figure II.4 (b), an OpenCL platform should be obtained first at line 20. An OpenCL device is obtained from the OpenCL platform (line 21). Then, an OpenCL context is created (line 23). The OpenCL context is a container comprising OpenCL objects such as devices, command-queues, programs, kernels, and memory objects. In order to invoke a kernel on the device, a command-queue for the device is created (line 25). In OpenCL, the host program controls compute devices with commands. The commands are enqueued to the command-

```
 1 #include <stdio.h>
 2 
 3 int main() {
 4   const int N = 1024;
 5   size_t bytes = N * sizeof(int);
 6 
 7   int* host_A = (int*)malloc(bytes);
 8   int* host_B = (int*)malloc(bytes);
 9   int* host_C = (int*)malloc(bytes);
10 
11   // Initialize host memory objects.
12   for(int i = 0; i < N; ++i) {
13     host_A[i] = i;
14     host_B[i] = i+3;
15   }
16 
17   // Calculate C = A + B.
18   for(int i = 0; i < N; ++i) {
19     host_C[i] = host_A[i] + host_B[i];
20   }
21 
22   return 0;
23 }
```

(a) A typical C program.

```
 1 #include <CL/opencl.h>
 2 #include <stdio.h>
 3 
 4 // An OpenCL kernel that calculates C = A + B.
 5 const char* source_str = "__kernel void vecAdd("
 6 "__global int* A, __global int* B, __global int * C,"
 7 " const int N) {"
 8 "    int id = get_global_id(0); "
 9 "    C[id] = A[id] + B[id];"
10 " }";
11 
12 int main() {
13   cl_platform_id   platform;
14   cl_device_id     device;
15   cl_context       context;
16   cl_command_queue cmd_q;
17   cl_int           err;
18 
19   // Get an OpenCL platform and device.
20   err = clGetPlatformIDs(1, &platform, NULL);
21   err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device,
        NULL);
22   // Create an OpenCL context.
23   context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
24   // Create a command-queue.
25   cmd_q = clCreateCommandQueue(context, device, 0, &err);
26 
27   const int N = 1024;
28   size_t bytes = N * sizeof(int);
```

Figure II.4: An example code computing `C = A + B`.

```
29   // Create OpenCL memory objects.
30   cl_mem mem_A = clCreateBuffer(context, CL_MEM_READ_WRITE,
         bytes, NULL, &err);
31   cl_mem mem_B = clCreateBuffer(context, CL_MEM_READ_WRITE,
         bytes, NULL, &err);
32   cl_mem mem_C = clCreateBuffer(context, CL_MEM_READ_WRITE,
         bytes, NULL, &err);
33
34   int* host_A = (int*)malloc(bytes);
35   int* host_B = (int*)malloc(bytes);
36   int* host_C = (int*)malloc(bytes);
37
38   // Initialize host memory objects.
39   for(int i = 0; i < N; ++i) {
40     host_A[i] = i;
41     host_B[i] = i+3;
42   }
43
44   // Copy host memory objects to OpenCL memory objects.
45   err = clEnqueueWriteBuffer(cmd_q, mem_A, CL_TRUE, 0, bytes,
         host_A, 0, NULL, NULL);
46   err = clEnqueueWriteBuffer(cmd_q, mem_B, CL_TRUE, 0, bytes,
         host_B, 0, NULL, NULL);
47
48   // Create an OpenCL program from source.
49   cl_program pg = clCreateProgramWithSource(context, 1, (const
         char**)&source_str, NULL, &err);
50
51   // Build the OpenCL program.
52   err = clBuildProgram(pg, 1, &device, NULL, NULL, NULL);
53
54   // Create an OpenCL kernel.
55   cl_kernel kernel = clCreateKernel(pg, "vecAdd", &err);
56
57   // Set arguments of the OpenCL kernel.
58   err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &mem_A);
59   err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &mem_B);
60   err = clSetKernelArg(kernel, 2, sizeof(cl_mem), &mem_C);
61   err = clSetKernelArg(kernel, 3, sizeof(int), &N);
62
63   size_t lws[1] = {32};
64   size_t gws[1] = {N};
65   // Launch the OpenCL kernel.
66   err = clEnqueueNDRangeKernel(cmd_q, kernel, 1, NULL, gws, lws,
          0, NULL, NULL);
67
68   // Get the result from the device.
69   err = clEnqueueReadBuffer(cmd_q, mem_C, CL_TRUE, 0, bytes,
         host_C, 0, NULL, NULL);
70
71   return 0;
72 }
```

(b) An OpenCL program.

Figure II.4: An example code computing `C = A + B` (continued).

queue. The command-queue is attached to a single compute device, and commands enqueued in the command-queue are executed on the attached compute device.

There are three types of commands in OpenCL: *kernel execution* commands, *memory* commands, and *synchronization* commands. Kernel execution commands are used to execute a kernel on a compute device. Memory commands are used to transfer data between the host and compute devices or between compute devices. Synchronization commands are used to enforce order between commands.

According to the architecture model of OpenCL, the host and a compute device do not share memory in general. Hence, OpenCL memory objects should be created to allocate memory regions on a compute device (line 30-32). The initialized data is copied to OpenCL memory objects using `clEnqueueWriteBuffer()` (line 45-46). The OpenCL kernel `vecAdd` is written in the OpenCL C language and included as a C string (line 5-10). This string becomes an OpenCL program (line 49). An OpenCL program is a container consisting of multiple OpenCL kernels. In this example the OpenCL program has only one kernel `vecAdd`. The source string is built to a binary using `clBuildProgram()` (line 52).

After building the OpenCL program, a kernel object is created (line 55). Using the kernel object, arguments of the kernel are set using `clSetKernelArg()` (line 58-61). Then, the kernel is invoked

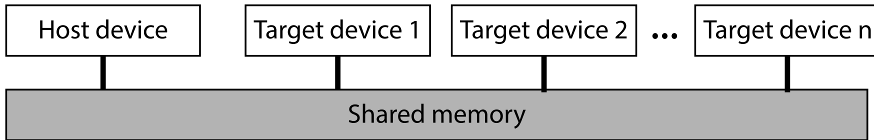


Figure II.5: The architecture model of OpenMP.

(line 66). When the kernel is invoked, programmers should specify the size of a work-group (`lws` at line 63) and the size of the NDRange (`gws` at line 64). Finally, the result is copied back to the host using `clEnqueueReadBuffer()` (line 69).

II.1.2 OpenMP

OpenMP was originally proposed to provide ease of programming for multicore CPUs. OpenMP has been extended to support accelerators since OpenMP 4.0. In OpenMP 4.0, device constructs such as the `target` construct are proposed for accelerators.

II.1.2.1 Architecture Model

Figure II.5 shows the architecture model of OpenMP. In contrast to OpenCL, it is much simpler. OpenMP does not expose any architectural detail of accelerators. There is a single host device and multiple target devices. An OpenMP program is executed on the host device at the beginning. When a `target` construct is encountered, the code in the construct is executed on a target device. Since OpenMP provides

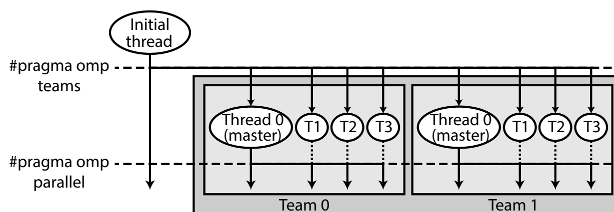


Figure II.6: The execution model of OpenMP.

a shared-memory model, there is a single memory shared by the host device and target devices.

II.1.2.2 Execution Model

Figure II.6 shows the execution model of OpenMP. At first, the host program is executed in the initial thread on the host device. When a `target` construct is encountered, the code is executed on a target device. All threads in all teams are created when a `teams` construct is encountered. In the `teams` construct, the number of teams and the maximum number of threads are defined. Only the master thread (*i.e.*, the first thread) in each team executes code before a `parallel` construct is encountered. When a `parallel` construct is encountered, all threads begins execution.

II.1.2.3 Description of an OpenMP Program

Figure II.7 shows an OpenMP example program computing `C = A + B`. The code is almost the same as in Figure II.4 (a). The only

```
 1 #include <stdio.h>
 2
 3 int main() {
 4   const int N = 1024;
 5   size_t bytes = N * sizeof(int);
 6
 7   int* host_A = (int*)malloc(bytes);
 8   int* host_B = (int*)malloc(bytes);
 9   int* host_C = (int*)malloc(bytes);
10
11   // Initialize host memory objects.
12   for(int i = 0; i < N; ++i) {
13     host_A[i] = i;
14     host_B[i] = i+3;
15   }
16
17   // Below code is executed on the target device.
18 #pragma omp target teams distribute parallel for\
19   map(to:host_A[0:N], host_B[0:N]) \
20   map(from:host_C[0:N])
21   for(int i = 0; i < N; ++i) {
22     host_C[i] = host_A[i] + host_B[i];
23   }
24
25   return 0;
26 }
```

Figure II.7: An OpenMP example code computing `C = A + B`.

difference is the compiler directives at line 18-20. The `for` loop at line 21 is executed on a target device by simply inserting OpenMP compiler directives immediately before the loop.

Even though OpenMP assumes a shared memory model, memory regions accessed by pointers should be declared using `map` clauses with array sections (*e.g.*, `map(from:host_C[0:N])` at line 20). Since the pointers `host_A` and `host_B` are only read in the loop, they are declared using the `map(to:)` clause at line 19. Using `map(to:)`, the memory regions pointed to are copied from the host to the target device before executing the loop on the target device. Since the pointer `host_C` is

only written in the loop, it is declared using the `map(from:)` clause at line 20. Using `map(from:)`, the memory region pointed to is copied from the target device to the host after execution of the loop has finished.

II.2 Related Work

II.2.1 Programming Complexity

There have been many studies that lower the programming complexity for heterogeneous systems. For example, SYCL[10] is a C++ programming model for OpenCL that exploits portability and efficiency of OpenCL and lowers programming complexity of OpenCL. Application developers write a general C++ program using the SYCL libraries, and the program exploits the full range of capabilities of OpenCL internally. However, there is a disadvantage to this approach in that application developers have to learn how to use SYCL.

hiCUDA[35] is a high-level directive-based language for CUDA. It is very similar to OpenACC, but it exposes large amount of details of CUDA. Hence, programmers need to know the underlying details of CUDA to use hiCUDA even though what they have to do is to simply insert hiCUDA compiler directives. In addition, it is not easy to convert hiCUDA programs to programming languages other than CUDA.

As discussed in Section I.1.1, OpenACC[6] and OpenMP 4.0[7] were proposed to lower the programming complexity for heterogeneous systems. The PGI compiler[5] supports OpenACC directives for heterogeneous computing. Currently, there is no compiler that supports OpenMP 4.0 device constructs correctly.

There have been studies that try to use OpenMP for heterogeneous systems. OpenMP/Clang[4] tries to implement the OpenMP 4.0 specification using clang[46]. However, it focuses on OpenMP 4.0 features that are not the device constructs. While it supports parsing device constructs, it does not have functionality to offload computation to accelerators. Their work is merged to clang/llvm. This thesis targets to execute the code fragment annotated with OpenMP device constructs on an accelerator.

Ayguade *et al.*[19, 20] propose an extension of the OpenMP 3.0 tasking model to integrate heterogeneity while preserving simplicity and portability. Programmers denote tasks and their dependences. The compiler and runtime execute the code in parallel by analyzing the dependences between tasks. They also present challenges of implementing the extension for various architectures and show that they obtain reasonable performance compared to the hand-tuned version. While they target the task-based programming model, we target the data parallel programming model.

Beyer *et al.*[22] also propose an extension of the OpenMP programming model. Unlike the Ayguade *et al.*'s approach of the parallelizing tasks, they parallelize loops. They propose several new constructs based on the PGI accelerator directives. Since they implement their approach before OpenMP 4.0 was released, their device constructs are different from OpenMP 4.0 device constructs. Moreover, they report performance of only two applications. Thus, it is not enough to see the practicality of their approach. On the other hand, our framework translates OpenMP 4.0 device constructs, and we show its effectiveness by evaluating 17 benchmark applications on various hardware platforms. Also, its practicality is demonstrated by comparing with the commercial PGI compiler.

Liao *et al.*[47] propose a compiler that translates a program using the OpenMP accelerator model in OpenMP Release Candidate 2 to CUDA by modifying the ROSE compiler[57]. They translate an OpenMP program to CUDA while we translate an OpenMP program to OpenCL. While their translator works only for NVIDIA GPUs, our translator work with various accelerators that support OpenCL. Since they only targeted NVIDIA GPUs, they did not consider the performance portability of their approach. On the other hand, we target various hardware platforms, so we make an effort in that the translated program performs well for all hardware platforms. Also, we propose the data transfer minimization technique to lower programming complexity even more. This is not considered in their paper.

Scogland *et al.*[64] propose co-scheduling mechanisms that exploit CPUs and GPUs together by extending OpenMP directives. They propose four scheduling policies to handle a variety of application behaviors. They distribute loop iterations across a CPU and a GPU. While they target the co-scheduling mechanism using OpenMP, we target the translation from OpenMP to OpenCL.

All aforementioned approaches are implemented before OpenMP 4.0 was released. Hence, none of them supports OpenMP 4.0 correctly. This paper translates device constructs from the OpenMP 4.0 standard and evaluates the translated programs to show the effectiveness and practicality of the approach.

II.2.2 Support for a Heterogeneous Cluster

There have been many studies to enable OpenCL applications to run on heterogeneous clusters[13, 16, 25, 40, 42, 45, 70, 73]. SnuCL[42] is our previous open-source OpenCL framework for a heterogeneous cluster. It divides cluster nodes into two categories: a single host node and compute nodes. The host node executes the host program of an OpenCL application while the compute nodes execute OpenCL commands by coordination of the host node. SnuCL proposes collective communication extensions, which are similar to MPI collective communication operations, to OpenCL to boost performance.

dOpenCL[40] is another OpenCL framework for distributed cluster systems. It divides nodes into two classes: the client and the servers. The client executes the host program of an OpenCL application while servers provide access to their devices over the network. To execute multiple OpenCL applications concurrently, they proposed a device manager. The device manager manages devices in the cluster, and provides devices to the OpenCL application when necessary.

*cl*OpenCL[13] makes OpenCL applications run on a heterogeneous cluster. It provides wrapper functions of OpenCL API functions. The wrapper functions call the hardware vendor's OpenCL platform. It uses Open-MX as a communication library.

Hybrid OpenCL[16] also targets heterogeneous cluster systems. It is based on the FOXC OpenCL runtime. They add a network layer using sockets to the OpenCL runtime to support communication between nodes.

There are several open-source projects (SocketCL[25], CLara[45], DistributedCL[70], and CLuMPI[73]) that execute OpenCL applications on a cluster system.

rCUDA[29, 56, 58] proposes a framework that shares GPUs using CUDA in a cluster to save energy consumption and increase resource utilization. DS-CUDA[54] is a GPU virtualization tool to enable the ability to use GPUs located in different nodes. In addition, it supports redundant calculations to increase reliability.

All aforementioned approaches have a centralized host node that executes the host program. Other nodes in the cluster perform computation controlled by the host node. Thus, the centralized host mechanism may become a bottleneck. However, most of the previous approaches evaluate their frameworks with a small-scale cluster. Among them, only SnuCL evaluates itself using a large cluster system with 256 nodes and reports poor scalability for some applications. This thesis tries to solve the scalability problem of previous centralized approaches.

Some studies propose different interfaces to exploit accelerators in a cluster. Grasoo *et al.*[31] proposed libWater, a library to simplify the programming of heterogeneous clusters. They demonstrated that it is possible to use an SQL-like programming model as an abstraction level of OpenCL without losing control over performance.

Barak *et al.*[21] proposed a *many GPUs package* (MGP) that provides an OpenMP-like API layer that exploits the MOSIX virtual OpenCL layer to execute programs on a heterogeneous cluster. They showed that kernels are executed on remote devices in a cluster efficiently.

Augonnet *et al.*[18] proposed StarPU, a task programming library for heterogeneous architectures. They also extended StarPU with MPI as a name of StarPU-MPI[17]. It exploited the task-based paradigm for a GPU cluster. They presented how the task paradigm of StarPU is combined with MPI to exploit a GPU cluster.

Duran *et al.*[30] proposed OmpSs, another task-based programming model based on OpenMP. They proposed heterogeneous extensions to OpenMP to exploit accelerators. Bueno *et al.* extended OmpSs with MPI to exploit a GPU cluster[24]. To do this, the runtime system determined how data should be moved between different nodes to minimize the impact of communication.

Hartely *et al.*[36] proposed APC+ as a simple and highly efficient task-based programming model for high-performance applications. They achieved good load balance through a work-stealing engine while providing efficient network usage by using a dedicated storage layer.

Charm++[1] is an object-based message-passing programming model. Kunzman *et al.*[44] extended Charm++ to support a heterogeneous cluster. A program using only Charm++ can be executed on a heterogeneous cluster through the underlying runtime system that maps the execution of the program across the available host and accelerator cores.

There are other studies using distributed approaches to overcome scalability problems in other research fields. Directory-based cache coherence protocols adapted a distributed approach to improve scalability[26, 67]. There are three categories in directory-based cache coherence protocols: *full-map directories*, *limited directories*, and *chained directories*. Full-map directories have a single global memory to store

states of cache lines, and each cache accesses and modifies the states simultaneously. This scheme has a problem in that the number of state bits of a cache line increases proportional to the number of caches (*i.e.*, processors). To overcome this problem, limited directories are proposed. Limited directories maintain the limited number of states, which allows a cache block to be located in only the limited number of caches. If the number of caches that claim a single cache block exceeds the limit, some caches will invalidate their cache lines.

Since the full-map directories and limited directories maintain the single global memory to store states of cache lines, the single global memory can limit the scalability of the system. They are centralized approaches. To improve scalability, the chained directories are proposed. The chained directories distribute the states in caches. Each cache line in chained directories has a pointer, and this pointer makes a chain of caches that contain the same cache line. This is a distributed approach that the states of a cache block are distributed in each cache. The chained directories are more scalable than the full-map directories and the limited directories.

The research of file systems advanced towards distributed approaches. Traditional file systems like NFS contain a central server to provide all file system services. Such a centralized server can be a performance and reliability bottleneck[15, 38, 48, 62]. To overcome this bottleneck, many distributed file systems were proposed[63, 69, 71]. They distribute control and data to improve scalability.

Chapter III

Lowering the Programming Complexity

To lower the programming complexity of OpenCL, this thesis proposes a translation framework that converts an OpenMP program to OpenCL. Since programming using OpenMP is much easier than programming using OpenCL, it is expected that application developers will be able to easily exploit accelerators with OpenMP. By converting OpenMP programs to OpenCL, the programs can be executed on many hardware platforms that support OpenCL. Currently, examples of platforms that support OpenCL are multicore CPUs, GPUs, FPGA, DSPs, and Intel Xeon Phi coprocessors.

III.1 Motivating Example

This section introduces important OpenMP device constructs and presents a motivating example.

III.1.1 Device Constructs

Figure III.1 (a) shows a sequential program that computes `c = Ax + b`. While `A` is a matrix, `c`, `x`, and `b` are vectors. Figure III.1 (b) shows an OpenMP program that exploits an accelerator. There are two `for` loops, and each of them is annotated by an OpenMP 4.0 device construct.

The `target` construct (line 5 in Figure III.1 (b)) allows for the code in the next structured block to be executed on a target device. The target device can be explicitly given, or the environment variable `OMP_DEFAULT_DEVICE` sets the default target device to use in a `target` construct. The `target` construct at line 5 in Figure III.1 (b) uses the default device.

In addition, the `target` construct creates a new device data environment. When an OpenMP program begins, each device has an initial device data environment that provides a context for execution. Directives accepting data-mapping attribute clauses (`map`) determine how an original variable is mapped to a corresponding variable in the

```
1  // c = Ax + b
2  void MatVecMul(double* c, double* A, double* x,
3      double* b, int N) {
4    // 1. c = Ax
5    for(int i = 0; i < N; ++i) {
6      double sum = 0.0;
7      for(int j = 0; j < N; ++j) {
8        sum += A[i*N+j] * x[j];
9      }
10     c[i] = sum;
11   }
12   // 2. c = c + b
13   for(int i = 0; i < N; ++i) {
14     c[i] = c[i] + b[i];
15   }
16 }
```

(a) A sequential program.

```
1  // c = Ax + b
2  void MatVecMul(double* c, double* A, double* x,
3      double* b, int N) {
4    // 1. c = Ax
5  #pragma omp target teams distribute parallel for \
6    map(to:A[0:N*N], x[0:N]) \
7    map(from:c[0:N])
8    for(int i = 0; i < N; ++i) {
9      double sum = 0.0;
10     for(int j = 0; j < N; ++j) {
11       sum += A[i*N+j] * x[j];
12     }
13     c[i] = sum;
14   }
15   // 2. c = c + b
16 #pragma omp target teams distribute parallel for \
17   map(tofrom:c[0:N]) \
18   map(to:b[0:N])
19   for(int i = 0; i < N; ++i) {
20     c[i] = c[i] + b[i];
21   }
22 }
```

(b) An OpenMP program.

Figure III.1: An example code computing `c = Ax + b`.

```
1  // c = Ax + b
2  void MatVecMul(double* c, double* A, double* x,
3      double* b, int N) {
4  #pragma omp target data \
5    map(to:A[0:N*N], x[0:N], b[0:N]) \
6    map(from:c[0:N])
7    {
8      // 1. c = Ax
9  #pragma omp target teams distribute parallel for \
10   map(to:A[0:N*N], x[0:N]) \
11   map(from:c[0:N])
12     for(int i = 0; i < N; ++i) {
13       double sum = 0.0;
14       for(int j = 0; j < N; ++j) {
15         sum += A[i*N+j] * x[j];
16       }
17       c[i] = sum;
18     }
19     // 2. c = c + b
20 #pragma omp target teams distribute parallel for \
21   map(tofrom:c[0:N]) \
22   map(to:b[0:N])\
23     for(int i = 0; i < N; ++i) {
24       c[i] = c[i] + b[i];
25     }
26   }
27 }
```

(c) An OpenMP program with a device data environment.

Figure III.1: An example code computing `c = Ax + b` (continued).

created device data environment. The created device data environment remains until the associated structured block exits. At the exit, the created device data environment is destroyed.

To create a memory object on the target device, a `map` clause is used. OpenMP 4.0 supports four map-types for the `map` clause. The description of each map-type is shown in Table III.1. A cell marked as "X" means that the associated map-type performs the corresponding operation. For example, map-type `from` allocates a memory object on the target device and copies the contents of the object from the

Table III.1: Description of four map-types.

map-type	allocate	copy at the beginning (host→device)	copy at the end (device→host)
`alloc`	X		
`to`	X	X	
`from`	X		X
`tofrom`	X	X	X

target device to the host at the end of the next structured block. In Figure III.1 (b), the contents of `c` are copied from the target device to the host at the end of the first `for` loop because the memory object allocated for `c` in the target device is destroyed at the end of the loop. Then, the contents of `c` are copied from the host to the target device before the execution of the second `for` loop on the target device.

However, this data transfer is useless because there is no modification done to the contents of `c` on the host between the two `for` loops. To avoid unnecessary data transfers, we need to make a device data environment that encloses the two loops. As shown in Figure III.1 (c), the `target data` construct at line 4 does this. This construct only creates a new device data environment, which eliminates the unnecessary data transfers associated with `c`.

The `target data` construct in Figure III.1 (c) creates four memory objects on the target device: `A`, `x`, `b`, and `c`. Since `A`, `x`, and `b` should be copied to the target device before executing loops, they are mapped using `map(to:)`. The map-type `to` allows for the contents of listed memory objects to get copied to the target device at the be-

ginning of the construct. Since the contents of `c` must be updated at the end of the function, it is declared with `map(from:)`. Since all of these objects are pointers, the size of each memory object should be informed to the compiler through array sections in the `map` clauses. The syntax of an array section is `[`*lower-bound*`:`*length*`]`. For example, array `A` should be mapped with `A[0:N*N]` because it is accessed from `A[0]` to `A[N*N-1]`.

According to the OpenMP 4.0 specification, a `map` clause for an already mapped object is ignored by the compiler. Thus, the `map` clauses for `A`, `x`, `c`, and `b` at line 10, 11, 21, and 22 are ignored because they are already mapped by the enclosing `target data` construct at line 4. At the end of the `target data` construct at line 26, the contents of the memory object associated with `c` on the device are copied back to the host by `map(from:c[0:N])` on line 6.

The `target teams distribute parallel for` construct at line 9 in Figure III.1 (c) is a combined construct consisting of three constructs `target`, `teams`, and `distribute parallel for`. The `teams` construct creates a league of thread teams, and the master thread of each team executes the next structured block. Programmers can define the number of teams using `num_teams` and the maximum number of threads in a team using `thread_limit` in this construct. If they are not specified, the default numbers are used. The iterations of the `for` loop at line 12 are distributed to all threads in all teams in the target device due to the `distribute parallel for` construct.

```
1  void vecAdd(double* A, double* B, double* C, int N) {
2    #pragma omp target teams
3    {
4      #pragma omp distribute
5      for(int i=0; i<N; i+=num_threads){
6        #pragma omp parallel for
7        for(int j=i; j<MIN(i+num_threads,N); ++j){
8          C[j] = A[j] + B[j];
9        }
10     }
11   }
12 }
```

(a) A usage of `distribute`.

```
1 void vecAdd(double* A, double* B, double* C, int N) {
2   #pragma omp target teams
3   {
4     #pragma omp distribute parallel for
5     for(int i=0; i<N; ++i) {
6       C[i] = A[i] + B[i];
7     }
8   }
9 }
```

(b) A usage of `distribute parallel for`.

Figure III.2: Usages of `distribute` and `distribute parallel for`. Above programs do the same computation.

The `distribute` construct specifies how the next `for` loop is distributed to the master thread of each team. An example is shown on line 4 in Figure III.2 (a). Iterations of the first loop on line 5 are distributed to each master thread of each team. Due to the `parallel for` construct on line 6, iterations of the inner loop on line 7 are distributed to all threads in each team. Note that although constructs `parallel` and `for` are not included in the set of device constructs, they can be used with device constructs to distribute workload to multiple threads in a team.

The `distribute parallel for` construct combines three constructs `distribute`, `parallel`, and `for`. However, it is slightly different from using the three constructs separately. It specifies how the iterations of the next `for` loop is distributed to all threads in all teams. An example is shown in Figure III.2 (b). Iterations of the loop are distributed to all threads in all teams. Figure III.2 (b) does the same computation as Figure III.2 (a).

III.1.2 Needs for Data Transfer Optimization

Making an efficient data environment for a device is not always straightforward and causes headaches to programmers. There are two difficulties.

The first difficulty is finding optimal locations of device data environments. For example, if the function `MatVecMul()` in Figure III.1 (a) is called at multiple program points, programmers have to determine which call uses which data and where the device data environment should be created to minimize data transfers. The programmers need to track all paths to the function call and identify how the data flows along the paths. It requires a whole-program analysis. If there are multiple source files and pointers are used, making such a device data environment becomes much difficult.

The second difficulty is that programmers should explicitly insert data transfer operations if data should be transferred within the device

data environment. For example, if the host modifies data in a device data environment and the data gets copied to the target device before executing the next kernel, the programmer should specify which data has to be copied to the target device. For example, if the host modifies `c` in Figure III.1 (c) between the two `for` loops, `c` would be brought from the device to the host. After the host modifies `c`, it would be copied to the target device. In this case, the data transfer should be explicitly specified by the programmer. For this explicit data transfer, the `target update` construct can be used as shown in Figure III.3.

The `target update` construct accepts two motion clauses: `to` and `from`. Variables in the motion clause `to` are copied from the host to the target device. Variables in the motion clause `from` are copied from the target device to the host.

Instead of manually creating device data environments properly, this thesis proposes a runtime optimization technique that automatically minimizes data transfers between the host and the target device. It is called *data transfer optimization* (DTM). Consider Figure III.1 (b). It is known that `c`, `A`, `x`, and `b` are not accessed by the host because the two `for` loops are executed on the target device. In the first `for` loop, `A` and `x` are copied to the target device. This is necessary because they are accessed for the first time by the target device. However, it is not required to copy the contents of `c` back to the host because it is not used by the host between the first loop and the second loop. Before the execution of the second loop, we know that the contents of

```
// c = Ax + b
void MatVecMul(double* c, double* A, double* x,
    double* b, int N) {
#pragma omp target data \
  map(to:A[0:N*N], x[0:N], b[0:N]) \
  map(from:c[0:N])
  {
    // 1. c = Ax
#pragma omp target teams distribute parallel for \
  map(to:A[0:N*N], x[0:N]) \
  map(from:c[0:N])
    for(int i = 0; i < N; ++i) {
      double sum = 0.0;
      for(int j = 0; j < N; ++j) {
        sum += A[i*N+j] * x[j];
      }
      c[i] = sum;
    }

    // Bring the result of c[0] from the target device.
#pragma omp target update from(c[0])
    // Modify the value on the host.
    c[0] = c[0] + 3;
    // Update the modified value to the target device.
#pragma omp target update to(c[0])

    // 2. c = c + b
#pragma omp target teams distribute parallel for \
  map(tofrom:c[0:N]) \
  map(to:b[0:N])\
    for(int i = 0; i < N; ++i) {
      c[i] = c[i] + b[i];
    }
  }
}
```

Figure III.3: An example code in which the host modifies data within a device data environment.

c are in the target device and the host has not modified c, hence it is not required to copy c from the host to the target device. On the other hand, since b is accessed by the target device for the first time, it should be copied to the target device. At the time when the host attempts to accesses c later, it is then copied from the target device to the host.

We implement this mechanism by exploiting page faults. Using DTM, we achieve almost the same performance as that of programs where the device data environment creation is optimized manually.

III.2 Mapping OpenMP to OpenCL

III.2.1 Architecture Model

The architecture model of OpenCL and OpenMP is presented in Section II.1.2.1 and Section II.1.1.1, respectively. OpenMP and OpenCL are similar in that they assume a single host device and multiple target devices. However, there are two major differences between them.

One is the memory model. Since OpenCL assumes a distributed memory model unlike OpenMP, data accessed by the target device in a `target` construct gets copied to the device when translating an OpenMP device construct to OpenCL. This can be done without any problem because of `map` clauses in OpenMP. The contents of all variables, including pointers, get copied to the compute device if they are declared with map clauses. Note that the memory locations accessed through a pointer in the `target` construct must be declared with a `map` clause in OpenMP.

The other difference is that OpenCL allows multiple platforms. Unlike OpenMP, OpenCL allows multiple OpenCL platforms to coex-

ist in a single system. To define a mapping from OpenMP to OpenCL, all compute devices in OpenCL are flattened to give the same view of the architecture model of OpenMP. For example, if a system has two AMD GPUs and two NVIDIA GPUs, there are two OpenCL platforms in the system. However, because there is no concept of platforms in OpenMP, the proposed framework shows four target devices (two AMD GPUs and two NVIDIA GPUs) to application developers using OpenMP.

III.2.2 Execution Model

The execution model of OpenCL and OpenMP is presented in Section II.1.1.2 and Section II.1.2.2, respectively. A work-item in OpenCL corresponds to a thread in OpenMP. A work-group in OpenCL corresponds to a team in OpenMP. The only difference comes from the execution between the `teams` construct and the `parallel` construct. All work-items begin their execution when a `teams` construct is encountered. For statements before a `parallel` construct, only the designated master work-item (*e.g.*, the first work-item in a work-group) executes the statements.

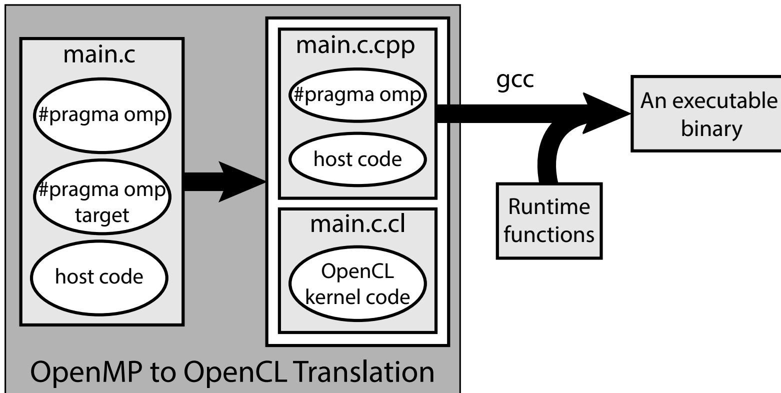


Figure III.4: The translation process.

III.3 Code Translation

This section introduces how an OpenMP program is translated to an OpenCL program. Since both OpenMP and OpenCL express two levels of parallelism, OpenMP programs can be easily translated to OpenCL.

III.3.1 Translation Process

Figure III.4 shows the translation process. Source file `main.c` contains the OpenMP compiler directives including device constructs and other constructs. The proposed translation framework only translates device constructs. The code fragments in a `target` construct are translated to OpenCL kernel functions. The kernel functions are stored in a different file. Code in the `target` construct is replaced with our runtime

```
void func() {
  // shared by all threads in all teams
  int global_shared[N];
  #pragma omp target data map(global_shared[0:N])
  {
    // executed on the host
    stmts;
    #pragma omp target
    {
      #pragma omp teams num_teams(32) thread_limit(32)
      {
        // private to a team
        // shared by all threads in a team
        int team_private[N];
        // executed by a master thread in each team
        #pragma omp distribute
        for(...) {

          #pragma omp parallel
          {
            // private to a thread
            int thread_private[N];
            // executed by a thread
            #pragma omp for
            for(...) { ... }
          }
        }
      }
    }
  }
}
```

Figure III.5: An OpenMP program using the device constructs.

function calls. The runtime functions perform memory object allocation, data transfer, and kernel invocation. Other OpenMP constructs for multicore CPUs remains as the same.

The translated host code is built to an executable binary by using general compilers such as gcc. At the build process, our runtime functions are inserted as a library. The remaining OpenMP constructs can be parallelized by gcc. At the beginning of the program, OpenCL compute devices are initialized and all kernel functions are built.

III.3.2 Translating OpenMP to OpenCL

Figure III.5 shows a template of OpenMP programs that use the device constructs. In the `target data` construct on line 4, an array `global_shared` is created on the target device using a `map` clause. This array is allocated in the global memory in OpenCL. Since the `target data` construct only defines a device data environment, code in this construct is executed on the host device. Code in the `target` construct on line 8 is executed on the target device. The `teams` construct on line 10 defines the number of teams and the maximum number of threads, allowing us to determine the number of work groups and the number of work items using this information. Then, we can launch an OpenCL kernel. Code in this construct is translated to an OpenCL kernel. According to the OpenMP specification, there should be no statements between `teams` and `target`. Hence, there is no problem if code fragments in a `teams` construct are translated to an OpenCL kernel instead of code fragments in a `target` construct.

Code between `teams` and `parallel` is executed by the master thread in each team in OpenMP. This code is executed only by the master work-item in OpenCL. If there are declarations between `teams` and `parallel`, they are allocated in the local memory in OpenCL. The assignment to the variables is executed only by the master thread, but the data can be accessed by any threads in the team. For example, the array `team_private` on line 14 is allocated in the local mem-

ory in OpenCL. The local memory in OpenCL can be accessed only by work-items in the same work-group. Note that programmers can exploit OpenCL's local memory to boost performance even though OpenMP does not introduce any concept of the local memory. Using the local memory in OpenCL is one of the most important techniques to significantly improve performance. If there is a `for` loop, programmers can use the `distribute` construct to distribute loop iterations to master threads.

On line 19, there is a `parallel` construct. Code in the `parallel` construct is executed by all threads in each team. If there are any declarations in the `parallel` construct, they will be allocated in the private memory in OpenCL. The private memory in OpenCL is private to a work-item and is not accessible by other work-items. If there is a `for` loop, programmers can use the `for` construct to distribute loop iterations to all threads in a team.

Until now, we have shown the generic translation process. However, we can also use combined constructs for simplicity as shown in Figure III.1. In this figure, three constructs `target`, `teams`, and `distribute parallel for` are combined to distribute iterations of a `for` loop to all threads in a target device. It is a more natural and easier way to parallelize a loop using OpenMP.

As a result, programmers can implement using either the simple version or the complicated version. For the simple version, program-

```
1  __kernel void vecAdd_cpp_vecAdd_0(__global double* A,
2  __global double* B, __global double* C, __global int* N) {
3    int gid = get_global_id(0);
4    int gws = get_global_size(0);
5
6  #ifdef __CPU__
7    int num_iters = *N;
8    int chunk = num_iters / gws;
9    int start = chunk * gid;
10   int end = start + chunk;
11   if( gid == gws-1 )
12     end = *N;
13   for(int i=start; i<end; ++i) {
14     C[i] = A[i] + B[i];
15   }
16 #endif // __CPU__
17
18 #ifdef __GPU__
19   for(int i=gid; i<*N; i+=gws) {
20     C[i] = A[i] + B[i];
21   }
22 #endif // __GPU__
23 }
```

(a) An OpenCL kernel.

Figure III.6: The OpenCL program translated from the program in Figure III.2 (b).

mers insert a `target teams distribute parallel for` construct immediately before a `for` loop. For the complicated version, expert programmers insert constructs `target`, `teams`, and `parallel` to specific program points in order to optimize performance.

III.3.3 Example of Code Translation

We explain the translation using an OpenMP 4.0 program in Figure III.2 (b). It can be translated to an OpenCL program as shown in Figure III.6.

```
void vecAdd(double* A, double* B, double* C, int N) {
  //#pragma omp target
  {
    int created_A;
    cl_mem m_A = CreateOrGetOpenCLBuffer(&A[0],
      N*sizeof(A[0]), 1, &created_A);
    int created_B;
    cl_mem m_B = CreateOrGetOpenCLBuffer(&B[0],
      N*sizeof(B[0]), 1, &created_B);
    int created_C;
    cl_mem m_C = CreateOrGetOpenCLBuffer(&C[0],
      N*sizeof(C[0]), 0, &created_C);

    //#pragma omp teams
    {
      int created_N;
      cl_mem m_N = CreateOrGetOpenCLBuffer(&N,
        sizeof(N), 1, &created_N);

      OMP2OCL_SetThreadLimit(64);
      OMP2OCL_SetNumTeams(32);
      cl_kernel k = OMP2OCL_GetKernel("vecAdd.cpp",
        "vecAdd", "0");
      clSetKernelArg(k, 0, sizeof(cl_mem), &m_A);
      clSetKernelArg(k, 1, sizeof(cl_mem), &m_B);
      clSetKernelArg(k, 2, sizeof(cl_mem), &m_C);
      clSetKernelArg(k, 3, sizeof(cl_mem), &m_N);
      OMP2OCL_LaunchKernel(k);

      DestroyOpenCLBuffer(&N, sizeof(N), 1, created_N);
    }
    DestroyOpenCLBuffer(&A[0], N*sizeof(A[0]), 0, created_A);
    DestroyOpenCLBuffer(&B[0], N*sizeof(B[0]), 0, created_B);
    DestroyOpenCLBuffer(&C[0], N*sizeof(C[0]), 1, created_C);
  }
}
```

(b) The OpenCL host program.

Figure III.6: The OpenCL program translated from the program in Figure III.2 (b) (continued).

Since the `target teams distribute parallel for` construct as shown in Figure III.1 is a combined construct, our translator internally divides it into three constructs `target`, `teams`, and `distribute parallel for`. Hence, exactly the same technique can be applied to the code in Figure III.1.

Table III.2: Prototypes of runtime functions.

Prototype and description
`cl_mem CreateOrGetOpenCLBuffer(void* host_ptr, size_t size, int copy, int* created);` • If there is a memory object corresponding to `host_ptr` already on the device, it returns the memory object. • Otherwise, it allocates an OpenCL memory object in the device and returns the object.
`void DestroyOpenCLBuffer(void* host_ptr, size_t size, int copy, int created);` • Destroys the memory object corresponding to `host_ptr` only if `created` is non-zero.
`void OMP2OCL_SetThreadLimit(int thread_limit);` • Set the maximum number of threads in a team to `thread_limit`.
`void OMP2OCL_SetNumTeams(int num_teams);` • Set the number of teams to `num_teams`.
`cl_kernel OMP2OCL_GetKernel(const char* filename, const char* funcname, const char* index);` • Returns the corresponding OpenCL kernel object based on `filename`, `funcname`, and `index`.
`void OMP2OCL_LaunchKernel(cl_kernel k);` • Launches the OpenCL kernel `k`. It waits until the kernel finishes.

For the `target` construct, our translator only creates a device data environment. Even though code in the `target` construct should be executed on the target device, our translator converts code in the `teams` construct to an OpenCL kernel. According to the OpenMP 4.0 specification, there must be no statements between constructs `target` and `teams`. Hence, there is no problem even if our translator converts code in the `teams` construct to an OpenCL kernel. The reason is that the number of total work-items and the number of work-items in a work-group (*i.e.*, work-group size) are required to launch an OpenCL kernel, and this information is obtained from the clauses `num_teams` and `thread_limit` declared in the `teams` construct. If there is no `teams` construct in the `target` construct, our translator inserts a `teams` construct with `num_teams(1)` immediately after the `target` construct so that the `teams` construct encloses all statements in the `target` construct. The `for` loop in the `distribute parallel for` construct is

transformed to code that describes work that a work-item has to perform in the OpenCL kernel.

III.3.3.1 Host Code Translation

At line 2 in Figure III.2 (b), there is a clause `map(to:A[0:N], B[0:N])` in the `target` construct. It creates the memory objects corresponding to `A` and `B` in the OpenCL device global memory. They are destroyed at the end of the construct. Our translator converts this memory allocation and deallocation to calls of runtime functions `CreateOrGetOpenCLBuffer()` and `DestroyOpenCLBuffer()`. The prototypes of runtime functions are shown in Table III.2. Their detailed description is presented in Section III.3.3.3

To allocate a memory object, our translator inserts `CreateOrGetOpenCLBuffer()` at the beginning of the construct as shown at line 5-6 in Figure III.6 (b). To destroy the memory object in the device, our translator inserts `DestroyOpenCLBuffer()` at the end of the construct as shown at line 32 in Figure III.6 (b). As a result, our translator converts `map(to:A[0:N], B[0:N])` at line 2 in Figure III.2 (b) to function calls of `CreateOrGetOpenCLBuffer()` and `Destroy OpenCLBuffer()` at line 4-9 and 32-33 in Figure III.6 (b). Since they are declared using the `map(to:)` clause, `copy` of `CreateOrGetOpenCLBuffer()` is set to one while `copy` of `DestroyOpenCLBuffer()` is set to zero.

Code in the `teams` construct at line 7-10 in Figure III.2 (b) is translated to an OpenCL kernel. The name of the kernel is determined as `(file name)_(function name)_(kernel index)` to avoid the name collision of kernels. A dot(.) in the file name is replaced with an underscore(_). The kernel index is determined by the order of occurrence of the `teams` construct in the function, and it starts from zero. An example of a kernel name is "`vecAdd_cpp_vecAdd_0`" as shown at line 1 in Figure III.6 (a).

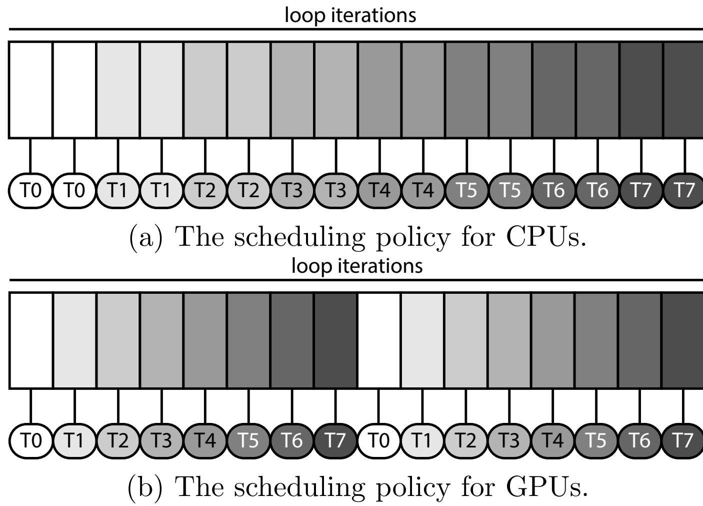
The code at line 7-10 in Figure III.2 (b) is replaced with our runtime function calls. Due to the `thread_limit(64)` clause, `OMP2OCL_SetThreadLimit(64)` is inserted as shown at line 20 in Figure III.6 (b). This function call tells the maximum number of threads in a team to the runtime. If the device does not support the specified maximum number of threads in a team, the OpenMP runtime should adjust the value to the maximum number that the device is able to support. Due to `num_teams(32)` clause, `OMP2OCL_SetNumTeams(32)` is inserted as shown at line 21 in Figure III.6 (b). This function tells the number of teams to the runtime. Using this information, the runtime determines work-group size and the number of total work-items to launch an OpenCL kernel. If they are not specified, the runtime uses their default values. How to determine default values are discussed in Section III.3.3.4.

To launch the kernel, our translator first inserts `OMP2OCL_GetKernel()` (line 22 in Figure III.6 (b)) to get the

OpenCL kernel object. Our runtime searches kernel files in the current working directory, and builds them at the beginning of the program. The function receives three parameters, the file name, the function name, and the kernel index. Based on these parameters, it returns the corresponding OpenCL kernel object.

Then, the arguments of the kernel are set by inserting OpenCL API function `clSetKernelArg()`. Variables that are referenced in the `teams` construct and declared outside of the construct should be inserted to kernel parameters to access the variables in the OpenCL kernel. The code in the `teams` construct at line 7-10 in Figure III.2 (b) accesses `A`, `B`, `C`, and `N`. The memory objects corresponding to `A`, `B`, and `C` are already allocated in the device due to the `map` clauses, but the memory object corresponding to `N` is not allocated. According to the OpenMP 4.0 specification, a variable referenced in a `target` construct that is not declared in the construct is implicitly treated as if it had appeared in a `map` clause with the map-type of `tofrom`[7]. Hence, `N` is treated as it is declared with `map(to:N)`. The memory object corresponding to `N` is allocated in the device as shown at line 16-18 and destroyed at line 30 in Figure III.6 (b). The memory objects are passed as parameters to the kernel as shown at line 24-27 in Figure III.6 (b).

Finally, our translator inserts `OMP2OCL_LaunchKernel()` (line 28 in Figure III.6 (b)). This function launches the kernel on the target device using OpenCL API function `clEnqueueNDRangeKernel()`.



(a) The scheduling policy for CPUs.

(b) The scheduling policy for GPUs.

Figure III.7: Scheduling policies for CPUs and GPUs.

When the kernel is invoked, the runtime determines the number of total work-items and the number of work-items in a work-group based on the number of teams and the maximum number of threads. This function is blocked until the kernel finishes its execution.

III.3.3.2 Kernel code translation

The `for` loop in the `distribute parallel for` construct at line 8-10 in Figure III.2 (b) is transformed to code that describes what a work-item is required to do. The iteration space of the loop is divided into chunks that are approximately equal in size, and each chunk is executed on a work-item. For example, if there are 16 iterations and 8 work-items in a work-group, the first two iterations are executed on the first work-item as shown in Figure III.7 (a). This scheduling

policy performs well for CPUs, but it does not perform well for GPUs. As a result, we should apply different scheduling policies for different types of devices to enhance performance portability. Further detail is discussed later in Section III.3.5. Code inside the `for` loop remains the same.

If constructs `distribute` and `parallel for` are used separately as shown in Figure III.6 (a), the `for` loop in each construct is transformed to code following the scheduling policy as described before.

III.3.3.3 Description of Runtime Functions

CreateOrGetOpenCLBuffer(). It first checks whether the memory object corresponding to `host_ptr` is already allocated in the device or not. Our runtime maintains mapping information between the host memory region and the memory object in the target device. If the corresponding memory object exists in the target device, the function just returns the memory object. Note that `cl_mem` is the type of an OpenCL memory object. If the corresponding memory object does not exist, the function allocates a memory object with the specified `size` using OpenCL API function `clCreateBuffer()`. This new mapping information is recorded in our runtime. If `copy` is not zero, the function copies `size` bytes from the memory region pointed by `host_ptr` to the newly allocated memory object in the device using OpenCL API function `clEnqueueWriteBuffer()`. The variable `copy` is not zero if

the map-type of the memory object is `to` or `tofrom`. Otherwise, `copy` is set to zero. For the last parameter `created`, an integer variable per memory object is declared before calling the function (line 4 in Figure III.6 (b)). This function sets `created` to a non-zero value if a new memory object is allocated. Otherwise, `created` is set to zero.

The variable `created` is required to determine whether the corresponding memory object should be destroyed in `DestroyOpenCLBuffer()` or not. According to the OpenMP 4.0 specification, if the corresponding memory object is already allocated in the enclosing device data environment, the new data environment uses the already allocated memory object. No additional storage is allocated and neither initialization nor assignment is performed, regardless of the map-type[7]. For example, the function calls of `CreateOrGetOpenCLBuffer()` and `DestroyOpenCLBuffer()` for `A` are inserted twice due to `map` clauses at line 5 and 10 in Figure III.1 (c), but only the function calls for the `map` clause at line 5 should allocate and deallocate the corresponding memory object in the device. The other function calls for the `map` clause at line 10 should not be performed. The variable `created` in `CreateOrGetOpenCLBuffer()` for `A` at line 10 is set to zero because it is already allocated by `CreateOrGetOpenCLBuffer()` for `A` at line 5. `DestroyOpenCLBuffer()` for `A` at line 10 does nothing because `created` is passed as the value of zero.

DestroyOpenCLBuffer(). It first checks `created`. If it is zero, this function immediately returns because the memory object corresponding to `host_ptr` is allocated in another construct. Note that the variable `created` is passed from the last parameter of `CreateOrGetOpenCLBuffer()`. If it is non-zero, it means the memory object is allocated in this construct. Hence, the memory object should be destroyed in this function. If `copy` is non-zero, the contents of the corresponding memory object in the device are copied to the host using OpenCL API function `clEnqueueReadBuffer()`. The variable `copy` is not zero if the map-type of the memory object is `from` or `tofrom`. Then, the memory object is destroyed in the device using OpenCL API function `clReleaseMemObject()`. The mapping information between the host memory region and the memory object in the device is also removed.

III.3.3.4 Determining Default Values

Default value of thread_limit. Since the optimal default value of `thread_limit` can be different for each application, we empirically select the number for each device. We choose the size of a wavefront (*i.e.*, 64) for the AMD GPU while we choose the size of a warp (*i.e.*, 32) for the NVIDIA GPU. For the Intel CPU, we choose the number of words in a cache block (*i.e.*, 8).

Table III.3: Formula that calculates the default number of teams.

Work-group size = (maximum work-group size) / (thread_limit)
*Number of parallel work-groups = (number of CUs) * (work-group size)*
*Default number of teams = (number of parallel work-groups) * (multiplier)*

Default value of num_teams. The default value of `num_teams` is calculated as shown in Table III.3. *Work-group size* is determined as the maximum work-group size divided by the value of `thread_limit`. *Parallel work-groups* is work-groups that can be scheduled together on the device. The number of parallel work-groups is defined as the number of CUs multiplied by the work-group size. The maximum work-group size and the number of CUs are obtained from OpenCL API function `clGetDeviceInfo()`.

Due to load imbalance, it is not good to create the number of teams equal to the number of parallel work-groups, hence we introduce *multiplier*. If the multiplier is one, the number of parallel work-groups and the default number of teams are the same. In this case, all teams are scheduled to the device at once. No more scheduling is required. If the execution time of each team varies a lot, the performance would be bad due to load imbalance. If the multiplier is large (*i.e.*, the default number of teams is large), the scheduling overhead of work-groups increases in the OpenCL runtime. Hence, there is a trade-off of determining the multiplier. In our framework, we find the multiplier empirically as 16. There is no single value to perform the best for all applications, and the optimal value of the multiplier can be different for each application.

Table III.4: Actions according to events to minimize data transfers.

Event	Action

map(to:)	In `CreateOrGetOpenCLBuffer()`
	• If no corresponding memory object exists,
	  • Allocate the memory object.
	  • Copy the data to the target device.
	• If the corresponding memory object exists,
	  • If it is marked as modified,
	    • Copy the data to the target device.
	    • Remove the modification mark.
	• Remove write permissions of pages of the memory region.
	In `DestroyOpenCLBuffer()`
	• Do not destroy the memory object in the device.

map(from:)	In `CreateOrGetOpenCLBuffer()`
	• If no corresponding memory object exists,
	  • Allocate the memory object.
	• If the corresponding memory object exists,
	  • Do nothing.
	In `DestroyOpenCLBuffer()`
	• Do not destroy the memory object in the device.
	• Remove read permissions of pages of the memory region.

A read fault	The corresponding memory object is copied (device→host). Give read permissions to the pages of the memory region.
A write fault	The corresponding memory object is marked as modified. Give write permissions to the pages of the memory region.

If the number of teams required for a loop is smaller than the default number of teams, the required number of teams is used instead. The number of teams required for a loop is calculated from the number of iterations of the loop divided by the value of `thread_limit`. For example, if only two teams are required to execute a loop and the default number of teams is 16, the number of teams for the loop is set to two. It is because teams other than the first two teams have nothing to work.

III.3.4 Data Transfer Minimization (DTM)

As discussed in Section III.1.2, it is important to bind multiple `target` constructs within a `target data` construct in order to minimize data transfers between the host and the target device. However, it is hard to find the location of the `target data` construct to minimize data transfers. It becomes more serious if the call depth is deep and source code is separated in multiple files.

To eliminate this difficulty, we provide a runtime technique that cleverly minimizes data transfers between the host and the target device automatically. It is called *data transfer minimization* (DTM). There are two basic principles: 1) bring data from the target device to the host when actually the host accesses it. 2) update data from the host to the device only if the host modifies the data and the device needs the data.

In order to detect whether the host reads or writes memory regions in the host, we exploit the page fault mechanism. We can detect a page fault at the application level using the system call in Linux. The detailed actions for events are described in Table III.4. The `map(tofrom:)` clause can be considered as a combination of `map(to:)` and `map(from:)`. The `map(alloc:)` clause does not require data transfers.

In addition, the memory object should not be destroyed even though its device data environment is destroyed. This increases the possibility to reuse the same memory object, but it has a disadvantage. It increases the memory footprint on the target device. This can make the situation that the target device does not have enough memory to allocate a new memory object even though the memory object can be allocated without DTM. When it happens, we reclaim memory objects whose device data environment has been already destroyed in the order of LRU (*i.e.*, least recently used).

Illustration of DTM. Using this technique, the numbers of data transfers are the same between Figure III.1 (b) and (c). We describe how it operates using the program in Figure III.1 (b) with actions described in Table III.4. At line 5 in Figure III.1 (b), `A` and `x` are declared using `map(to:)`. At the beginning of the construct, `CreateOrGetOpenCLBuffer()` checks that the corresponding memory object exists in the device. Since this is the first time that they are declared, there are no memory objects corresponding to them in the device. Hence, the corresponding memory objects are allocated on the device and data is copied from the host to the device. Then, write permissions of pages for them are removed. At the end of the construct, the corresponding memory objects are not destroyed in `DestroyOpenCLBuffer()`.

Since `c` is declared using `map(from:)` for the first time, the corresponding memory object is allocated in the device in

`CreateOrGetOpenCLBuffer()` at the beginning of the construct. At the end of the `target` construct, `c` is not copied back to the host in `DestroyOpenCLBuffer()`. The corresponding memory object is not destroyed, and the read permissions of pages of `c` are removed.

At line 16 in Figure III.1 (b), `c` is declared using `map(tofrom:)`. At the beginning of the construct, `CreateOrGetOpenCLBuffer()` is called. Since the memory object corresponding to `c` exists and it is not modified by the host, the function does not copy data to the target device. The write permissions of pages of `c` are removed. At the end of the construct, `DestroyOpenCLBuffer()` is called. The corresponding memory object is not destroyed and the read permissions of pages of `c` are removed.

Since `b` is declared using `map(to:)` for the first time, the corresponding memory object is allocated and copied in `CreateOrGetOpenCLBuffer()` at the beginning of the construct. The write permissions of pages of `b` are removed. The corresponding memory object is not destroyed in `DestroyOpenCLBuffer()` at the end of the construct.

When the host accesses `c` later, a read fault occurs. In the read fault handler, the contents of `c` in the device are copied to the host using OpenCL API function `clEnqueueReadBuffer()`. It gives read permissions to the pages of `c`.

If the host attempts to modify `A`, a write fault occurs. In the write fault handler, `A` is marked as modified. It gives write permissions to the pages of `A`. When `MatVecMul()` is called once again, `A` is copied from the host to the device in `CreateOrGetOpenCLBuffer()` because `A` is marked as modified. Then, the modification mark of `A` is removed to prohibit unnecessary copies in the future.

Host memory sharing. In addition, if the host and the target device share the main memory, it does not need to data transfer between them. For example, CPUs can be an OpenCL compute device. If the host and the target device share the main memory, the data transfer is not performed and the original memory region is used. Our runtime detects the sharing information using OpenCL API function `clGetDeviceInfo()`.

Minor discussions. If the host memory region is in the stack, it is not possible to remove read or write permissions of the memory region. In this case, we disable DTM for the memory region. If two host memory regions share a page and a page fault occurs for that page, actions are performed for both memory regions. This is because the page fault handler should give proper access permissions to the pages of the first memory region after handling the page fault. In this case, we may not capture a page fault for the other memory region. To reduce this behavior, our runtime replaces the original dynamic memory allocator with our dynamic memory allocator. Our dynamic memory allocator avoids the page-sharing between memory regions.

III.3.5 Performance Portability Enhancement (PPE)

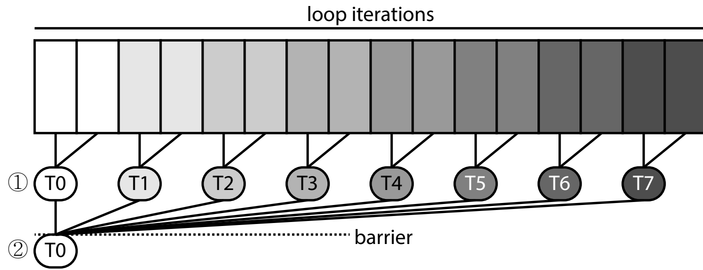
We propose the efficient scheduling and reduction policies for CPUs and GPUs. Using our translation framework, we can achieve both performance portability and code portability for OpenMP programs.

Scheduling policy. When a loop is translated to an OpenCL kernel by our translation framework, the scheduling policy of the loop should be carefully considered because an OpenCL kernel describes instructions from a work-item point of view. It is known that OpenCL does not have performance portability[59, 65, 75]. In other words, an OpenCL kernel developed for CPUs may not perform well for GPUs. The main reason is that their execution mechanism at the hardware level is different. Therefore, different scheduling policies should be applied for different architectures.

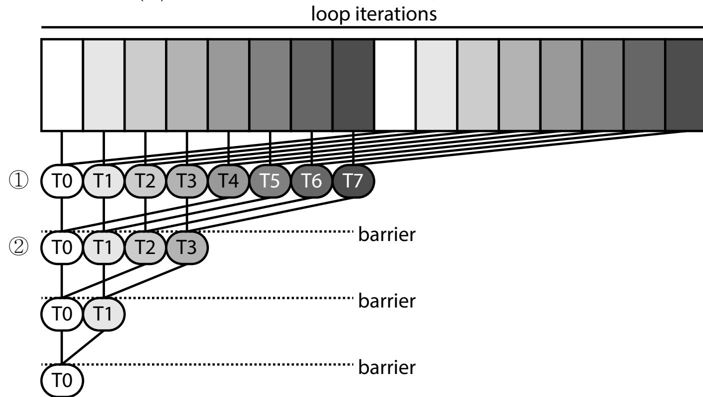
For example, if we parallelize a loop, it is the best option to distribute a chunk of iterations to a thread for CPUs to increase cache utilization[65, 66] as described in Figure III.7 (a). In OpenCL implementations for multicore CPUs, a CPU core is mapped to a CU[34, 42]. Hence, work-items in a work-group assigned to a CU are executed one by one. Based on this execution mechanism, it is best for CPUs that a work-item performs contiguous iterations as many as possible to increase cache utilization. Note that, in general, adjacent iterations tend to access adjacent memory locations. The example code is shown at line 7-15 in Figure III.6 (a).

However, it does not perform well for GPUs. It is better to distribute a chunk of iterations to a group of threads in a team for GPUs to exploit coalesced memory accesses[39, 60, 74]. For example, if there are 16 iterations and 8 work-items in a work-group, the first and ninth iterations are executed on the first work-item as shown in Figure III.7 (b). Since GPUs have PEs in hardware unlike CPUs, multiple work-items can be executed in parallel. Moreover, if multiple work-items access a contiguous memory block at the same time, the access requests are *coalesced*[8, 52]. This greatly reduces memory access latency. Based on this execution mechanism, it is best for GPUs that a group of work-items performs a group of iterations. The example code is shown at line 19-21 in Figure III.6 (a).

In order to obtain good performance for both CPUs and GPUs for a single OpenCL kernel, programmers should implement and maintain two different kernels for each hardware architecture. On the other hand, if the loop with OpenMP device constructs is translated to an OpenCL kernel by our translation framework, the burden is removed from programmers. Programmers only consider the algorithm and choose which loops to be parallelized. Our translator generates an OpenCL kernel for the target hardware. For GPUs, it distributes every single iteration to a work-item in a round robin manner. For CPUs, it distributes a chunk of iterations to a work-item. For another architecture, it can distribute loop iterations for the architecture.



(a) The reduction policy for CPUs.



(b) The reduction policy for GPUs.

Figure III.8: Reduction policies for CPUs and GPUs.

Our translation framework does this using conditional compilation. It generates code for both CPUs and GPUs, and code for each device type is encapsulated with the specific symbol. The example is shown at line 6-22 in Figure III.6 (a). When OpenCL kernels are built by our runtime, the symbol `__CPU__` or `__GPU__` is defined according to the type of the target device.

Reduction policy. The optimal reduction code can be different depending on the type of the target device. Figure III.8 shows well-known performance-efficient reduction policies for CPUs and GPUs[3]. For CPUs, each work-item performs the reduction for the assigned

iterations by following the efficient scheduling policy for CPUs (①) as shown in Figure III.8 (a). This makes a single value for each work-item. Then, the first work-item performs the reduction from the result of all work-items after a barrier (②). This is the best option because a CPU core performs work-items sequentially.

For GPUs, each work-item performs the reduction for the assigned iterations by following the efficient scheduling policy for GPUs (①) as shown in Figure III.8 (b). This makes a single value for each work-item. Since GPUs can execute multiple work-items simultaneously, it is the best option to perform the reduction in parallel. The first half of work-items perform the reduction with the values of the second half of work-items after a barrier (②). This process continues until only one work-item remains. To eliminate the overhead of synchronization caused by barriers, it is possible to remove barriers if the number of work-items is smaller than the size of a *warp* or *wavefront*. A warp for NVIDIA GPUs or a wavefront for AMD GPUs is a group of work-items that are executed simultaneously by hardware.

III.4 Performance Evaluation

In this section, we evaluate our OpenMP 4.0 device constructs translation. First, we show the effectiveness of the optimization techniques. Next, we compare the performance of the translated programs with the performance of the original OpenCL and OpenMP programs. In

addition, we compare the performance with the PGI compiler for practicality.

III.4.1 Evaluation Methodology

Applications. For our evaluation, we use applications from Rodinia 3.0[27, 28]. Rodinia provides programs written in OpenMP, CUDA, and OpenCL. Since the original OpenMP programs do not use OpenMP 4.0 device constructs, we manually modify the programs to use the device constructs. In this modification, we insert the combined construct `target teams distribute parallel for` immediately before a `for` loop that needs to be parallelized.

There are some applications that use double pointers, but OpenMP 4.0 does not allow using double pointers in a `map` clause. According to the OpenMP specification, a memory region in a `map` clause should be contiguous, and the memory region pointed by a double pointer may not be contiguous. We convert double pointers to single pointers for these applications.

We tried to port all OpenMP programs in Rodinia, but we were unable to port two programs (`leukocyte` and `mummergpu`). A library that contains double pointers is used in `leukocyte`, and `mummergpu` uses CUDA API functions in its source code. In addition, `nn` reads the contents of a file continuously in the `for` loop in the original OpenMP version. However, it is not possible to read a file in an OpenCL kernel,

Table III.5: Applications used.

Application	Problem size
lavaMD	20x20x20 boxes
myocyte	xmax=70, 1000 workloads
nn	33,554,432 records
streamcluster	64K points, 256 dimensions
b+tree	128M nodes for j, 64K nodes for k
kmeans	800K points, 34 features
backprop	2M input nodes
hotspot	8Kx8K points
particlefilter	400,000 particles
bfs	128K nodes
heartwall	test.avi
srad	An 502x458 image
pathfinder	100,000 width, 100 steps
nw	2Kx2K data points
cfd	missile.domn.0.2M
lud	A 4Kx4K matrix
jacobi	A 4Kx4K mesh

we modify the program to read all contents in the file before launching the kernel.

In addition, we add another application `jacobi` from OpenACC Programming and Best Practices Guide[55]. In this case, we convert OpenACC directives to OpenMP 4.0 device constructs. Table III.5 summarizes applications used for our evaluation and their problem sizes.

We implement two versions of OpenMP programs: *base* and *hand-tuned* versions. The base version is the program where the combined construct is inserted immediately before `for` loops (Figure III.1 (b)). The hand-tuned version is the program where device data environment

is inserted to avoid unnecessary data transfer in addition to the base version (Figure III.1 (c)). We will show that the base version with our optimization techniques performs similar to the hand-tuned version in Section III.4.2. Since implementing the hand-tuned version is much more complicated than the base version, our translation framework further lowers programming complexity of OpenMP without performance degradation.

As far as we know, there is no available compiler that supports OpenMP 4.0 device constructs correctly and completely. Even though gcc claims that it supports OpenMP 4.0 offloading features since gcc 5.2[2], current gcc does not have the functionality that offloads computations to GPUs. Thus, to show the effectiveness of our OpenMP 4.0 compiler, we choose the PGI compiler[5] to compare with. Since the PGI compiler only supports OpenACC, we port all the hand-tuned versions of the applications but `jacobi` to OpenACC by hand. Most of OpenACC directives have one-to-one correspondences to OpenMP 4.0 directives.

For all applications, we measure the wall clock time of the entire execution without I/O time. I/O time takes a major portion of the execution time for some applications, and I/O time is not a focus of this evaluation. Each application is executed 10 times and their average is reported.

Table III.6: Evaluation environment.

CPU	2x Intel 2.0Ghz octa-core Xeon E5-2650
Memory	128GB
OS	CentOS 7
GPU	AMD Radeon R9 390x NVIDIA GeForce GTX Titan X
OpenCL	Intel OpenCL 15.1 (work-group size = 8) AMD APP SDK 3.0 (work-group size = 64, 256 for PGI) NVIDIA driver 352.55 (work-group size = 32, 128 for PGI)
GCC	4.8.3
PGI compiler	15.10

Environment. We modify clang 3.5.1[46] to implement our source-to-source translation framework. The evaluation environment is shown in Table III.6. We use two GPUs for the evaluation. In total, we have three OpenCL devices: a CPU device from Intel, a GPU device from AMD, and a GPU device from NVIDIA. We use the OpenCL implementation from the hardware vendor. For example, we use Intel's OpenCL implementation for the Intel CPU and NVIDIA's OpenCL implementation for the NVIDIA GPU.

For each device, we choose the default work-group size as shown in Table III.6. For the fair comparison, we try to use the same values for the number of teams and the number of threads for the PGI OpenACC compiler, but programs built by the PGI compiler does not perform well if the number of threads in a team is explicitly specified. Hence, We have to use default numbers for the number of teams and the maximum number of threads in a team of the PGI compiler. The values are obtained using profilers from AMD and NVIDIA. The default number of threads in a team is 256 for the NVIDIA GPU and 128 for the AMD GPU. Also, we measure the performance of our

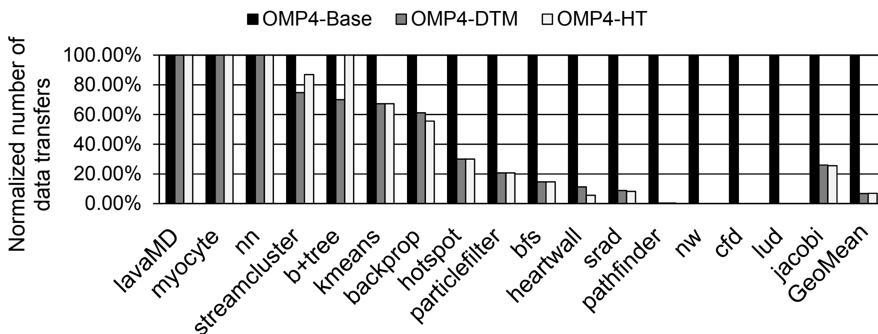


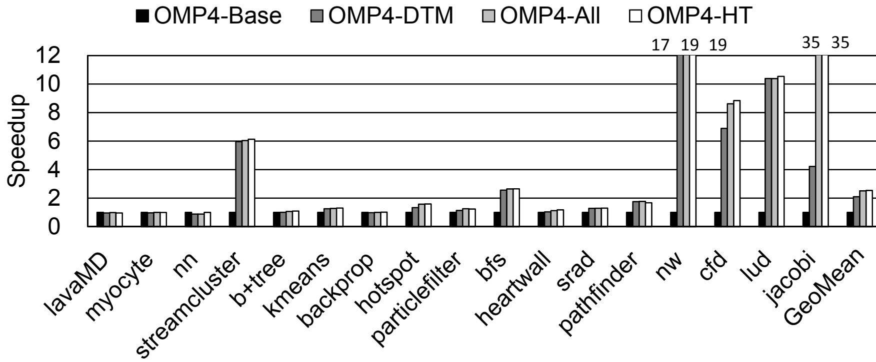
Figure III.9: Reduction in the number of data transfers with DTM on the AMD GPU.

target OpenCL programs with the default value of the PGI compiler. Note that our translation framework performs well if the number of teams and the number of threads in a team explicitly are specified. The default number of teams for PGI is the number of iterations of a loop divided by the number of threads in a team. For CPUs, the PGI compiler directly executes the code on the CPU like OpenMP.

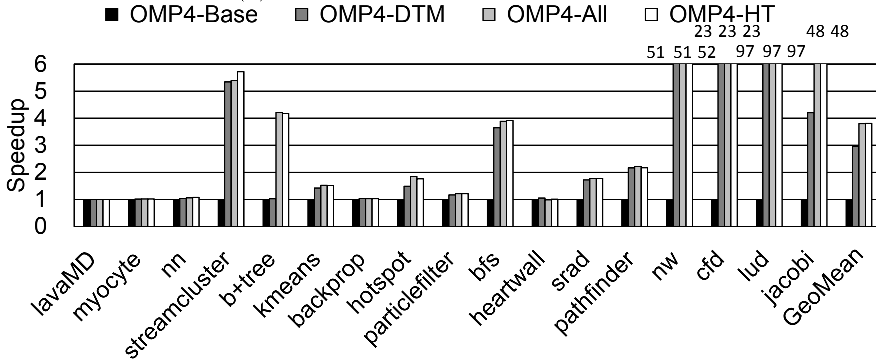
III.4.2 Effectiveness of Optimization Techniques

Reduced number of data transfers. Figure III.9 shows the effectiveness of DTM. Since the reduction in the number of data transfer is important, we show the result only for the AMD GPU. OMP4-Base and OMP4-DTM show the result of the base version without and with DTM, respectively. OMP4-HT show the result of the hand-tuned version.

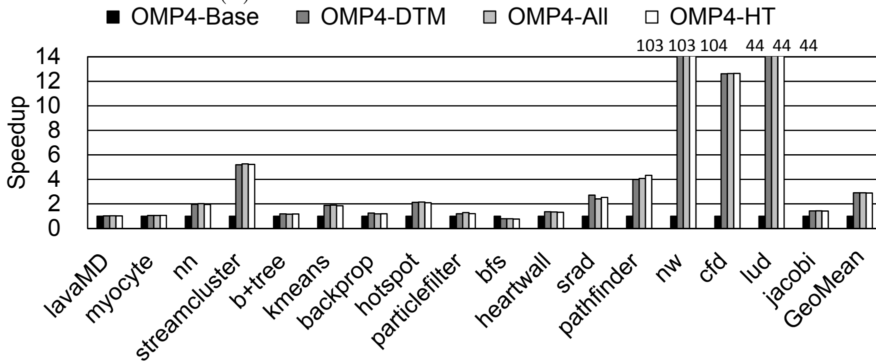
We obtain the numbers using AMD's OpenCL profiler. The number of data transfers is normalized to OMP4-Base. For all applications,



(a) Performance on the AMD GPU.



(b) Performance on the NVIDIA GPU.



(c) Performance on the Intel CPU.

Figure III.10: Effectiveness of optimization techniques.

the reduction in the number of data transfers by DTM (OMP4-DTM) is comparable to OMP4-HT. Surprisingly, the number of data transfers from OMP4-DTM is smaller than that of OMP4-HT for three applications: `streamcluster`, `b+tree`, and `cfd`. This indicates that it is not always easy to manually insert device data environments properly as described in Section III.3.4. DTM reduces the number of data transfers by 93% on average.

III.4.2.1 Performance on the AMD GPU

Figure III.10 shows the execution time of various versions normalized to that of OMP4-Base. OMP4-SCHED represents the speedup that PPE in Section III.3.5 is applied. OMP4-DTM represents the version that DTM in Section III.3.4 is applied. OMP4-All represents the version that both DTM and PPE are applied. OMP4-HT represents the hand-tuned version with PPE.

Performance improvement of PPE. Figure III.10 (a) shows the speedup of applications on the AMD GPU. OMP4-SCHED improves performance by 4% on average compared to OMP4-Base. This is not satisfactory. There are three reasons.

The first reason is that there are not enough iterations of loops. It means that each work-item takes only one iteration of the loop. In this case, there is no difference between two scheduling policies in Figure III.7. There are four applications (`backprop`, `srad`, `nw`, and

lud) belonged to this case.

The second reason is that the memory access pattern in the kernel is not coalesced. The examples are arrays of structures (*e.g.*, a[i].x) and random accesses (*e.g.*, a[b[i]]). In this case, we cannot obtain benefit from the scheduling policy for GPUs. There are seven applications (lavaMD, myocyte, nn, streamcluster, b+tree, heartwall, and pathfinder) belonged to this case.

The third reason is that the performance improvement takes small portion of the execution time because the data transfer time takes a major portion. In this case, the performance improvement is significant when DTM is applied together. There are six applications (kmeans, hotspot, particlefilter, bfs, cfd, and jacobi) belonged to this case. As shown in Figure III.10 (a), OMP4-All (both optimizations are applied) for these applications shows noticeable performance improvement compared to OMP4-DTM. Especially, jacobi shows significant speedup (4x in OMP4-DTM to 35x in OMP4-All) when PPE is applied with DTM.

Performance improvement of DTM. OMP4-DTM improves performance by 110% on average compared to OMP4-Base. The performance improvement is significant for seven applications (streamcluster, bfs, pathfinder, nw, cfd, lud, and jacobi). This shows that the performance is not proportional to the reduction ratio because the time consumed by data transfer takes different portion

of the execution time for different applications. Of course, the applications that show large speedup have noticeable reduction in the number of data transfers.

Performance improvement of both optimization techniques. OMP4-All improves performance by 152% on average compared to OMP4-Base. Since OMP4-HT also applies enhancing performance portability, the speedup is similar between OMP4-All and OMP4-HT. Note that the performance difference between them is how they minimize data transfers (*i.e.*, DTM vs. hand-tuned). The reduction ratio of DTM is almost the same as that of the hand-tuned version as shown in Figure III.9, so the reduced execution time must be similar. Hence, the speedup must be similar between OMP4-All and OMP4-HT.

III.4.2.2 Performance on the NVIDIA GPU

Figure III.10 (b) shows the speedup of applications on the NVIDIA GPU. Basically, the speedup is similar compared to the speedup on the AMD GPU. There is one exception. Surprisingly, `b+tree` shows significant performance improvement for OMP4-SCHED. After our inspection using NVIDIA's OpenCL profiler, the total execution time of kernels is reduced by 97% compared to OMP4-Base. Hence, the scheduling policy for GPUs is more effective than the scheduling policy for CPUs for `b+tree` on the NVIDIA GPU. On average, the perfor-

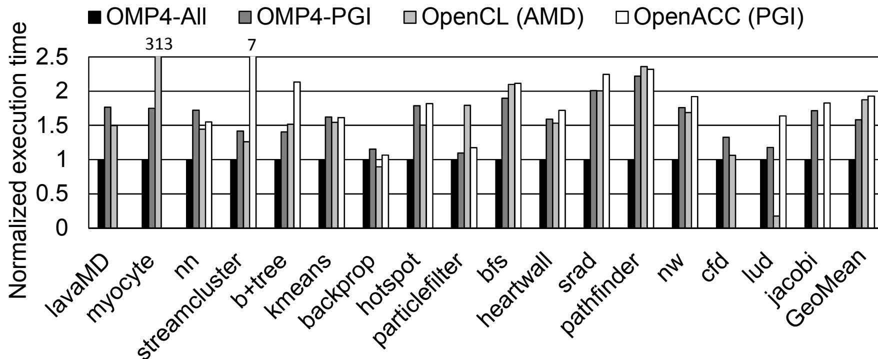
mance improvement is 11% for OMP4-SCHED, 196% for OMP4-DTM, and 280% for OMP4-All.

III.4.2.3 Performance on The Intel CPU

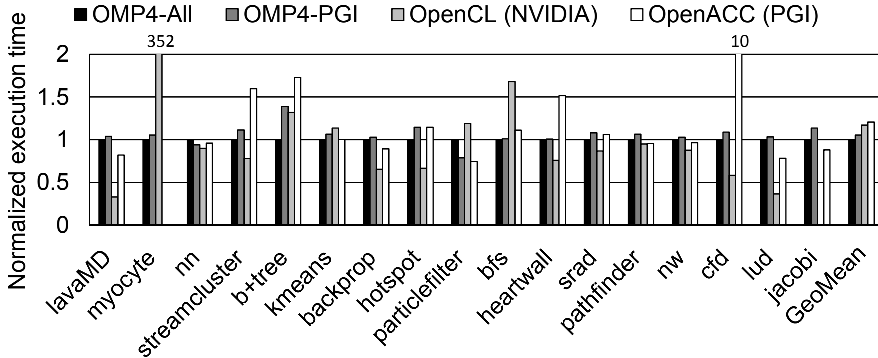
Figure III.10 (c) shows the speedup of applications on the Intel CPU. For CPUs, OMP4-SCHED is not different from OMP4-Base because the base scheduling policy is for CPUs in PPE. OMP4-DTM is effective because data transfer does not happen with OMP4-DTM because the host and the CPU device share the main memory. On average, the performance improvement is 0% for OMP4-SCHED, 190% for OMP4-DTM, and 190% for OMP4-All.

III.4.3 Comparison with Other Implementations

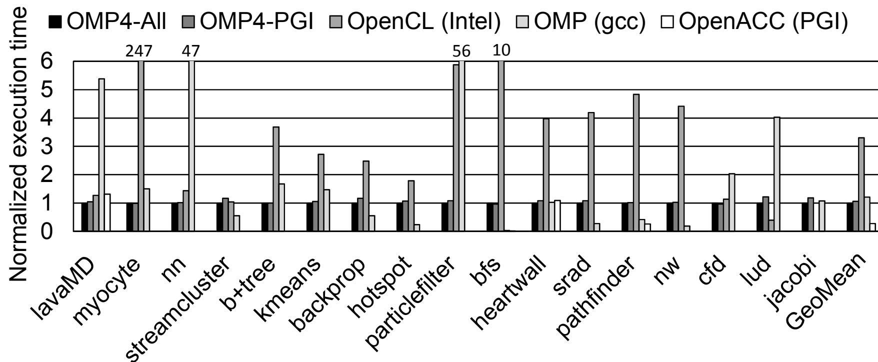
Since Rodinia provides both OpenCL and OpenMP versions for the same application, we can compare the performance of the target OpenCL code to that of the original OpenCL version to see the effectiveness of our translation framework. Note that the original OpenMP version does not use device constructs, so it is executed only on the Intel CPU. The OpenCL version can be executed on all three devices: the Intel CPU, the AMD GPU, and the NVIDA GPU. Figure III.11 shows the comparison result. OMP4-All stands for the execution time of our translation result with optimization techniques. The execution time is normalized by the execution time of OMP4-All. OMP4-PGI rep-



(a) The performance on the AMD GPU.



(b) The performance on the NVIDIA GPU.



(c) The performance on the Intel CPU.

Figure III.11: Performance comparison to other implementations.

resents the normalized execution time of our translation result with the default number of teams and the default number of threads of the PGI compiler. OpenCL (AMD), OpenCL (NVIDIA), and OpenCL (Intel) represent the normalized execution time of original OpenCL version executed on each hardware vendor's OpenCL framework. OMP (GCC) represents the normalized execution time of original OpenMP applications built by gcc. OpenACC (PGI) represents the normalized execution time of OpenACC programs built by the PGI compiler.

III.4.3.1 Performance on the AMD GPU

For the AMD GPU, the result is shown in Figure III.11 (a). Since `jacobi` just has OpenMP and OpenACC versions, it does not have a bar that corresponds to the original OpenCL version.

Comparison with OMP4-PGI. Only the difference between OMP4-All and OMP4-PGI is the number of teams and the number of threads in a team. On average, OMP4-PGI is 59% slower than OMP4-All. This indicates that our default numbers are better than the default numbers of the PGI compiler. It is worth noting that The AMD GPU is very sensitive the number of teams and the number of threads in a team compared to the NVIDIA GPU.

Comparison with OpenCL (AMD). Except two applications (`backprop` and `lud`), the original OpenCL applications (OpenCL (AMD)) are slower than OMP4-All. There are two reasons.

The first reason is that the number of work-items in a work-group is hard-coded in the program, and the value is not optimized well for the AMD GPU.

The second reason is that the number of total work-items is defined as the number of iterations of the original loop for each kernel. That is, each kernel describes an iteration of the loop. This is the same as how the PGI compiler distributes iterations of the loop. This can increase scheduling overhead of work-groups as described in Section III.3.3.

Even though original OpenCL applications are optimized by hand when they are ported to OpenCL, OMP4-All performs better than OpenCL (AMD) due to the inefficient number of total work-items and the inefficient number of work-items for kernels.

The application `myocyte` provides two modes of parallelization. The first mode utilizes only two threads due to the data dependences while the second utilizes more threads. The original OpenCL version only parallelizes the first mode while OMP4-All parallelizes the second mode because of the original OpenMP version. As shown in Figure III.11 (a), this makes OMP4-All much faster than OpenCL (AMD) for `myocyte`.

For `backprop`, OpenCL (AMD) is 10% faster than OMP4-All. In this program, the same `for` loop is performed twice. The number of iterations for the second is one, and this execution is performed on the host in OpenCL (AMD) while it is performed on a GPU in OMP4-

All. The OpenCL runtime overhead and the performance difference between the CPU and the GPU make the performance difference.

For `lud`, OpenCL (AMD) is 82% faster than OMP4-All because OpenCL (AMD) effectively utilizes the GPU local memory. Overall, OpenCL (AMD) is, on average, about 88% slower than OMP4-All.

Comparison with OpenACC (PGI). Since the PGI compiler does not have DTM, for fair comparison, we port the hand-tuned version of OpenMP 4.0 programs to OpenACC. The PGI compiler fails to compile `lavaMD`, `myocyte`, and `cfd`. There are two reasons of the performance difference between OMP4-All and OpenACC (PGI).

The first reason comes from the number of teams and the number of threads in a team. For most applications except three (`streamcluster`, `b+tree`, and `lud`), the performance is similar between OMP4-PGI and PGI. As discussed before, the performance difference between OMP4-All and OMP4-PGI is due to these numbers. This indicates that OMP4-PGI can perform better if the PGI compiler is able to use our default values.

The second reason comes from the execution time of kernels. Since we do not know how the PGI compiler generates binaries for the AMD GPU, we use AMD's GPU profiler. As a result, the performance difference mostly comes from the execution time of kernels. Overall, OpenACC (PGI) is 93% slower than OMP4-All on average.

III.4.3.2 Performance on the NVIDIA GPU

Figure III.11 (b) shows the result on the NVIDIA GPU. Note that `jacobi` does not have OpenCL (NVIDIA) for the same reason of the AMD GPU.

Comparison with OMP4-PGI. Unlike the AMD GPU, the NVIDIA GPU is less sensitive to the number of teams and the number of threads in a team. As a result, the performance difference between OMP4-All and OMP4-PGI is 5% on average.

Comparison with OpenCL (NVIDIA). Like the AMD GPU, OpenCL (NVIDIA) is faster than OMP4-All for `backprop` and `lud` while OpenCL (NVIDIA) is slower than OMP4-All for `myocyte`.

OpenCL (NVIDIA) is faster than OMP4-All for other three applications (`lavaMD`, `hotspot`, and `cfd`). It is because the kernel execution time of OpenCL (NVIDIA) is faster than OMP4-All . The execution time of kernels are obtained from NVIDIA's OpenCL profiler. This means that NVIDIA's OpenCL compiler generates better code for OpenCL (NVIDIA) for these applications. Meanwhile, OMP4-All is faster than or comparable to OpenCL (AMD) for `lavaMD`, `hotspot`, and `cfd`. Also, the performance gap for `lud` on the AMD GPU is much bigger than that on the NVIDIA GPU. This indicates that OpenCL does not have performance portabilty. Tuning an application for a specific device may not be equally effective on a device of the same

kind from different vendor. Overall, OpenCL (NVIDIA) is 17% slower than OMP4-All on average.

Comparison with OpenACC (PGI). The PGI compiler fails to compile only one application, `myocyte`. For all applications but four applications (`streamcluster`, `b+tree`, `heartwall`, and `cfd`), OMP4-All is comparable to OpenACC (PGI). For those four applications, the performance difference comes from the execution time of kernels. Our framework generates better kernel code for them. Overall, OpenACC (PGI) is 21% slower than OMP4-All on average.

III.4.3.3 Performance on the Intel Multicore CPU

Since we can execute the original OpenMP version (*i.e.*, the version that does not contain OpenMP 4.0 device constructs) on the Intel CPU, there are five bars: OMP4-All, OMP4-PGI, OpenCL (Intel), OMP (gcc), and OpenACC (PGI). However, similar to previous cases, `jacobi` does not have OpenCL (Intel).

Comparison with OMP4-PGI. Since the CPU device is insensitive to the number of teams and the number of threads in a team, the performance difference between OMP4-All and OMP4-PGI is 6% on average.

Comparison with OpenCL (Intel). OMP4-All is faster than OpenCL (Intel) for all applications except `lud`. There are two reasons

for this.

The first reason is that the original version of OpenCL is typically implemented having discrete GPUs in mind, and the GPU does not share main memory with the host processor. Thus, there are unnecessary data transfer operations when the original OpenCL version is executed on multicore CPUs.

The other is that, as discussed in Section III.3.5, the scheduling and reduction policy for GPUs may not be suitable for multicore CPUs in general.

One exceptional case is that OpenCL (Intel) is faster than OMP4-All for `lud` due to the kernel execution time. The kernels of `lud` are optimized very well when they are ported to OpenCL. Overall, OpenCL (Intel) is 230% slower than OMP4-All on average.

Comparison with OMP (gcc). For seven applications (`streamcluster`, `backprop`, `hotspot`, `bfs`, `srad`, `pathfinder`, and `nw`), OMP (gcc) is much faster than OMP4-All. The reason is that the OpenCL runtime overhead (*e.g.*, searching platforms and devices, scheduling and issuing commands, etc.) takes a large portion of the execution time. This happens when the total execution time is small or the execution time of each kernel is small. OpenMP does not have this kind of runtime overhead.

For seven applications (`lavaMD`, `myocyte`, `nn`, `b+tree`,

particlefilter, cfd, and lud), OMP4-All is faster than OMP (gcc). Since our OpenMP 4.0 programs are ported based on the original OpenMP programs, the performance difference only comes from the execution time of kernels. Since nn is significantly modified due to file read operations in the for loop as described before. This difference causes a large performance difference between OMP4-All and OMP (gcc). For other applications, they have different execution for each iteration. It means that the execution time of each iteration can be different. While OpenMP statically distributes iterations by default, OpenCL dynamically distributes iterations by default because scheduling occurs in a unit of a work-group. If a work-item of a kernel has different execution time, OpenCL performs better due to the dynamic work-group scheduling. As a result, OMP4-All is faster than OMP (gcc) due to load imbalance of the applications. Overall, OMP (gcc) is 21% slower than OMP4-All on average.

Comparison with OpenACC (PGI). The PGI compiler supports OpenACC for multicore CPUs since version 15.10 (released in November 2015). However, the PGI compiler fails to execute 13 applications. It fails to compile myocyte. For five applications (nn, streamcluster, kmeans, pathfinder, and jacobi), the generated binaries produce incorrect results. For seven applications (b+tree, backprop, hotspot, particlefiler, srad, nw, and lud), the binaries built by the PGI compiler generate runtime errors such as segmentation faults and aborts.

Only four applications (lavaMD, bfs, heartwall, and pathfinder) are compiled and executed successfully. OpenACC (PGI) is faster than OMP4-All for bfs and pathfinder because the inherent overhead of the OpenCL runtime as described in the comparison with OMP (gcc). Our translation framework generates OpenCL programs while the PGI compiler generates an executable binary for the multicore CPU. OMP4-All is faster than OpenACC (PGI) for lavaMD and heartwall. OMP4-All and OpenACC (PGI) perform the same computation because equivalent directives are inserted. Hence, the performance difference comes from the computation. This indicates that if the kernel execution time is long enough to amortize the inherent OpenCL runtime overhead, OMP4-All performs better than OpenACC (PGI). Overall, OpenACC (PGI) is 82% faster than OMP4-All for the applications that executed correctly on average.

III.4.3.4 Summary

If only PPE is applied, the performance is improved not much. However, DTM is applied together, the performance is improved up to 35x (jacobi) for the AMD GPU, up to 97x (lud) for the NVIDIA GPU, and up to 103x (nw) for the Intel CPU.

To see the effectiveness of our translator, we compare the original OpenCL and OpenMP versions. On average, the original OpenCL programs are 88%, 17%, and 230% slower than the target OpenCL

programs on the AMD GPU, the NVIDIA GPU, and the Intel CPU, respectively. The original OpenMP programs are 21% slower than the target OpenCL programs on the Intel CPU on average.

Except for the Intel multicore CPU, programs generated by our translation framework perform much better or as good as programs built by the PGI compiler. For the multicore CPUs, some programs generated by our translation framework perform worse because of the inherent overhead of the OpenCL runtime. Unfortunately, the PGI compiler fails to generate binaries that work correctly for some applications. Applications that execute successfully with the PGI compiler are 13/17 for the AMD GPU, 16/17 for the NVIDIA GPU, and 4/17 for the Intel multicore CPU. On the other hand, our translation framework successfully translates all the programs for all three devices.

Chapter IV

Support for a Heterogeneous Cluster

IV.1 Problems of Previous Approaches

Since the centralized host node schedules and delivers OpenCL commands to compute nodes, it would be a problem when the execution time of each command is smaller than the command scheduling and delivery (communication) overhead. Even though the target compute nodes have enough bandwidth to execute many commands, the host node is not able to deliver an enough number of commands to them due to the scheduling and delivery overhead. This problem gets more serious as the number of nodes increases.

For example, assume that there are 128 nodes (N001 - N128) in the cluster, and each node has a compute device (a total of 128 devices, D001 - D128). In addition, assume that one command-queue is attached to each compute device. Now, the OpenCL host program enqueues a command for each device, a total of 128 commands (C001 - C128), then it again enqueues a command for each device, another total of 128 commands (C129 - C256). In the end, each device has two commands to execute. The host node is in charge of scheduling and delivering all the commands. The host node schedules C001 for D001 and delivers it to D001. Then, the host node schedules C002 and delivers it to D002. This process continues until C128 is scheduled and delivered to D128. Then, C129 is scheduled and delivered to D001. Even though D001 finishes C001 early, it has to wait and to be idle until it receives C129 from the host node.

IV.2 The Approach of SnuCL-D

As shown in Figure I.3, SnuCL-D is laid between the cluster and an OpenCL application. SnuCL-D provides an OpenCL application with an illusion that all compute devices in the cluster are located in a single node. The application can use these devices as if they were located locally. This is the same as what the previous approaches did.

The major difference between previous approaches and SnuCL-D comes from the existence of the host node. In the previous approaches,

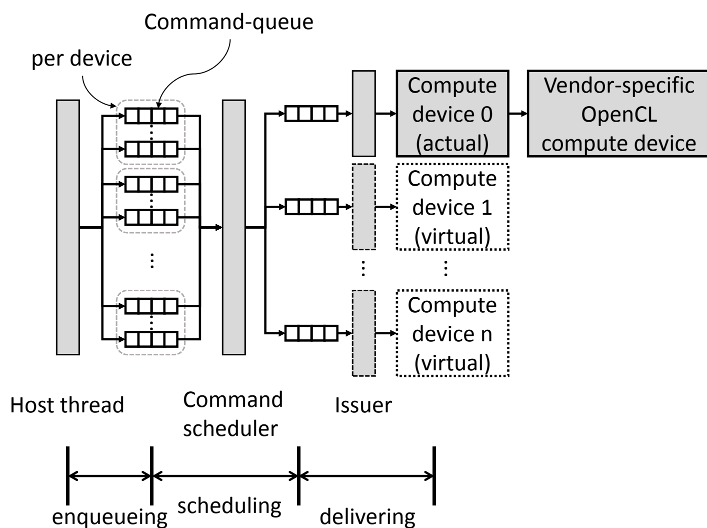


Figure IV.1: The organization of the SnuCL-D runtime.

the host node executes the host program. It also schedules commands and delivers them to appropriate target nodes. On the other hand, SnuCL-D does not have any designated host node. A copy of the host program in a single OpenCL application is executed in every node in the cluster. In addition, a SnuCL-D runtime instance shown in Figure IV.1 is executed in every node. When a command is enqueued to a command-queue by the host program instance running in each node, the command is scheduled by the command scheduler of the SnuCL-D runtime instance of the same node.

Each node in the cluster may have multiple vendor-specific OpenCL platforms for different accelerators. The SnuCL-D runtime instance in the node selects and controls the OpenCL platforms using the OpenCL installable client driver (ICD) mechanism[32]. When a

command is scheduled for the devices in the node, it is sent to the issuer thread of the device through a non-blocking producer-consumer queue. The issuer then forwards the command to an appropriate vendor-specific OpenCL platform implementation in the node.

IV.2.1 Overhead Analysis

In Figure IV.1, the execution path of a command from the host thread to a compute device is divided into three parts. Then, three different overheads are defined along the path: enqueueing, scheduling, and delivering overheads.

The *delivering overhead* occurs when a command is delivered to the target compute device. Since the host in centralized approaches must deliver all commands to appropriate devices in compute nodes, this overhead cannot be avoided in centralized approaches. However, in SnuCL-D, each node delivers commands only to its actual devices. Since the delivery occurs locally in a node in SnuCL-D, the delivering overhead is significantly alleviated compared to centralized approaches.

The *scheduling overhead* is caused by the command scheduler. The command scheduler determines the execution order of commands and makes sure that memory objects in the compute devices are updated. This scheduling overhead can be significantly alleviated using a new API function **clAtttachBufferToDevice()**. This function is

described later in Section IV.5.

The *enqueueing overhead* occurs when a command is enqueued by the OpenCL runtime. Since a command must be enqueued to a command queue in OpenCL, we cannot avoid this overhead in centralized approaches. However, in SnuCL-D, if it is known that a command will be discarded eventually because the target device is virtual, the command does not need to be enqueued. The enqueueing overhead is alleviated by a new API function and the queueing optimization technique. The queueing optimization is described later in Section IV.6.

IV.2.2 Remote Device Virtualization

The SnuCL-D runtime instance in each node makes a remote device (*i.e.*, the device residing in another node) visible to the node. This mechanism is called as *remote device virtualization* (RDV). When the node sees a compute device installed in another node through SnuCL-D, the device is called as a *virtual device* to the node. If a compute device is installed in the node itself, it is called an *actual device*.

To implement RDV, the SnuCL-D runtime instance in each node exchanges its device information with SnuCL-D runtime instances in other nodes when the host program invokes an OpenCL API function for the first time in the application. Then, each SnuCL-D runtime instance assigns a unique device ID to each device across the cluster and creates virtual devices in its node.

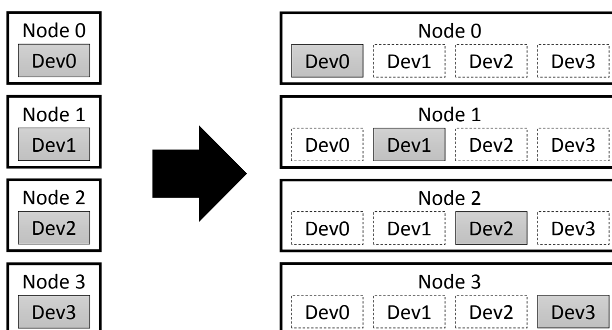


Figure IV.2: Remote device virtualization.

The SnuCL-D runtime instance in each node schedules commands asynchronously. If there is a command for an actual device in the node, it schedules and executes the command without any synchronization with other nodes at all. For example, there are four nodes in the cluster shown in Figure IV.2. Each node has an actual compute device. Because of RDV, each node appears to have four compute devices including three virtual devices. Node 0 has an actual device Dev0 and three virtual devices Dev1, Dev2, and Dev3. Node 1 also has an actual device Dev1 and three virtual devices Dev0, Dev2, and Dev3. Since the same host program is executed in each node, commands enqueued by the host program for Dev0 are executed in Node 0, for Dev1 in Node 1, for Dev2 in Node 2, and for Dev3 in Node 3.

IV.2.3 Redundant Computation and Data Replication

Since the OpenCL host program is executed in every node with SnuCL-D, the data produced by the host program is replicated in every node.

This makes SnuCL-D faster than the centralized approaches because it eliminates most of the extra data transfer overhead of the centralized approaches between the host and compute nodes.

Previous studies[37, 43, 61] have exploited redundant computation to reduce data sharing. They showed that the communication overhead could be reduced by redundant computation. There are some other studies[12, 72] that have exploited data replication to increase data locality. They showed that the read latency can be reduced by duplicating data locally.

In the centralized approaches, the host node needs to send data to a compute device in a compute node through the interconnection network in the cluster, and this typically happens to all compute devices simultaneously. Thus, as the number of nodes grows, the cost of data transfer increases.

On the other hand, in SnuCL-D, every node locally computes necessary data. The node does not need any communication, and just sends data to its local compute devices. Performing redundant computation with data replication, SnuCL-D alleviates the delivering overhead significantly.

This can also be easily applied to the file system. Since the identical host program is executed on each node, file-read operations can be executed simultaneously without any synchronization. In this case, the data is replicated across all nodes. All nodes in the cluster typi-

cally share a file system so that every node reads the same data from the same file. For file-write operations, only a designated root node (*e.g.*, whose MPI rank is zero) executes the file-write operation, while other nodes do nothing. If there is any synchronization required due to redundant computation, SnuCL-D exploits wrapper functions. For example, if there is an `srand()` call to initialize a seed value, a wrapper for `srand()` is implemented so that makes every node have the same seed value.

IV.2.4 Memory-read Commands

Memory-read commands (*e.g.*, **clEnqueueReadBuffer()**) copy data from a device to the host memory. For memory consistency, the host memory of each node must be kept up-to-date to run the host program correctly. Therefore, after a memory-read command is executed on an actual device, the SnuCL-D runtime instance propagates the data to other nodes. When a memory-read command for a virtual device is scheduled by the command scheduler in a node, the runtime instance in the node receives the data from the node that owns the corresponding actual device. This may cause performance degradation. However, frequent memory-read commands are not encouraged because it degrades performance significantly in heterogeneous computing.

IV.3 Consistency Management

OpenCL allows multiple commands to access the same memory object simultaneously. This may cause a consistency problem. To solve this problem, a multiple-writers protocol[41] may be employed. This was used in traditional software shared virtual memory (SVM) systems[41]. However, this incurs a significant consistency management overhead because of twins (copies of the original memory object) and identifying differences between the twin and the modified memory object.

Instead, SnuCL-D tries to avoid the situation by following essentially the same consistency management mechanism as that of SnuCL. That is, SnuCL-D executes commands one by one. However, the distributed consistency management scheme in SnuCL-D has a big advantage over SnuCL.

When the command scheduler schedules a command, SnuCL-D checks if there is a conflict between the command being scheduled and any commands being executed. A conflict occurs if two commands access the same memory object, and at least one writes to it. If such a case is detected, the command scheduler synchronizes them with the event synchronization mechanism in OpenCL, *i.e.*, it serializes them.

In addition, buffers (memory objects) accessed by a command should have consistent up-to-date data before executing the command.

Table IV.1: Actions of a special command for consistency management in SnuCL-D.

	B is actual (source)	**B is virtual (source)**
A is actual (destination)	Copy data from B to A	Receive data from the node of B
A is virtual (destination)	Sends data to node of A	Do nothing

However, a memory object is not associated with a compute device in OpenCL. Thus, when a command accessing a memory object is executed, it is required for the runtime to find the device that has the latest copy of the memory object. To facilitate this, SnuCL-D maintains the list of devices that have the latest copy of each memory object. This list is called the *latest device list* for a memory object. The latest device list can contain either actual or virtual devices.

When the command scheduler schedules a command and the corresponding device (Device A) does not have up-to-date data, SnuCL-D inserts a special command to bring up-to-date data from the latest device (Device B). Actions of the special command are described in Table IV.1. If both A and B are actual, it means A and B are in the same node. Thus, data is copied from B to A. If A is actual and B is virtual, it means A is in the node while B is in another node. The node receives data from the node where B resides and updates the data to A. If A is virtual and B is actual, it means A is in another node while B is in the node. The node sends data to the node where A resides. If both A and B are virtual, it means A and B are in other nodes, and therefore nothing is done.

SnuCL performs this process only in the host node while SnuCL-D performs it in every node. The host node of SnuCL delivers a message to both the source device and the destination device. Then, the source device sends data to the destination device. As a result, the host node of SnuCL may be a significant performance bottleneck as the size of the cluster increases. Since SnuCL-D performs consistency management in every node, the source and the destination begin communication without receiving a message from the centralized host. This is a big advantage of SnuCL-D over SnuCL.

The total amount of overhead for consistency management in SnuCL-D seems to be bigger than that of SnuCL because every node performs consistency management in SnuCL-D. However, the wall-clock time spent on consistency management of SnuCL-D is smaller than that of SnuCL because the consistency management overhead is spread over all the nodes in the cluster, and there is no communication between the centralized host and the compute nodes. This is justified by using a microbenchmark in Section IV.7.2.

IV.4 Deterministic Command Scheduling

Another major difference in consistency management between SnuCL and SnuCL-D is the deterministic command scheduling mechanism in SnuCL-D. Since every node executes the same host program in SnuCL-D, the scheduling order of commands accessing the same memory

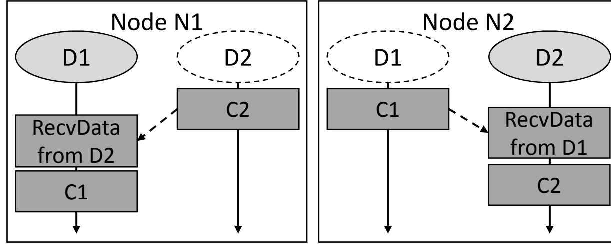


Figure IV.3: Commands C1 and C2 modify the same memory object.

object is the same across all the nodes.

Consider the scenario illustrated in Figure IV.3. There are two nodes N1 and N2 with devices D1 and D2, respectively. D1 is actual and D2 is virtual in N1 while D1 is virtual and D2 is actual in N2. There are two commands C1 and C2 modifying the same memory object M. C1 is enqueued to a command-queue attached to D1 and C2 is enqueued to a command-queue attached to D2. Since there is no synchronization between C1 and C2, they can be scheduled in any order.

Assume that C2 is scheduled first in N1. The scheduler in N1 records D2 in the latest device list of M. Since C2 is a command for a virtual device D2, it is discarded after scheduling by the SnuCL-D runtime. Then, C1 is scheduled in N1. Since the latest device of M is D2, the scheduler inserts a special command that receives data from D2.

On the other hand, assume that C1 is scheduled first in N2. D1 is inserted in the latest device list of M. Since C1 is a command for

a virtual device D1 in N2, it is discarded after scheduling. Then, C2 is scheduled in N2. Since the latest device of M is D1, the scheduler inserts a special command that receives data from D1. In this scenario, N1 tries to receive data from N2 while N2 tries to receive data from N1. Thus, deadlock occurs and there is no progress at all.

To solve this problem, SnuCL-D enforces the command scheduling order to be the same as the command enqueueing order specified by the host program, *i.e.,* the order of **clEnqueue**$\cdots$ host API calls appeared in the host program. For example, if C1 is enqueued first and then C2 is enqueued in the host program, C1 should be scheduled first.

If the host program is single-threaded, **clEnqueue**$\cdots$ API calls are executed sequentially. Thus, SnuCL-D enforces the same order of execution across all the nodes in the cluster. However, it is possible that **clEnqueue**$\cdots$ is called by two or more host threads in the same node because OpenCL API calls are thread-safe except **clSetKernelArg()**[32]. In this case, the enqueueing order is not deterministic across nodes. Since single-threaded OpenCL host programs are much more common, solving this multiple-host-threads problem is left as the future work.

IV.5 New API Function: clAttachBufferToDevice()

To eliminate the scheduling and consistency management overhead, we propose a new OpenCL host API function:

```
void clAttachBufferToDevice(cl_mem m, cl_device_id d);
```

This function can be inserted by a programmer to eliminate the scheduling overhead. If this API function is called, the runtime assumes that the compute device `d` always has the latest copy of the memory object `m`. As mentioned before, when a command that accesses an OpenCL memory object is scheduled, it is required for the runtime to find the device that has the latest copy of the memory object. If this function is called for a memory object by the host program, the runtime does not need to maintain the latest device list for the memory object. The runtime always selects `d` as the device that has the latest copy of the memory object. In many high-performance OpenCL applications using multiple devices, a memory object is typically modified by a single device. Such a memory object can be attached to the device to alleviate the scheduling overhead.

IV.6 Queueing Optimization

When the command scheduler detects that a command is for a virtual device, it does not deliver the command to the issuer of the device. However, the scheduler still needs to schedule these commands because it has to maintain the latest device list for each memory object.

For example, assume that a virtual device D1 modifies memory object M because of command C1. Then actual device D2 accesses the same memory object M because of command C2. Even though C1 is for a virtual device, it should be scheduled by the runtime because the latest device list of M should be updated to include D1.

However, since the latest device list is no longer required for the memory object declared by **clAttachBufferToDevice()**, commands that access the memory object attached to a virtual device do not need to be scheduled by the runtime. In addition, some commands for virtual devices do not need to even be enqueued by the runtime.

For example, assume that **clEnqueueNDRangeKernel()** is called once for every actual and virtual device. If there are 128 devices and only one of them is an actual device, 127 commands enqueued are discarded by the command scheduler. If each memory object accessed by those 127 commands is attached to a device (without regard to the device being actual or virtual) by **clAttachBufferToDevice()**, they do not need to even be enqueued either.

Table IV.2: Conditions under which commands do not need to be enqueued.

API	Conditions
clEnqueueNDRangeKernel()	1. Each memory object is attached to a device 2. The command-queue is for a virtual device
clEnqueueCopyBuffer() clEnqueueWriteBuffer()	1. Each memory object is attached to a virtual device 2. The command-queue is for a virtual device
clEnqueueReadBuffer()	Always enqueued to update the host memory in each node

Table IV.2 describes conditions under which a command does not need to be enqueued. If these conditions are met for a command, it is discarded rather than enqueued. The command enqueued by a **clEnqueueReadBuffer()** call needs to always be enqueued by the runtime and processed by the scheduler even if the associated device is virtual because it has to update the host memory of each node in the cluster.

IV.7 Performance Evaluation

This section describes experimental results of SnuCL-D that transparently extends OpenCL for heterogeneous clusters.

IV.7.1 Evaluation Methodology

The SnuCL-D implementation is based on SnuCL[42], our previous open-source OpenCL framework for a heterogeneous cluster. SnuCL-D is evaluated using a large-scale CPU cluster and a medium-scale GPU cluster. The configurations of the clusters are summarized in

Table IV.3: System configuration for the large-scale CPU cluster.

Number of nodes	512
Processor	2×Intel 2.93 Ghz quad-core Xeon x5570 for each node
Memory	24GB for each node
OS	Red Hat Enterprise Linux 5.3
Interconnect	Mellanox Infiniband QDR
OpenCL	AMD APP SDK v2.9
MPI	Open MPI 1.6.3
C compiler	GCC 4.4.6
Fortran compiler	GNU Fortran 3.4.6

Table IV.4: System configuration for the medium-scale GPU cluster.

Number of nodes	36
Processor	2×Intel 2.0 Ghz octa-core Xeon E5-2650 for each node
GPU	4×AMD Radeon HD 7970 (2GB each) for each node
Memory	128GB for each node
OS	Red Hat Enterprise Linux 6.3
Interconnect	Mellanox Infiniband QDR
OpenCL	AMD APP SDK v2.8
MPI	Open MPI 1.6.4
C compiler	GCC 4.4.6

Table IV.3 and Table IV.4. The focus of the evaluation is the scalability of OpenCL programs on large-scale heterogeneous clusters. Since it is not possible to have access to a large-scale heterogeneous cluster, the large-scale CPU-only homogeneous cluster is used to evaluate the scalability of SnuCL-D. To see the scalability of the heterogeneous cluster, a 36-node GPU cluster is used.

Table IV.5: Applications used.

Application	Source	Input	
		The CPU cluster	**The GPU cluster**
blackscholes	PARSEC	64M options	128M options
BinomialOption	AMD SDK	1M samples	1M samples
CP	Parboil	16K×16K	16K×16K
N-body	NVIDIA	2.5M bodies	10M bodies
MatrixMul	NVIDIA	16K×16K	10752×10752
EP	NPB	class E	class E
FT	NPB	class D	class C
CG	NPB	class E	class C
MG	NPB	class E	class C
SP	NPB	class E	class D
BT	NPB	class E	class D

First, SnuCL-D is compared to SnuCL using a microbenchmark. Then, the performance of SnuCL-D is compared to that of SnuCL and MPI. The reason of selecting SnuCL as a representative centralized approach is that it is the only centralized framework that has been evaluated with a large-scale cluster. Also, the effectiveness of the proposed techniques is presented.

The applications are from the SNU NPB suite[65], PARSEC[23], NVIDIA SDK[51], AMD[14], and Parboil[68] for the evaluation. Table IV.5 summarizes the applications used. The SNU NPB suite[65] is an OpenCL implementation of the NAS Parallel Benchmarks (NPB) suite[50]. It provides OpenCL NPB applications for multiple compute devices. Since `blackscholes` is a multi-threaded C program, it is manually translated to an OpenCL application for multiple devices. As `BinomialOption`, `CP`, `N-body`, and `MatrixMul` are OpenCL programs

for a single device, they are modified to distribute workload across multiple compute devices.

The original NPB applications are written in Fortran using MPI, and some of them require the number of MPI processes to be a square number (the square of an integer). Similarly, the corresponding SNU NPB applications also require the number of OpenCL compute devices to be a square number. The AMD APP SDK v2.9 on the CPU cluster configures all the CPU cores in a node as a single CPU compute device. Thus, the number of the compute devices may not be a square number. In this case, the SnuCL-D runtime divides a CPU device into two sub-devices using standard OpenCL API function **clCreateSubDevices()** to make the number of devices in the entire CPU cluster a square number.

There are two ways to measure the scalability of a parallel system: *strong scalability* and *weak scalability*. For strong scalability, the execution time is measured with different numbers of processors while the problem size is fixed. It is useful to see the overhead of the underlying runtime system on each processor. The overhead takes more portions of the execution time as the number of processors increases. It is because the total execution time decreases as the number of processors increases, but the execution time of the parallel overhead increases. For weak scalability, the execution time is measured with different numbers of processors while the problem size per processor is constant. It is useful to see the communication overhead between

processors. Since the execution time of the problem per processor remains constant, the communication overhead dominates weak scalability. Since what we want to see is the runtime overhead including communication overhead, the strong scalability of MPI, SnuCL, and SnuCL-D is measured.

The execution time of an NPB application in MPI or OpenCL is the execution time reported by default by the application. The data initialization and MPI initialization/finalization time is excluded in this case. For other OpenCL applications, the measured execution time is the wall clock time of their entire execution except for the MPI initialization/finalization time because SnuCL and SnuCL-D use MPI internally and the MPI initialization/finalization time has a large variation across different runs. The MPI initialization/finalization requires MPI processes to exchange data with all other MPI processes. These times vary significantly from one run to another. The source of the variation is MPI. Thus, they are excluded in the experiment to focus on the runtime behavior. The data initialization time is included, though.

IV.7.2 Evaluation with a Microbenchmark

As mentioned before, the major bottleneck of the centralized approach is the centralized host. Specifically, it is the case that the time taken by the host node to schedule a command is longer

than the execution time of the command in each compute node. In SnuCL, the host node always delivers commands to the target compute devices. For example, when executing a buffer-write command (**clEnqueueWriteBuffer()**), the host always delivers a message that contains information about the command and data to the target compute node. Similarly, when executing a buffer-copy command (**clEnqueueCopyBuffer()**), the host node always delivers a message to the source and destination nodes. Then, the two nodes communicate with each other and exchange data.

On the other hand, since SnuCL-D executes the host program in every node, there is no need to deliver such a message. For example, when executing a buffer-write command, only the node that owns the device performs the command, and no communication is required. When executing a buffer-copy command, the source and destination nodes know each other because the runtime maintains the same latest device list. Therefore, they just exchange data.

To show the effectiveness of the decentralized approach over a centralized approach (*e.g.*, SnuCL), a microbenchmark is executed. In the host program, each device copies the contents of its buffer to the buffers of all other devices using **clEnqueueCopyBuffer()**. The buffer size is 16 bytes to emphasize the delivering overhead. The total number of calls to **clEnqueueCopyBuffer()** in each iteration is the square of the number of devices. The host program repeats this process 100 times.

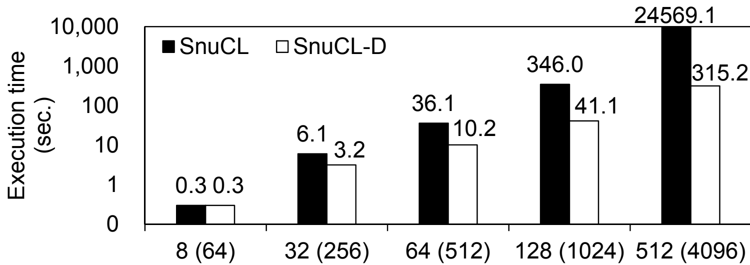


Figure IV.4: Comparison of SnuCL and SnuCL-D using a microbenchmark. Numbers in the parentheses represent numbers of CPU cores.

Figure IV.4 shows the evaluation results. The x-axis shows the number of nodes (numbers in the parentheses represent numbers of CPU cores), and the y-axis in logarithmic scale shows the execution time in seconds. When the number of nodes is small, the performance of SnuCL-D is similar to that of SnuCL. However, for more than 32 nodes (256 cores), SnuCL-D significantly outperforms SnuCL. SnuCL-D is 78 times faster than SnuCL for 512 nodes (4096 cores).

IV.7.3 Evaluation on the Large-scale CPU Cluster

Figure IV.5 shows the performance comparison between MPI, SnuCL, and SnuCL-D on the large-scale CPU cluster. The x-axis shows the number of nodes, and the y-axis shows speedup over 256 MPI processes running on 32 nodes (*i.e.*, 256 CPU cores) in logarithmic scale. Since E-class NPB applications do not run with less than 32 nodes due to the memory size, the applications are executed on 32, 128, and 512 nodes. An exception is FT. Since E-class FT requires a memory size that is bigger than the total amount of memory of 32 nodes, the

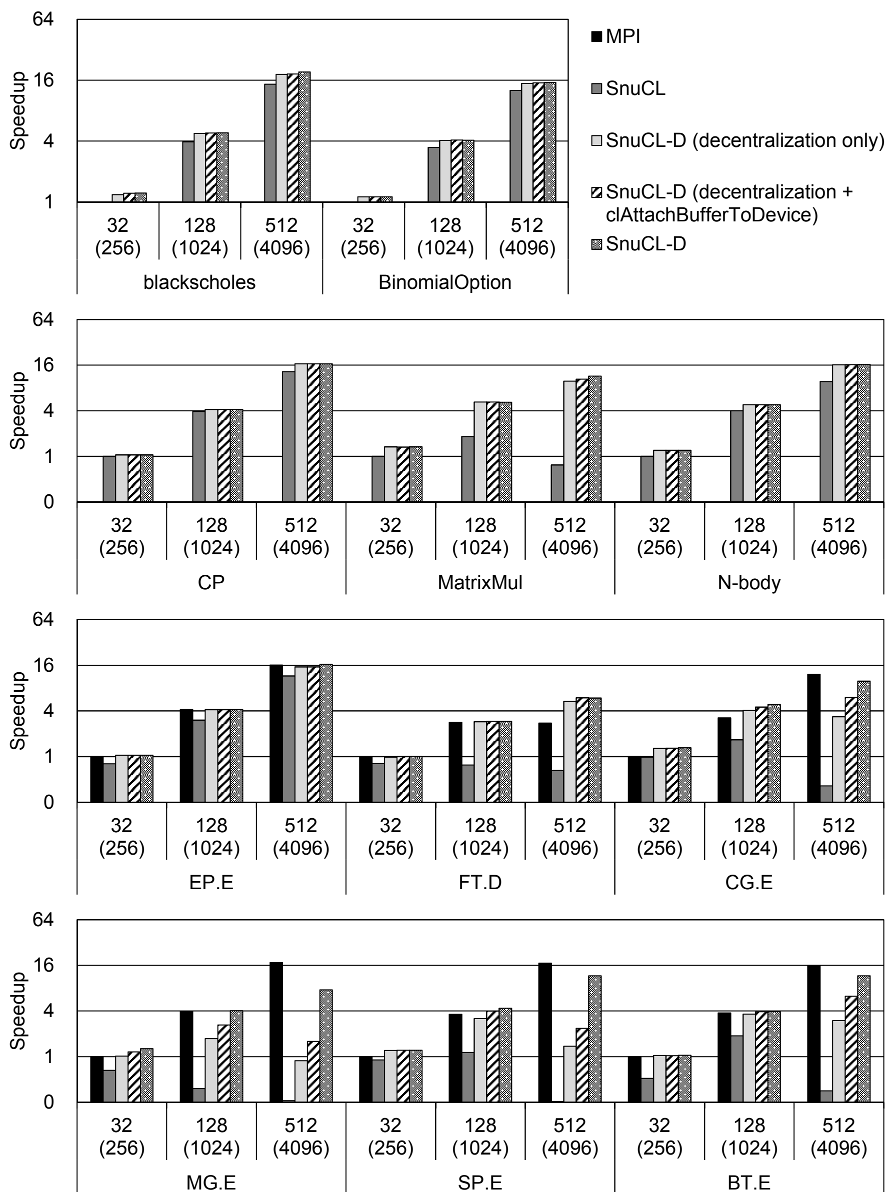


Figure IV.5: Comparison between MPI, SnuCL, and SnuCL-D on the large-scale CPU cluster (speedup over 256 MPI processes on 32 nodes with 256 CPU cores). Numbers in the parentheses represent numbers of CPU cores.

D-class input for FT is used.

The bars labeled MPI and SnuCL show the performance of MPI and SnuCL, respectively. Since `blackscholes`, `BinomialOption`, `CP`, `N-body`, and `MatrixMul` do not have an MPI version, the speedup is obtained based on SnuCL on 32 nodes (*i.e.*, 256 CPU cores).

The bar labeled SnuCL-D (decentralization only) indicates the speedup of the SnuCL-D version only with the decentralization technique with RDV. The runtime enqueues commands for virtual devices in addition to actual devices, and **clAttachBufferToDevice()** is not used in the applications. Thus, the scheduling overhead and the enqueueing overhead still remain in this version. The bar labeled SnuCL-D (decentralization + clAttachBufferToDevice) stands for the SnuCL-D version with the decentralization technique, and the applications use **clAttachBufferToDevice()**. Thus, this version alleviates the scheduling overhead but not the enqueueing overhead.

Finally, SnuCL-D is the speedup of SnuCL-D that does not enqueue commands for virtual devices. It is a decentralized version, and **clAttachBufferToDevice()** is used in the applications. In addition, when a command is enqueued for a virtual device by the host, the command is discarded by the runtime to reduce the enqueueing overhead. Since this optimization only takes effect with the API function **clAttachBufferToDevice()**, **clAttachBufferToDevice()** should be used in the applications. SnuCL-D alleviates not only the scheduling overhead

Table IV.6: The number of commands executed on the large-scale CPU cluster.

# of nodes (# of cores)	BlackScholes	BinomialOption	CP	MatrixMul	N-body	EP.E
32 (256)	6,848	12,864	2,688	256	192	320
128 (1024)	27,392	51,456	10,752	1,024	768	1,280
512 (4096)	109,568	205,824	43,008	4,096	3,072	5,120

# of nodes (# of cores)	FT.D	CG.E	MG.E	SP.E	BT.E
32 (256)	17,728	2,353,153	2,001,680	4,623,168	1,625,856
128 (1024)	70,912	9,412,609	7,519,568	36,977,920	12,672,000
512 (4096)	283,648	37,650,433	28,381,808	295,793,664	100,036,608

but also the enqueueing overhead.

IV.7.3.1 Applications Executing a Small Number of Commands

As shown in Table IV.6, `blackscholes`, `BinomialOption`, `CP`, `N-body`, `MatrixMul`, `EP`, and `FT` execute a relatively small number of commands. Thus, SnuCL-D (decentralization + clAttachBufferToDevice) and SnuCL-D do not improve performance significantly compared to SnuCL-D (decentralization only). However, the decentralized versions improve performance compared to the centralized version (SnuCL). The performance gap becomes bigger as the number of nodes becomes larger.

The execution time of `blackscholes`, `BinomialOption`, and `CP` is dominated by the execution time of their kernels. The performance gap between SnuCL and SnuCL-D is due to the delivery overhead of SnuCL. Since the host node always delivers commands to compute de-

vices in SnuCL, this cost increases as the number of nodes increases. On the other hand, SnuCL-D does not have any communication overhead for delivering commands. Thus, SnuCL-D scales better than SnuCL.

`N-body` shows noticeable performance degradation with SnuCL for 512 nodes compared to SnuCL-D. `N-body` manipulates two arrays `Pos` and `Vel`. The array `Vel` is divided and distributed to compute devices while `Pos` is replicated. For example, if there are 1024 devices and the input is 2.5M bodies, the size of `Vel` for each device is 40KB (1.5M×4 elements×4B(float)/1024). However, the size of `Pos` for each device is 40MB (1.5M×4 elements×4B(float)). The size of `Pos` remains the same even if the number of devices is changed. Hence, the total amount of data to be transferred increases as the number of devices increases (40MB for `Vel` and 40MB×(the number of devices) for `Pos`). Since SnuCL needs to transfer the data from the host to compute nodes, its performance is slightly degraded. On the other hand, SnuCL-D does not need to transfer data because every node has the data. Thus, SnuCL-D scales better than SnuCL.

The gap for `MatrixMul` is dramatic for 128 and 512 nodes. `MatrixMul` is an application that multiplies two matrices. It initializes large 16384×16384 float-type matrices, and the size of a matrix is 1GB (16K×16K×4B). To perform computation on a device, each device needs all elements of a matrix and some rows of the other matrix. For example, if there are 16 nodes and each node has a device,

the host node should transfer 1.06GB (1GB+1GB/16) to each compute node in SnuCL. If there are 64 nodes, the host should transfer 1.02GB (1GB+1GB/64) to each compute node. Thus, the host should transfer at least 1GB to each compute node even though the number of compute nodes is large. As a result, SnuCL does not scale well for 512 nodes. On the other hand, SnuCL-D does not require any data communication between nodes and scales well because every node initializes the matrices in SnuCL-D.

`EP` is an embarrassingly parallel application. Since a large portion of `EP`'s execution time is spent on kernel execution, the reason of the performance gap between SnuCL and SnuCL-D is similar to the case of `blackscholes`. The performance of SnuCL-D is comparable to that of MPI in this case.

API functions **clEnqueueNDRangeKernel()** and **clEnqueueAlltoAllBuffer()** consume most of the execution time of `FT`. The performance of SnuCL-D is comparable to that of MPI up to 128 nodes. However, MPI performs significantly worse than SnuCL-D for 512 nodes. The grid size of the D-class FT is 2048×1024×1024 (X×Y×Z), and **MPI_Alltoall()** communication is performed based on the z-axis. There are 4096 MPI processes for 512 nodes, which is bigger than the number of elements in the z-axis. Thus, MPI performs another level of **MPI_Alltoall()** communication along the y-axis in addition to the communication along the z-axis. On the contrary, since there are 512 compute devices in 512 nodes and this

makes 1024 sub-devices for SnuCL-D, which is equal to the number of elements in the z-axis, SnuCL-D performs more efficient one-level **clEnqueueAlltoAllBuffer()** along the z-axis. This is the reason why SnuCL-D outperforms MPI for 512 nodes.

SnuCL does not scale at all for FT and SnuCL-D is much faster than SnuCL when the number of nodes is large. The performance of SnuCL decreases as the number of nodes increases. Since **clEnqueueAlltoAllBuffer()** commands make the host deliver a point-to-point communication message to each compute node one by one, the amount of communication overhead between the centralized host and compute nodes becomes much more severe as the number of nodes increases.

IV.7.3.2 Applications Executing a Large Number of Commands

Unlike the applications executing a small number of commands, SnuCL-D (decentralization + clAttachBufferToDevice) and SnuCL-D significantly improve performance compared to SnuCL-D (decentralization only) for CG, MG, SP, and BT. In addition, the performance of the decentralized versions is much better than SnuCL for these applications.

As shown in Table IV.6, CG, MG, SP, and BT have a large number of commands executed. The decentralized versions alleviate the delivering overhead incurred by SnuCL. As expected, using **clAttach-**

Table IV.7: The number of commands for actual devices after queueing optimization (Numbers in the parentheses represent numbers of commands for all devices. Since there are two devices in a node, the number of commands for actual devices is about 2/64 (3.1%) for 32 nodes, 2/256 (0.8%) for 128 nodes, and 2/1024 (0.2%) for 512 nodes.)

# of nodes (# of cores)	BlackScholes	BinomialOption	CP	MatrixMul	N-body	EP.E
32 (256)	214 (6,848)	402 (12,864)	84 (2,688)	8 (256)	6 (192)	10 (320)
128 (1024)	214 (27,392)	402 (51,456)	84 (10,752)	8 (1,024)	6 (768)	10 (1,280)
512 (4096)	214 (109,568)	402 (205,824)	84 (43,008)	8 (4,096)	6 (3,072)	10 (5,120)

# of nodes (# of cores)	FT.D	CG.E	MG.E	SP.E	BT.E
32 (256)	554 (17,728)	73,537 (2,353,153)	62,272 (2,001,680)	144,474 (4,623,168)	50,808 (1,625,856)
128 (1024)	554 (70,912)	73,537 (9,412,609)	57,172 (7,519,568)	288,890 (36,977,920)	99,000 (12,672,000)
512 (4096)	554 (283,648)	73,537 (37,650,433)	52,123 (28,381,808)	577,722 (295,793,664)	195,384 (100,036,608)

BufferToDevice() (SnuCL-D (decentralization + clAttachBufferToDevice)) improves performance significantly because the scheduling overhead is alleviated.

Queueing optimization of commands for virtual devices (SnuCL-D) further improves the performance because it alleviates both the enqueueing overhead and the scheduling overhead. Table IV.7 shows the number of commands executed by actual devices for a node. The number of commands that do not need enqueueing increases as the number of nodes increases because the number of virtual devices increases. Therefore, it is expected that the performance improvement is more significant when the number of nodes is large.

However, the exception in Table IV.7 is `MG`. Its number of commands for actual devices decreases as the number of nodes increases. E-class `MG` performs eleven levels of computation in every iteration. For the last level, the grid size is 2048×2048×2048 (X×Y×Z). As a level is raised, the grid size becomes a half of the grid size at the level below in each dimension. If the grid size is smaller than the number of devices, `MG` does not perform computation beyond the level. Thus, as the number of nodes increases, the number of levels executed by `MG` decreases. This is the reason why the number of executed commands for actual devices decreases as the number of nodes increases for `MG`.

SnuCL-D performs slightly better than MPI for up to 128 nodes with `CG`, `MG`, `SP`, and `BT`. These applications execute a large number of **clEnqueueCopyBuffer()**. Since the number of MPI processes is four times larger than the number of OpenCL sub-devices in SnuCL-D, the amount of communication in MPI is four times larger than that in SnuCL-D. This is the reason why SnuCL-D outperforms MPI for up to 128 nodes. For 512 nodes, MPI performs better than SnuCL-D because the amount of work in the OpenCL kernel for 512 nodes is much smaller than that in the kernel for 32 or 128 nodes. In other words, the execution time of a kernel is not big enough to amortize the inherent overhead of the OpenCL runtime.

A command that is required to be executed locally in a node must be scheduled by the command scheduler of the same node. This overhead is inherent to the OpenCL runtime and may not be avoided. On

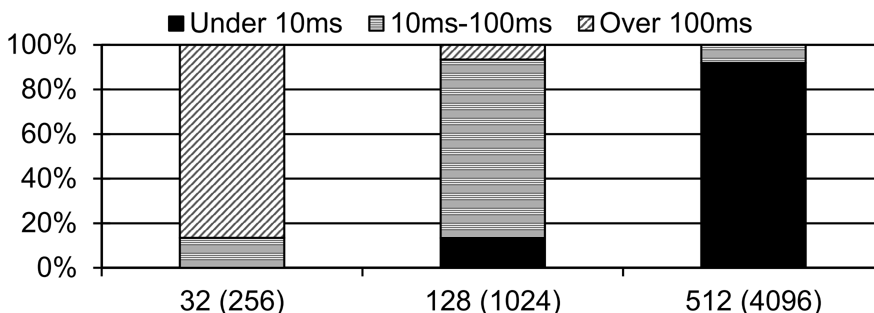


Figure IV.6: Distribution of the kernel execution time of `BT`.

the contrary, MPI just executes the code immediately. Since it is obvious that the amount of work in a kernel decreases as the number of nodes increases for the same input, the inherent runtime overhead in OpenCL takes a larger portion of the execution time for a large number of nodes. However, this overhead can be amortized if the input size is big enough. Since the largest input class allowed for the NPB applications is class E, it is not possible to increase the input size further. It is expected that the performance of SnuCL-D is comparable to MPI for the input sizes bigger than class E for 512 nodes.

On closer inspection, there is no kernel whose execution time is larger than 100 milli-seconds for `CG`, `MG`, `SP`, and `BT` for 512 nodes. For example, Figure IV.6 shows the distribution of the execution time of kernels in `BT`. For 32 nodes, kernels whose execution time is larger than 100 milli-seconds accounts for 86.5% of the total execution time of kernels. For 128 nodes, kernels whose execution time is larger than 10 milli-seconds accounts for 86.6% of the total execution time of kernels. For 512 nodes, kernels whose execution time is smaller than 10

Table IV.8: Speedup of SnuCL-D over SnuCL on the large-scale CPU cluster.

# of nodes (# of cores)	blackscholes	BinomialOption	CP	N-body	MatrixMul	EP.E
32 (256)	1.23	1.12	1.05	1.20	1.33	1.29
128 (1024)	1.23	1.18	1.06	1.20	2.84	1.37
512 (4096)	1.32	1.20	1.27	1.68	14.87	1.43

# of nodes (# of cores)	FT.D	CG.E	MG.E	SP.E	BT.E
32 (256)	1.24	1.32	1.93	1.34	2.00
128 (1024)	3.79	2.90	10.68	3.81	2.10
512 (4096)	9.03	24.15	28.93	45.31	32.85

milli-seconds accounts for 92% of the total execution time of kernels.

According to the evaluation result with a microbenchmark, the buffer-copy overhead of SnuCL is much bigger than that of SnuCL-D for a large number of nodes. Since `CG`, `MG`, `SP`, and `BT` execute many buffer-copy commands, they scale well with SnuCL-D.

IV.7.3.3 Speedup over SnuCL

Table IV.8 summarizes the speedup of SnuCL-D over SnuCL. Basically, the performance gap between SnuCL-D and SnuCL becomes bigger as the number of nodes increases. SnuCL-D is better than SnuCL for `blackscholes`, `BinomialOption`, `CP`, `N-body`, `MatrixMul`, `EP`, and `FT` because of the decentralization technique. For `CG`, `MG`, `SP`, and `BT`, the gap is much bigger because all the proposed techniques are effective. For 512 nodes, SnuCL-D is more than 20 times faster than SnuCL. Also, SnuCL-D is 45.31 times faster than SnuCL for `SP`.

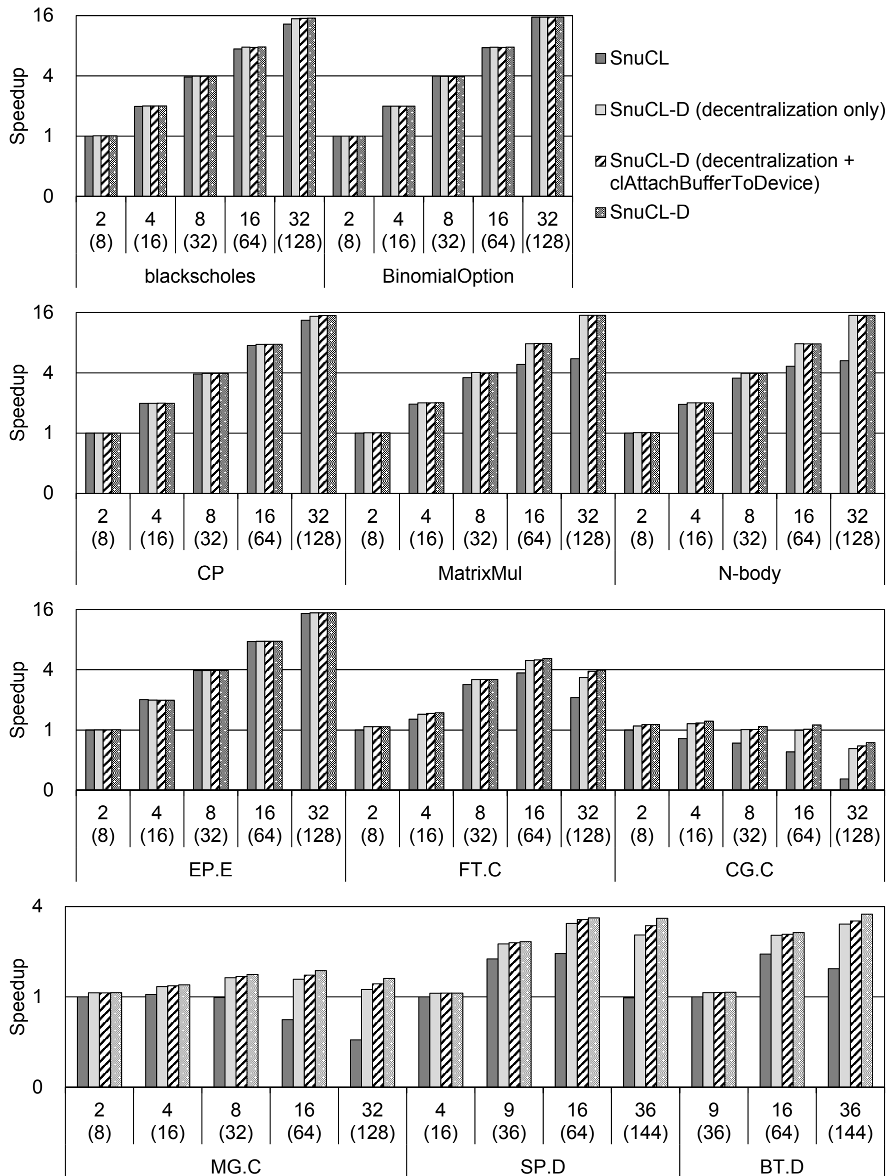


Figure IV.7: Comparison between SnuCL and SnuCL-D on the medium-scale heterogeneous cluster. Speedup is obtained over SnuCL with 8 GPU devices on 2 nodes, except SP and BT. Speedup of SP is obtained over SnuCL with 16 GPU devices on 4 nodes. Speedup of BT is obtained over SnuCL with 36 GPU devices on 9 nodes. Numbers in the parentheses represent numbers of GPU devices.

IV.7.4 Evaluation on the Medium-scale GPU Cluster

To see the scalability of SnuCL-D on a heterogeneous cluster, SnuCL and SnuCL-D are evaluated on the 36-node GPU cluster. Since it is not possible to have an access to a large-scale heterogeneous cluster, the scalability is measured on such a medium-scale GPU cluster.

Figure IV.7 shows the performance comparison of SnuCL and SnuCL-D. The x-axis shows the number of nodes and the y-axis shows the speedup of SnuCL-D over SnuCL with eight GPU devices in a logarithmic scale. It is better to show the speedup over SnuCL with a single GPU device, but a single GPU device cannot satisfy the memory requirements of some applications. For consistency, the speedup is measured based on SnuCL with eight GPU devices. Note that SP and BT require the number of devices to be a square number. The speedups of `SP` and `BT` are obtained over SnuCL with 16 GPU devices and 36 GPU devices, respectively.

Since `blackscholes`, `BinomialOptions`, and `CP` are applications that scale well, both SnuCL and SnuCL-D show good scalability. As described in Section IV.7.3.1, the total amount of transferred data in `N-body` and `MatrixMul` increases as the number of devices increases. Thus, SnuCL-D performs better than SnuCL because SnuCL-D does not need to transfer data. Note that the data delivery latency to a GPU device is bigger than that of a CPU device in SnuCL (the host $\rightarrow$ a compute node $\rightarrow$ a GPU). Therefore, the performance difference

between SnuCL and SnuCL-D is observed even with a small number of nodes. In `MatrixMul`, SnuCL-D performs 1.3 times and 2.7 times faster than SnuCL for 32 nodes on the large-scale CPU cluster and the medium-scale GPU cluster, respectively.

Since `EP` is an embarrassingly parallel application, it scales well with both SnuCL and SnuCL-D. Even if SnuCL-D scales well on the large-scale CPU cluster in Section IV.7.3, it does not scale well with `FT`, `CG`, `MG`, and `SP` on the GPU cluster. The reason is the relatively small input size. The total amount of work in each kernel is too small to amortize the OpenCL runtime overhead. In addition, the computing power of a GPU is much greater than that of a CPU. Since the total amount of memory in a GPU is limited, it is impossible to increase the input size further. This is the reason of why SnuCL-D shows inferior scalability for those four applications in the GPU cluster.

SnuCL-D performs significantly better than SnuCL especially for a large number of nodes. Since `CG`, `MG`, `SP`, and `BT` execute a large number of commands, the enqueueing, scheduling, and delivering overhead in SnuCL is more significant than that of SnuCL-D.

Chapter V

Conclusion and Future Work

This thesis presents two limitations of OpenCL: programming complexity and the lack of support for a heterogeneous cluster. It is not easy for application developers to write a program using OpenCL because they have to write coordination code and kernel code as well. It requires application developers to understand language extensions and various API functions. It adds maintenance cost and reduces productivity. For ease of programming, this thesis proposes a translation framework that converts an OpenMP program to an OpenCL program. Application developers can easily insert OpenMP compiler directives to an existing sequential program. Then, the pro-

posed translation framework converts the OpenMP program to an efficient OpenCL program. Due to OpenCL's portability, the OpenMP program can be executed on any hardware platform that supports OpenCL after the translation. This thesis also proposes two techniques for ease of programming and high performance: data transfer minimization (DTM) and performance portability enhancement (PPE). Using various benchmark applications, this thesis shows the translation framework is an efficient and practical solution.

Since OpenCL only focuses on a single OS instance, this thesis proposes SnuCL-D, a scalable and decentralized OpenCL framework for a heterogeneous cluster to provide ease of programming. Unlike previous OpenCL frameworks for a heterogeneous cluster, SnuCL-D executes the host program on every node by exploiting redundant computation and data replication. This thesis analyzes three overheads: delivering overhead, scheduling overhead, and enqueueing overhead. The delivering overhead is reduced by the redundant computation by a nature of the decentralization. To reduce the scheduling overhead and the enqueueing overhead, this thesis proposes a new OpenCL API function and queueing optimization. By reducing these kinds of overhead, SnuCL-D outperforms SnuCL, our previous OpenCL framework that applies the centralized approach.. Moreover, SnuCL-D performs comparable to MPI for a large-scale cluster.

With the two proposed frameworks, this thesis builds an environment that provides ease of programming for heterogeneous systems.

Application developers writes an OpenMP program for an accelerator, then the program is executed on an OpenCL framework with the proposed translation framework. Application developers write an OpenCL program for multiple compute devices, then the program is executed on a heterogeneous cluster with SnuCL-D.

There can be several directions to exploit this research. For lowering the programming complexity of OpenCL, three future research directions are proposed. First, this thesis targets OpenMP 4.0 currently, and there are some changes in OpenMP 4.5 which is the latest version of OpenMP. OpenMP 4.5 adds more features to handle accelerators efficiently. Supporting OpenMP 4.5 is required in the future. Second, more optimization techniques can be integrated in the framework to efficiently convert an OpenMP program to OpenCL. For example, efficient scheduling and reduction policies for other accelerators such as FPGAs, DSPs, and Intel Xeon Phi coprocessors can be developed. Third, the framework can generate better optimized code for specific hardware. For example, code that exploits the local memory in a GPU can be automatically generated by the translation framework.

For future work of supporting a heterogeneous cluster, there is a remaining research topic for SnuCL-D to become a complete OpenCL framework. Currently, SnuCL-D cannot work correctly if there are two or more OpenCL host threads. This is because the ordering of commands are defined by the time when they are enqueued. This is a limitation of SnuCL-D, and it should be solved in the future to

support multi-threaded OpenCL host programs.

There are two directions that can be taken in order to extend this research using both frameworks.

First, an OpenMP program that uses a single accelerator can be transparently executed on a heterogeneous cluster. In this case, an automatic workload distribution mechanism should be developed. Specifically, iterations of a `for` loop are distributed to all compute devices on a single OS instance. This is transparently extended to a heterogeneous cluster with SnuCL-D. Eventually, an OpenMP program using an accelerator can be executed on a heterogeneous cluster. This is the ultimate goal of ease of programming for heterogeneous systems.

Second, an OpenMP program that explicitly uses multiple accelerators is converted to an OpenCL program. In this case, an efficient method how to exploit multiple accelerators should be proposed using OpenMP. OpenMP has a notion of multiple devices in the specification, but it is unknown how to use multiple target devices efficiently. Then, an efficient translation technique should be proposed to translate an OpenMP program to OpenCL.

Bibliography

[1] Charm++. Website. http://charm.cs.uiuc.edu/.

[2] GCC, the GNU Compiler Collection. Website. https://gcc.gnu.org/.

[3] OpenCL™ Optimization Case Study: Simple Reductions. Website. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/.

[4] OpenMP/Clang. Website. https://clang-omp.github.io/.

[5] PGI Compilers and Tools. Website. http://www.pgroup.com/.

[6] *OpenACC 2.0a Specification*. June 2013.

[7] *OpenMP 4.0 Specification*. July 2013.

[8] *AMD OpenCL Optimization Guide*. August 2015.

[9] *OpenMP 4.5 Specification*. November 2015.

[10] *SYCL 1.2 Specification*. May 2015.

[11] Top500 lists. Website, June 2015. http://top500.org/lists/2015/06/.

[12] Giovanni Aloisio and Sandro Fiore. Towards Exascale Distributed Data Management. *International Journal of High Performance Computing Applications*, 23(4):398–400, 2009.

[13] Albano Alves, José Rufino, António Pina, and Luís Paulo Santos. clOpenCL: Supporting Distributed Heterogeneous Computing in HPC Clusters. In *Proceedings of the 18th International Conference on Parallel Processing Workshops*, Euro-Par'12, pages 112–122, Berlin, Heidelberg, 2013. Springer-Verlag.

[14] AMD. AMD Accelerated Parallel Processing (APP) SDK. Website, January 2014. `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`.

[15] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 109–126, New York, NY, USA, 1995. ACM.

[16] Ryo Aoki, Shuichi Oikawa, Ryoji Tsuchiyama, and Takashi Nakamura. Hybrid OpenCL: Connecting Different OpenCL Implementations over Network. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 2729–2735, Washington, DC, USA, 2010. IEEE Computer Society.

[17] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Samuel Thibault, and Raymond Namyst. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. Research Report RR-8538, May 2014.

[18] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task

scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[19] Eduard Ayguade, RosaM. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc Gonzalez, Francisco Igual, Daniel Jimenez-Gonzalez, Jesus Labarta, Luis Martinell, Xavier Martorell, Rafael Mayo, JosepM. Perez, Judit Planas, and Enrique S. Quintana-Orti. Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *International Journal of Parallel Programming*, 38(5-6):440–459, 2010.

[20] Eduard Ayguade, RosaM. Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez, Jesus Labarta, Xavier Martorell, Rafael Mayo, JosepM. Perez, and Enrique S. Quintana-Orti. A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In MatthiasS. Müller, BronisR. de Supinski, and BarbaraM. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 154–167. Springer Berlin Heidelberg, 2009.

[21] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–7, Sept 2010.

[22] JamesC. Beyer, EricJ. Stotzer, Alistair Hart, and BronisR. de Supinski. OpenMP for Accelerators. In BarbaraM. Chapman, WilliamD. Gropp, Kalyan Kumaran, and MatthiasS. Müller, editors, *OpenMP in the Petascale Era*, volume 6665 of *Lecture Notes in Computer Science*, pages 108–121. Springer Berlin Heidelberg, 2011.

[23] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[24] J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568, May 2012.

[25] Casten. SocketCL. Website. `http://sourceforge.net/projects/socketcl`.

[26] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *Computer*, 23(6):49–58, June 1990.

[27] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, 2009.

[28] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[29] J. Duato, A.J. Pena, F. Silla, R. Mayo, and E.S. Quintana-Orti. rCUDA: Reducing the number of GPU-based accelerators in high

performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 224–231, June 2010.

[30] Alejandro Duran, Eduard Ayguade, Rosa M. Badia, Jesus Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[31] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. LibWater: Heterogeneous Distributed Computing Made Easy. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 161–172, New York, NY, USA, 2013. ACM.

[32] Khronos Group. *OpenCL 1.2 Specification*. Khronos Group, November 2012. `http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/`.

[33] Khronos Group. *OpenCL 2.0 Specification*. Khronos Group, November 2013. `http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/`.

[34] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin Peaks: A Software Platform for Heterogeneous Computing on General-purpose and Graphics Processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[35] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM.

[36] Timothy D.R. Hartley, Erik Saule, and Ümit V. Çatalyürek. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing*, 38(6–7):289 – 309, 2012.

[37] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[38] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, 6(1):51–81, February 1988.

[39] Byunghyun Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):105–118, Jan 2011.

[40] P. Kegel, M. Steuwer, and S. Gorlatch. dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 174–186, May 2012.

[41] Pete Keleher, Alan L. Cox, Sandhay Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Technical Conference*, January 1994.

[42] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[43] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

[44] David M. Kunzman and Laxmikant V. Kalé. Programming Heterogeneous Clusters with Accelerators Using Object-Based Programming. *Scientific Programming*, 19(1):47–62, 2011.

[45] Björn König. CLara - OpenCL across the net. Website. `http://sourceforge.net/projects/clara`.

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

[47] Chunhua Liao, Yonghong Yan, BronisR. de Supinski, DanielJ. Quinlan, and Barbara Chapman. Early Experiences with the OpenMP Accelerator Model. In AlistairP. Rendell, BarbaraM. Chapman, and MatthiasS. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 84–98. Springer Berlin Heidelberg, 2013.

[48] Drew Major, Greg Minshall, and Kyle Powell. An Overview of the NetWare Operating System. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 27–27, Berkeley, CA, USA, 1994. USENIX Association.

[49] MPI Forum. MPI: A Message Passing Interface Standard. Version 3. Website, 2012. `http://www.mpi-forum.org`.

[50] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks version 3.3. Website. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[51] NVIDIA. NVIDIA CUDA Toolkit 4.0. Website. `http://developer.nvidia.com/cuda-toolkit-40`.

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

[53] NVIDIA. CUDA Zone. Website, January 2014. `http://www.nvidia.com/object/cuda_home_new.html`.

[54] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1207–1214, Nov 2012.

[55] OpenACC. *OpenACC Programming and Best Practices Guide*, June 2015.

[56] Antonio J. Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí, and José Duato. A complete and efficient CUDA-sharing solution for {HPC} clusters. *Parallel Computing*, 40(10):574 – 588, 2014.

[57] Dan Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[58] C. Reano, A.J. Pea, F. Silla, J. Duato, R. Mayo, and E.S. Quintana-Orti. CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10, Dec 2012.

[59] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An Experimental Study on Performance Portability of OpenCL Kernels. In *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, page 3, 2010.

[60] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[61] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen mei W. Hwu. Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing*, 68(10):1389 – 1401, 2008. General-Purpose Processing using Graphics Processing Units.

[62] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation or the Sun Network Filesystem. In *USENIX*, 1985.

[63] Mahadev Satyanarayanan. A Survey of Distributed File Systems. *Annual Review of Computer Science*, 4(1):73–104, 1990.

[64] T.R.W. Scogland, B. Rountree, Wu chun Feng, and B.R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 144–155, May 2012.

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

[66] Jie Shen, Jianbin Fang, H. Sips, and A.L. Varbanescu. Performance Traps in OpenCL for CPUs. In *Parallel, Distributed and*

Network-Based Processing (PDP), 2013 21st Euromicro International Conference on, pages 38–45, Feb 2013.

[67] P. Stenstrom. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12 –24, june 1990.

[68] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, vLi Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, University of Illinois at Urbana-Champaign, March 2012.

[69] Tran Doan Thanh, Subaji Mohan, Eunmi Choi, SangBum Kim, and Pilsung Kim. A Taxonomy and Survey on Distributed File Systems. In *Networked Computing and Advanced Information Management, 2008. NCM'08. Fourth International Conference on*, volume 1, pages 144–149. IEEE, 2008.

[70] André Tupinamba. DistributedCL. Website. `https://github.com/andrelrt/distributedcl`.

[71] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[72] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An Adaptive Data Replication Algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, June 1997.

[73] Alan Woodland. CLuMPI (OpenCL under MPI). Website. `http://sourceforge.net/projects/clumpi`.

[74] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU Compiler for Memory Optimization and Parallelism

Management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.

[75] Yao Zhang, II Sinclair, Mark, and AndrewA. Chien. Improving Performance Portability in OpenCL Programs. In JulianMartin Kunkel, Thomas Ludwig, and HansWerner Meuer, editors, *Supercomputing*, volume 7905 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin Heidelberg, 2013.

초 록

OpenCL은 이종 시스템을 위한 주요한 프로그래밍 모델 중 하나이다. 본 논문은 OpenCL의 두 가지 한계점에 대해 논하고, 쉬운 프로그래밍을 위하여 각 한계점에 대한 해결책을 제시한다.

첫 번째 한계점은 OpenCL을 이용하여 프로그래밍하기 복잡하다는 점이다. 이 프로그래밍 복잡도를 낮추기 위해, 본 논문에서는 높은 수준의 프로그래밍 언어(OpenMP)를 이용한 프로그램을 OpenCL로 변환하는 프레임워크를 제안한다. 또한, 본 논문에서는 쉬운 프로그래밍과 고성능을 달성하기 위하여 두 가지 기법(데이터 전송 최소화 (DTM), 성능 이식성 향상(PPE))을 제시한다. 제안한 변환 프레임워크를 다양한 하드웨어 플랫폼에서 다양한 벤치마크 프로그램을 실행하여 효율성을 검증하였다. 또한, 상용 컴파일러인 PGI 컴파일러와의 성능 평가를 통해 실용성을 검증하였다.

두 번째 한계점은 OpenCL이 이종 클러스터를 지원하지 않는다는 점이다. OpenCL을 이종 클러스터로 확장하기 위해서, 본 논문은 SnuCL-D라는 성능 확장성이 있고 분산된 OpenCL 프레임워크를

제안한다. SnuCL-D는 OpenCL만으로 작성된 프로그램을 이종 클러스터에서 실행될 수 있게 하는 프레임워크이다. 중앙 집중적 방식(centralized approach)를 이용한 기존의 연구와는 다르게, 본 논문에서 제안하는 프레임워크는 분산 방식(decentralized approach)를 적용하였다. 또한, 본 논문에서는 커맨드의 실행 경로에서 발생하는 세 가지 오버헤드를 분석하고 각 오버헤드를 줄이는 기법을 제시하였다. 실험을 통해 대규모 CPU 클러스터 및 중규모 GPU 클러스터에서 제안하는 프레임워크가 좋은 성능 확장성을 보이는 것을 확인하였다. 또한, 기존의 중앙 집중적 방식인 SnuCL과 분산 컴퓨팅 환경에서 사실상 표준 통신 기법인 MPI와의 비교를 통해 제시하는 프레임워크가 효율적이고 실용적임을 확인하였다.

본 논문에서 제시한 두 가지의 프레임워크를 통해, 응용 프로그램 개발자는 OpenMP를 이용하여 OpenCL을 지원하는 다양한 가속기를 쉽게 이용할 수 있을 뿐만 아니라, OpenCL만을 이용해 작성한 프로그램을 이종 클러스터 환경에서도 쉽게 실행시킬 수 있을 것이라 기대한다.

주 요 어 : 오픈엠피, 오픈씨엘, 쉬운 프로그래밍, 고성능, 클러스터, 이기종 시스템, 프로그래밍 모델, 가속기, 벤치마크

학 번 : 2009-30183