A STRATEGY FOR MAPPING THREADS TO GPUS IN A DIRECTIVE-BASED PROGRAMMING MODEL

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Chen Shen

April 2017

A STRATEGY FOR MAPPING THREADS TO GPUS IN A DIRECTIVE-BASED PROGRAMMING MODEL

Chen Shen

APPROVED:

Edgar Gabriel, Chairman
Dept. of Computer Science

Barbara Chapman
Stony Brook University

Lennart Johnsson
Dept. of Computer Science

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I would like to express my deepest gratitude to my advisor, Dr. Barbara Chapman, offerring me the opportunity with guidance and support to study in this exciting research area. She always give me trust with her patience when I was encountering difficulties in the research projects. I would also like to thank my other committee members, Dr.Edgar Gabriel and Dr. Lennart Johnsson, for their taking time to review my work. I have also taken courses from them. I really benefit a lot of HPC fundamentals through their courses.

I would like to thank my former mentor, Dr.Sunita Chandrasekaran, for her guidance and a lot of help on my first research project in HPCTools group.

I would like to thank my former group member, Xiaonan Tian. He has given me great help and valuable suggestions on my projects. He is always generous has rich experience in his field of work.

I would like to thank Dounia Khaldi for her helping me organizing and reviewing my work.

I would like to thank all HPCTools group members, they are all very nice and helpful, I had a great time working with them.

Finally, I feel very grateful to my parents for their love and support all the time.

A STRATEGY FOR MAPPING THREADS TO GPUS IN A DIRECTIVE-BASED PROGRAMMING MODEL

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Chen Shen

April 2017

Abstract

The proliferation of accelerators in modern clusters makes efficient coprocessor programming a key requirement if application codes are to achieve high levels of performance with acceptable energy consumption on such platforms. This has led to considerable effort to provide suitable programming models for these accelerators, especially within the OpenMP community. While OpenMP 4.5 offers a rich set of directives, clauses and runtime calls to fully utilize accelerators, an efficient implementation of OpenMP 4.5 for GPUs remains a non-trivial task, given their multiple levels of thread parallelism.

In this thesis, we describe a new implementation of the corresponding features of OpenMP 4.5 for GPUs based on a one-to-one mapping of its loop hierarchy parallelism to the GPU thread hierarchy. We assess the impact of this mapping, in particular the use of GPU warps to handle innermost loop execution, on the performance of GPU execution via a set of benchmarks that include a version of the NAS parallel benchmarks specifically developed for this research; we also used the Matrix-Matrix multiplication, Jacobi, Gauss and Laplacian kernels for better understanding the potential performance issues.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Most modern clusters include nodes with attached accelerators, NVIDIA GPU accelerators being the most commonly used. But recently, accelerators such as AMD GPUs, Intel Xeon Phis, DSPs, and FPGAs have been explored as well. The energy efficiency of accelerators and their resulting proliferation has made coprocessor programming essential if application codes are to efficiently exploit HPC platforms. As a result, considerable effort has been made to provide suitable programming models.

One may first distinguish between low-level and high-level programming libraries and languages; the user has to choose one programming API that suits her application code, level of programming, and timing. CUDA [1] and OpenCL [24] are low-level programming libraries. On the other hand, high-level directive-based APIs

for programming accelerators, such as OpenMP [12] and OpenACC [23], are easy to use by scientists, preserve high-level language features, and provide good performance portability features. Recently, multiple open-source compilers (e.g., GCC, LLVM, OpenUH) and commercial compilers such as Cray, IBM, Intel and PGI, have added support for these directive-based APIs (or are in the process of doing so).

OpenMP's multithreading features are already employed in many codes to exploit multicore systems. The broadly available standard has been rapidly evolving to meet the requirements of heterogeneous nodes. In 2013, Version 4.0 of the OpenMP specification made it possible for user-selected computations to be offloaded onto accelerators; some important features were added in version 4.5 [8, 9]. Implementations within vendor compilers are under way.

OpenMP 4.0 added a number of device constructs: `target` can be used to specify a region that should be launched on a device and define a data device environment, and `target data`, to map variables on that device. Pragma `teams` can be used inside `target` to spawn a set of teams, each containing multiple OpenMP threads; note that `teams` can be mapped to a CUDA block. Finally, `distribute` is used to partition the iterations of an enclosed loop to each team of such a set.

Compared to OpenMP 4.0, OpenMP 4.5 changed the semantics of the mapping of scalar variables in C/C++ target regions. It also provides support for asynchronous offloading using `nowait` and `depend` clauses on the `target` construct. The mapping of data can also be performed asynchronously in OpenMP 4.5 using `target enter data` and `target exit data`. The clause `is_device_ptr` was added to `target` to

indicate that a variable is a device pointer that is already in the device data environment, so it should be used directly. The clause `use_device_ptr` has been added to `target data` to convert variables into device pointers to the corresponding variable in the device data environment. Finally, device memory routines were added to support explicit allocation of memory and memory transfers between hosts and offloading devices, such as `omp_target_alloc`.

A typical GPU architecture, for which some of the above-mentioned features were designed, provides three levels of parallelism, namely thread blocks, warps, and threads inside each warp. OpenMP 4.5 provides three levels of loop hierarchy parallelism as well, namely teams, parallel, and SIMD. For traditional CPU architectures and CPU-like accelerators, the mapping of these three levels of parallelism is straightforward. However, a typical GPU architecture differs substantially from a CPU, and thus the design of an efficient implementation of OpenMP 4.5 on GPUs is not a trivial task. In this work, we describe an efficient implementation of the OpenMP offloading constructs in the open source OpenUH compiler [10] for NVIDIA GPUs.

Our implementation of OpenMP 4.x offloading is based on a one-to-one mapping of OpenMP levels of parallelism to CUDA's levels of parallelism, i.e, grid, thread block, and warp. To assess the suitability of this approach, we also implemented an OpenMP accelerator version of the NAS Parallel Benchmarks, and used it to test our implementation.

1.2 Contribution

This thesis makes the following contributions:

- we propose a one-to-one mapping of the loop hierarchy parallelism available in OpenMP 4.x to the GPU thread hierarchy and implement this mapping in the OpenUH compiler;
- we use a set of benchmarks to assess the impact of this mapping, especially the use of GPU warps to handle innermost loop execution, on the performance of GPU execution;
- we adapt the NAS parallel benchmarks to use OpenMP offloading, and make them available for use by the OpenMP community to test other implementations and platforms.

1.3 Organization

Part of this thesis has been published in [26]. The organization of this thesis is as follows. Chapter 2 gives a brief introduction of modern accelerators' architectures. Based on those hardware platforms, we go through major programming models designed for accelerators. Chapter 3 will discuss about current support of other compiler vendor's implementation of OpenMP 4.x. Then we introduce the OpenUH compiler infrastructure. Chapter 4 reviews the OpenMP 4.x programming model using most commonly used directives. Chapter 5 will discuss about how the offloading

support is integrated in OpenUH. Chapter 6 describes the one-to-one loop hierarchy mapping to the GPU thread hierarchy that we have adopted. Performance results, comparison and discussions are given in Chapter 7. Chapter 8 concludes the whole work in the thesis and looks into some future directions of the current work.

Chapter 2

Background

In this chapter, we provide a brief overview of the architecture of modern GPUs and Intel Xeon Phi accelerator, then we will give a brief introduction of high-level and low-level prevailing programming models.

2.1 Modern GPU Architecture

Modern GPUs consist of multiple streaming multiprocessors (SMs or SMXs); each SM consists of many scalar processors (SPs, also referred to as cores). Each GPU supports the concurrent execution of hundreds to thousands of threads following the Single Instruction Multiple Threads (SIMT) paradigm, and each thread is executed by a scalar core. The smallest scheduling and execution unit is called a warp, which is typically composed of 32 threads. Warps of threads are grouped together into a thread *block*, and blocks are grouped into a *grid*. Both the thread blocks and the

grid can be organized into a one-, two- or three-dimensional topology.

SIMT execution within a warp in GPUs can be compared to SIMD execution in a CPU. In each case, the same instruction is broadcasted to multiple arithmetic units. However, the CPU comes with vector instructions to process several elements of short arrays in parallel, whereas, in SIMT mode (within a warp), several elements of an array are processed in parallel.

Modern GPUs deploy a deep memory hierarchy, which includes several different memory spaces. Each of them has special properties. The global memory is the main memory in GPUs that all threads can make use of it. The so-called shared memory is shared by threads within a thread block only. The texture memory is read-only memory that allows adjacent reads in a warp.

Each kind of memory requires specific attention. For example, accesses to global memory may be coalesced or not; accesses to texture memory could come with a spatial locality penalty; accesses to shared memory might suffer from bank conflicts; and accesses to constant memory may be broadcast. Memory coalescing is a key enabler of data locality in modern GPU architectures. Under it, memory requests by individual threads in a warp are grouped together into large transactions. When consecutive threads access consecutive memory addresses, this enables the exploitation of spatial data locality within a warp.

Modern NVIDIA GPUs are using SMXs as the fundamental part of the processing core architecture. Each SMX unit consists of a lot of CUDA cores. Each CUDA core contains arithmetic logic units for integer and float point operations. There are

hardware level scheduler inside each SMX for thread scheduling. The GPU threads are scheduled in groups consists of 32 threads. At a certain time, the warp scheduler will select several warps to issue concurrently.

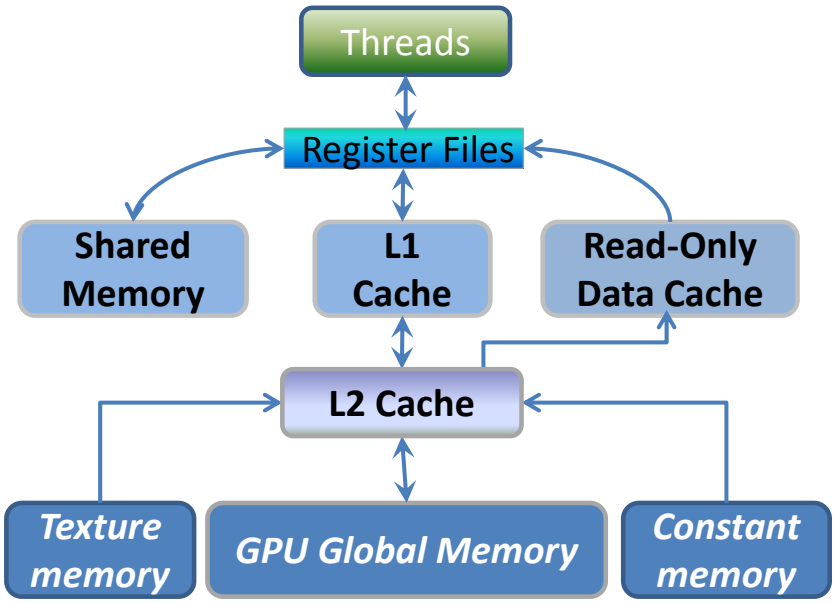

Figure 2.1: Memory hierarchy of NVIDIA GPUs

NVIDIA GPUs also have a deep memory hierarchy, each GPU thread can load data from a variety of types of memory spaces. The closest memory for each compute thread is the thread local register files, the farthest memory for each thread on the device is in GPU gloabal memory. The global memory is connected with host memory via PCI-Express interface. Threads within each SMX are able to share L1 cache, read-only cache and shared memory. The constant memory and texture memory are bundled with the global memory, they will be loaded into L2 cache.

2.2 Intel Xeon Phi Architecture

Xeon Phi is a name of a series of Intel branded manycore processors, it is targeting the high performance computing markets. It supports some common parallel programming models such as OpenMP and OpenCL [4]. The Xeon Phi contains many low-power architecture cores [28]. Those cores are organized in a ring interconnect. There are four hardware threads of each core. Each core has two pipelines: scalar processing and vector processing. The scalar and vector processing units can share data from L1 and L2 cache. The vector processing unit (VPU) features a 512-bit length SIMD instruction set. The VPU contains mask registers for per lane predicted execution. The interconnect ring of the cores is bidirectional. The interconnect ring provides the cores with the ability to share data, commands and addresses.

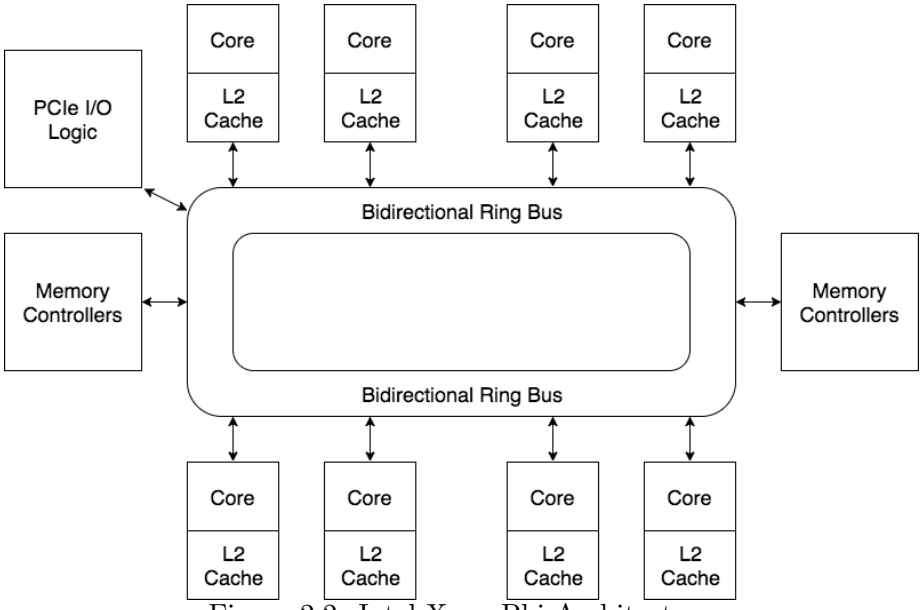

Figure 2.2: Intel Xeon Phi Architecture

The global memory is placed in a interleaved manner with compute cores. At a

certain time when a core is requesting memory, it will fetch from the L2 cache, if there is a cache miss, an address request will be generated and put on the interconnect ring, the request will be captured by the global memory controller, then the data will be fetched from global memory and sent back to the ring.

Chapter 3

Related Work

3.1 Related Implementations of OpenMP Target Model

Support for OpenMP 4.5 in vendor compilers is still a work in progress. The Intel compiler 16.0, released in August 2015, fully supports OpenMP 4.0 for the Intel Xeon Phi coprocessor. The Cray Compiling Environment 8.4 was released in September 2015. It provides support for the OpenMP 4.0 specification for C, C++, and Fortran, enabling the execution of OpenMP 4.0 `target` regions on NVIDIA GPUs.

Regarding open-source compilers, GCC 6 was released in April 2016; it fully supports the OpenMP 4.5 specification for C and C++. GCC targets Intel XeonPhi Knights Landing and AMD's HSAIL (Heterogeneous System Architecture Intermediate Language).

LLVM 3.8.0 was released in March 2016. Its frontend Clang supports all of OpenMP 3.1 and some elements of OpenMP 4.0 and 4.5. LLVM uses an OpenMP offloading library called *libomptarget*. The current implementation of this library can be divided into three components: target-agnostic offloading, target-specific offloading plugins, and target-specific runtime library. The *libomptarget* library was designed with the goal of supporting multiple devices; it currently targets the IBM Power architecture, x86_64, Nvidia GPUs, and Intel Xeon Phi. In [5], its OpenMP 4.0 offloading strategy is described and some optimizations are applied, such as a control loop scheme [6] that avoids dynamic spawning of GPU threads inside the target region; this implementation targets NVIDIA GPUs. As already discussed in the experimental section, the current implementation of LLVM does not efficiently map the `simd` OpenMP level to GPU warps, thereby hurting performance.

3.2 Current OpenUH Infrastructure

The OpenUH compiler [10] is a branch of the open-source Open64 compiler suite for C, C++, and Fortran 95/2003. It offers a complete implementation of OpenACC 1.0 [27], Fortran Coarrays [14], and OpenMP3.0 [19]. OpenUH is an industrial-strength optimizing compiler that integrates all the components needed of a modern compiler; it serves as a parallel programming infrastructure in the compiler research community. OpenUH has been the basis for a broad range of research endeavors, such as language research [15, 16, 14], static analysis of parallel programs [11], performance analysis [25], task scheduling [2] and dynamic optimization [7].

The major functional parts of the compiler are the front ends (the Fortran 95 front end was originally developed by Cray Research and the C/C++ front end comes from GNU GCC 4.2.0), the inter-procedural analyzer/optimizer (IPA/IPO) and the middle-end/backend, which is further subdivided into the loop nest optimizer (LNO), including an auto-parallelizer (with an OpenMP optimization module), global optimizer (WOPT), and code generator (CG). Currently, x86-64, IA-64, IA-32, MIPS, and PTX are supported by its backend. OpenUH may also be used as a source-to-source compiler for other machines using its IR (Intermediate Representation)-to-source tools. OpenUH uses a tree-based IR called WHIRL which comprises 5 levels, from Very High (VH) to Very Low (VL), to enable a broad range of optimizations. This design allows the compiler to perform various optimizations on different levels.

Chapter 4

OpenMP 4.x Target Model Application Programming Interface

OpenMP 4.x target model makes it possible for user-selected computations to be offloaded onto accelerators. It allows programmers to add directives as hints to the compiler, and the compiler will map those directives to compute regions on the accelerator. These directives includes data management and thread management. It extends its original shared memory model to multiple devices. Programmers can easily adapt their serialized code to parallel version with minimum modification.

In this chapter, we will first describe the memory and execution model of OpenMP 4.x target model, then we will give a brief introduction of major offloading directives according to OpenMP 4.x specification. We will also provide some code examples to

illustrate to usage of those directives.

4.1 Memory Model

OpenMP 4.x supports offloading compute regions to compute devices. Since each compute device has its own memory space, the device and host can not have access of data of each other directly. It is necessary to provide a mechanism to support data transferring between device and host. In OpenMP 4.x target model, each device has its own data environment and maintains a shared-memory environment for device threads.

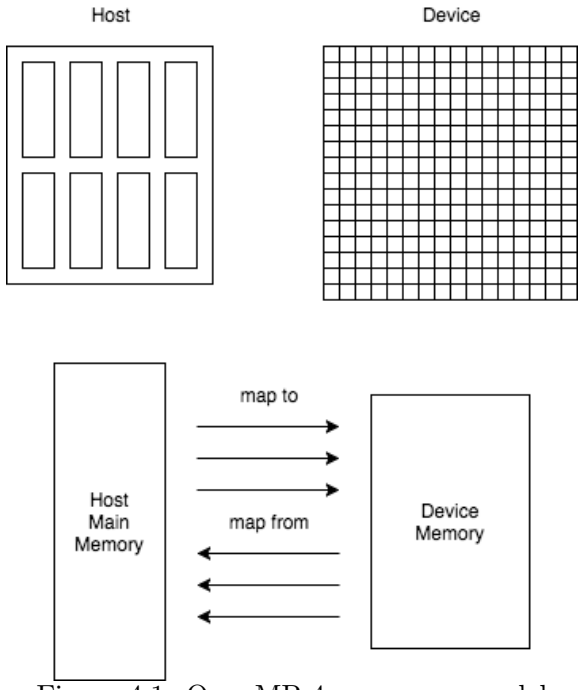

Figure 4.1: OpenMP 4.x memory model

Explicit data movement can be accomplished through a variety of data mapping

constructs. Those data movement operations are implemented by the compiler and runtime, the compiler will generate low-level data mapping APIs from high-level OpenMP directives, and at runtime the data transferring will happen between host and device. This mechanism is very similar to some existing parallel programming models such as OpenACC, CUDA and OpenCL, those models also suppose discrete memory spaces and thus data mapping APIs are important.

The separate memory space between host and device implies that it is the users responsibility to carefully manage those data movements. The user should consider data movement overhead while exchanging data since the transferring cost could be high; the device memory size could be smaller than host memory so that multiple data exchanged could be used; it is important to maintain a reliable data synchronization mechanism between host and device.

4.2 Execution Model

The accelerators are used to parallelize compute-intensive tasks on devices. Since the host and the device are supposed to have separate memory space, the accelerators need to provide thread management for its offloading region. The host should guide the execution at the entry and the exit of the offloading region. The compute kernels should be executed in parallel on the device. This host-centric execution model requires that the program should begin with some host threads, when the program encounters the OpenMP target region, a target task will be generated and the data referenced by the compute region will be implicitly allocated and initiated on the

device, device threads will then be created and execute the computation task. The host thread which creates the offloading task will wait until the device has finished its job. At the exit of the compute region, device data will be transferred back to the host threads.

OpenMP target model provides a variety of device constructs for users to control the data transferring, thread spawning and executing. Those constructs could transfer the control flow of the host threads to the device threads in a scoped device data environment. Users should consider the tradeoffs between host executing and device executing since they may have quite different architecture.

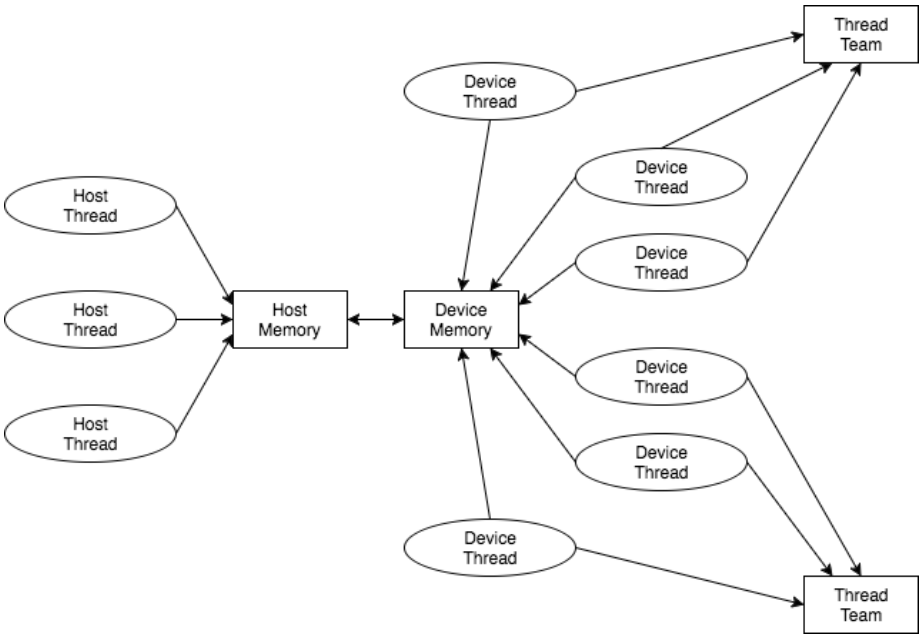

Figure 4.2: OpenMP 4.x execution model

OpenMP 4.x provides up to three levels of parallelism, each level contains bundles of threads. Most of the state-of-the-art accelerators are capable of using three levels of parallelism. However, due to the hardware design differences, those accelerator

could vary a lot in architecture, a high-level abstraction of thread hierarchy should be well defined. In general, each accelerator may contain some groups of execution units [18] [21], each group of execution units contain multiple basic execution threads, each thread may have its own SIMD lane which could issue SIMD instructions. OpenMP target model contains teams, parallel and SIMD level parallelism. The coarse-grain team's level bundles a number of execution threads produced by hardware units. The fine-grain parallel level will spread workload across threads in each team. The SIMD level provides vectorization for loops with SIMD vector registers. The compiler should be able to choose the best mapping strategies for different types of accelerators, and the user will also be responsible to better utilize those directives to reach good performance.

4.3 Offloading Directives

We will discuss the definition and usage of the most commonly used directives in OpenMP 4.x target model. Those directives are supported in current OpenUH implementation and will cover most use cases in real-world applications.

4.3.1 Data Transferring Constructs

The current OpenUH supports `target data` and `target update` constructs. The `target data` construct encloses a structured block, at the entry of the structured block, the device data environment will be created but the device compute kernels

will be launched after the `target` construct is encountered.

```
#pragma omp target data map(tofrom: a[0:N], b[0:N], c[0:N])
{
       c[i] = a[i] + b[i];
}
```

Figure 4.3: OpenMP 4.x data transferring construct

Note that if there is no target region specified inside this block, the execution will still be done on the host side, the `target data` construct only create the device data environment but not transfer the control flow. The `map` clause is always associated with target data construct, it is used to describe the data referenced in the device data environment. Users can control the data flow and data range by specifying the map-type in the syntax. The data variable listed in the `map` clause could either be a scalar or an array pointer, if it is a array pointer, users can specify its data range, the array section should reside in an contiguous memory space. The `map` clause provides several map-type keywords for users to manage data movement. The alloc-type means the data will be created on the device and with undefined initial value; the to-type means the data will be created and initialized with the host value; the from-type means the data will be copied from the device back to host; the tofrom-type is a combination of to-type and from-type.

```
#pragma omp target data map(to: a[0:N], b[0:N]) map(alloc: c[0:N])
{
       c[i] = a[i] + b[i];
}
#pragma omp target update from(c[0:N])
```

Figure 4.4: OpenMP 4.x data updating construct

The `target update` construct is used to maintain consistency of host data environment and device data environment. There are two motion-clauses associated

with this construct, the to-clause assigns the value of the original array section to the corresponding device array section, the from-clause assigns the value of the device array section to the original host array section. If the list array section does not exist on host or device, the data assignment will not happen.

4.3.2 Target Construct

The `target` construct encloses a structured block that will be executed on the device. When the host thread encounters the `target` construct, a device task is generated and the control flow of the host thread will be offloaded on the device. The `map` clause can also be applied to `target` construct, which means that the data environment will also be created at the entry of the target region, and the behavior of `map` clause is identical with `target data` construct. Inside the `target` construct, some thread creating constructs are always included for parallel execution of the device task. Figure 4.5 shows an example code snippet.

```
#pragma omp target data map(to: a[0:N], b[0:N], c[0:N]) map(tofrom: d[0:N])
{
	#pragma omp target
	#pragma omp parallel for
	for (int i = 0; i < N; ++i)
		b[i] = a[i];

	#pragma omp target
	#pragma omp parallel for
	for (int j = 0; j < N; ++j)
		c[j] = b[j] + 2;

	#pragma omp target
	#pragma omp parallel for
	for (int k = 0; k < N; ++k)
		d[k] = c[k] + b[k];
}
```

Figure 4.5: OpenMP 4.x target construct

In the above figure, there are multiple target regions inside the `target data` block, this is the situation that multiple compute kernels are launched, each kernel has its own loop scheduling plan. At the end of each kernel region, the target task finishes its job and the data remains in the device memory, then at the end of the `target data` region, device data with map-type `tofrom` will be copied back to host.

4.3.3 Teams and Distribute Constructs

The `teams` construct is used to create a league of thread teams on the device, each team contains a number of threads. The number of teams is determined by `num_teams` clause and the number of threads is determined by `thread_limit` clause. The number of teams will not be changed during execution of the `omp teams` region. During the execution period of the teams construct, only the master thread is active and other thread are idle, the other thread will not be involving in the execution until a further parallel region is specified. Note that each master thread of the teams will do the complete iterations, this can be explained in the following Figure.

Figure 4.6 shows that the highlighted master thread of each team will have the same execution plan, the complete loops enclosed by the `teams` construct will executed by master threads. The `teams` construct should be strictly contained within a `target` construct. At the end of the `teams` region, the encountering thread which triggers the creation of thread teams will continue its execution in the `target` region.

The `distribute` construct is as the name suggests, the execution of the associated loop will be spread across the master threads of each thread team, each master thread

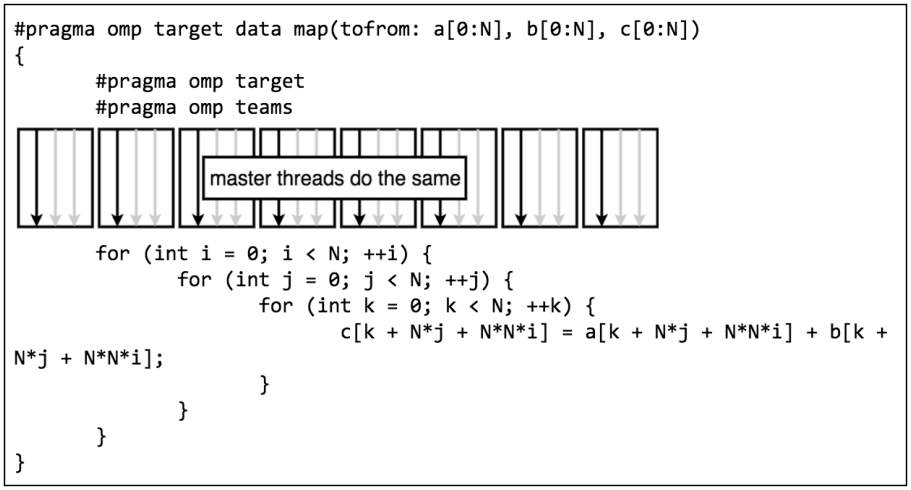

Figure 4.6: OpenMP 4.x teams construct

will obtain a number of iterations.

Figure 4.7 shows that `distribute` construct behaves like a worksharing construct, the outer loop will be shared among master threads. The `distribute` construct should be tightly nested inside the `teams` construct.

4.3.4 Parallel Loop Construct

Before OpenMP 4.x, the `parallel` loop construct is a combined construct which will distribute the execution of its associated loop among the threads of the team. Each thread will get a few iterations of its binded loop. OpenMP 4.x has extended its usage to the scope of device threads. After a team of threads is created on the device, the workloads of its associated loop will be shared by the team members including the master thread. This is the fine-grain level of parallelism for most accelerator types.

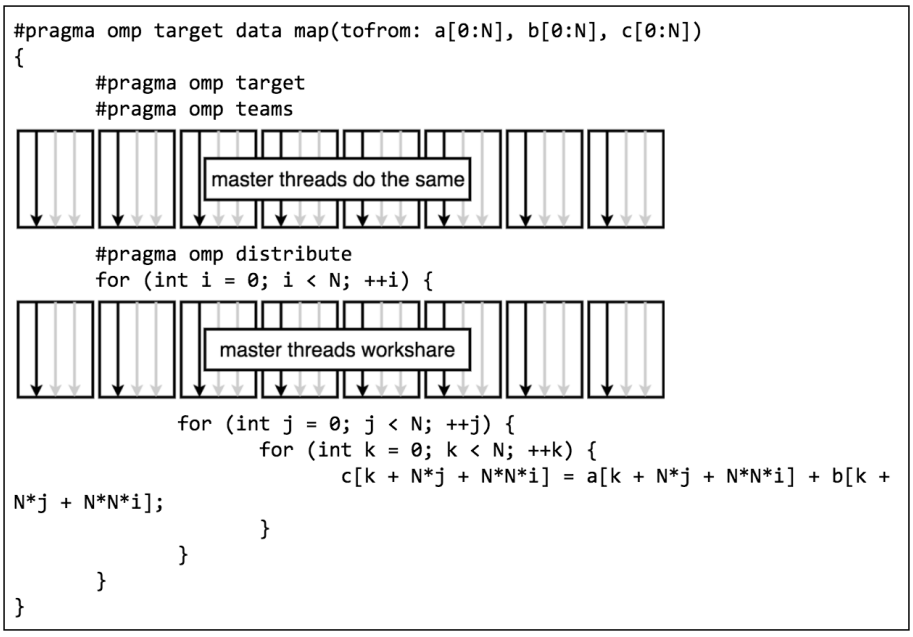

Figure 4.7: OpenMP 4.x teams distribute construct

The `parallel` loop construct contains an implicit barrier at the exit of its execution region, unless the `nowait` clause has been specified, there will be a synchronization for threads. If the `parallel` loop construct appears within the `target` region, it becomes a worksharing region for current thread team on the device. If there is no `teams` construct inside the `target` region, a single team will be created and the workload will be shared among team members.

4.3.5 SIMD Construct

In the definition of the SIMD execution model, the loop with simd hint denotes that it will be translated into a SIMD loop. Modern accelerators usually support vectorization operations using vector processing units at hardware level. The `simd` construct provides the vectorization capability in OpenMP which means multiple

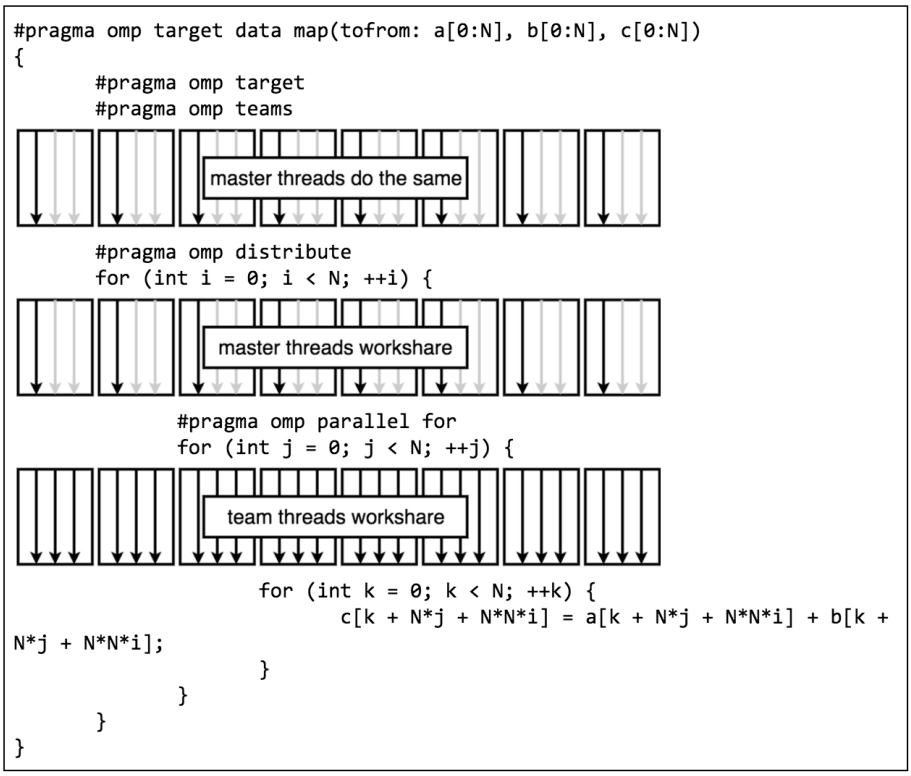

Figure 4.8: OpenMP 4.x target parallel loop construct

data elements will be processed simultaneously using a single hardware instruction. The SIMD instruction should follow the dependency restrictions among iterations and within each iteration.

In the hardware implementation of a vectorization unit, data elements are always stored in a vector type simd register. For example, a 128 bit vector can hold 16 bytes or 4 words or 2 double words depending on the data element type, a data array with 4 float type elements could be packed into this vector and the iterations of the loop will be mapped into the register. At any time during execution, the data in the simd vector will have the same execution pattern, which means that there is only one program counter shared by the data elements in that vector.

In order to load data more efficiently into vectorization units, SIMD may require memory alignment for better performance. The `simd` construct provides data alignment clause for users to specify alignment information for a certain hardware. Another important issue for simd processing is divergent branching, branching instructions will affect the efficiency of SIMD execution because vector elements should be decomposed and processed sequentially. When there is control flow within the binding loop of `simd` construct, an authenticity check will be performed to evaluate if the condition is true or false, then both true and false conditions are executed on the running SIMD lanes. After the calculation of vector elements, for each result that satisfy the condition evaluation, the outcome will be selected and stored in the resulting vector. In general, SIMD operations usually use masking operations for both conditions.

Figure 4.9 shows the complete three levels of parallelism, the innermost loop is vectorized using `simd` construct. The length of SIMD lane depends on the hardware implementation. Each thread of the teams will issue its SIMD instructions using thread local hardware resources.

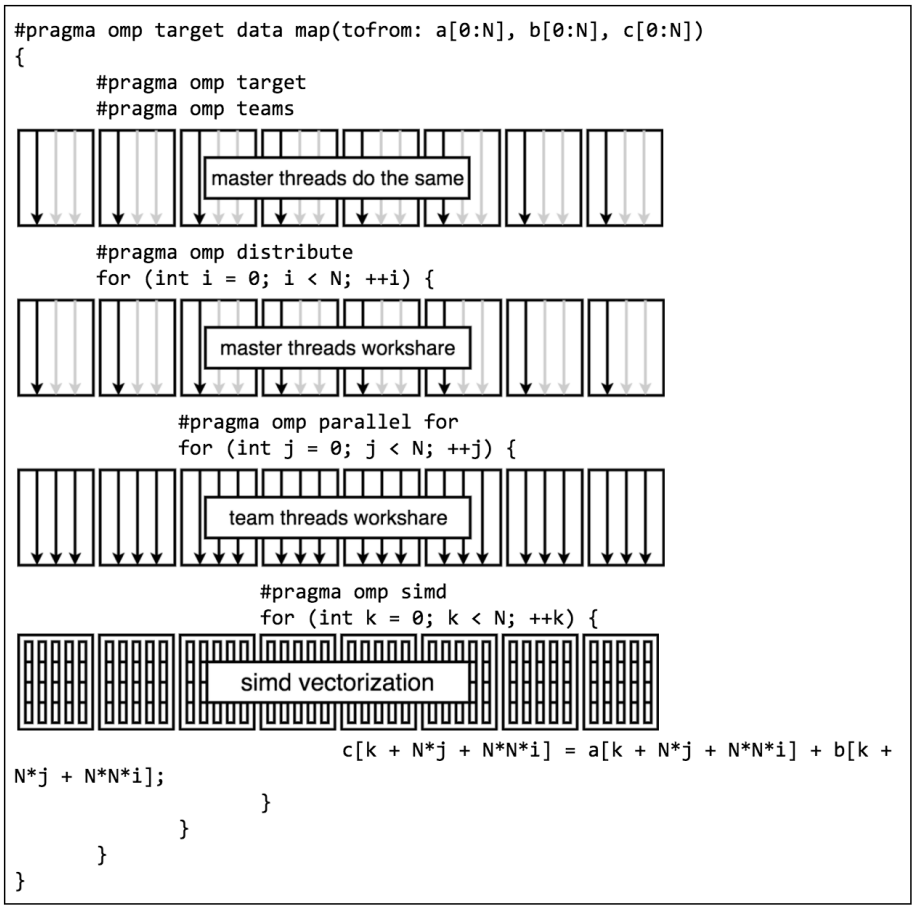

Figure 4.9: OpenMP 4.x simd construct

Chapter 5

OpenUH GPU Offloading Model

We have implemented the OpenMP accelerator constructs in the OpenUH compiler. Note that OpenUH also supports OpenACC, another directive-based API for device programming. The existing OpenACC implementation generates CUDA/OpenCL kernel functions for NVIDIA and AMD GPUs, respectively. In this work, we target NVIDIA GPUs because of their wide popularity in the HPC community. We plan to address AMD GPUs in the future work.

The creation of an OpenMP compiler for accelerators requires a significant implementation effort to meet the challenges of mapping high-level loop constructs to low-level threading architectures. This implementation is divided into frontend, backend and CUDA kernel generation phases.

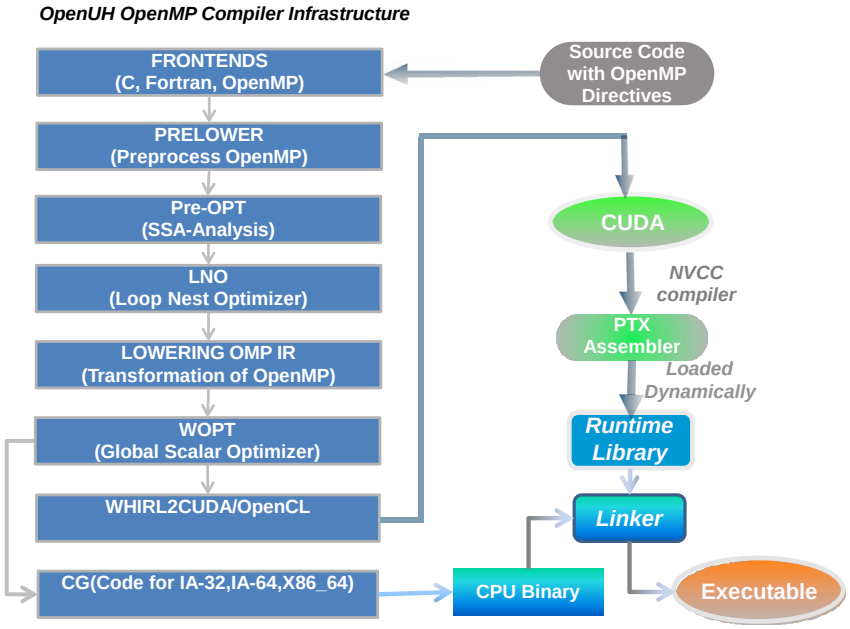

Figure 5.1: OpenUH compiler framework for OpenMP offloading constructs

5.1 Front-end Support

OpenUH compiler is using GNU GCC 4.2 front-end for parsing C/C++ syntax, the front-end will generate very high level WHIRL IR (Intermediate Representation). We have extended the existing front-end to support the following directives and clauses of OpenMP 4.5: `target`, `target data`, `target update`, and the loop parallelism directives, namely `distribute`, `teams` and `simd`. We also extended the parallel loop sharing construct for GPU threads. Those construct are parsed and a variety of new IRs are generated.

5.2 Back-end Support

In this phase, we transform the generated IR nodes into runtime library calls, which further invoke lower-level CUDA library routines to allocate/deallocate memory within GPUs; this corresponds to the LOWERING module in Figure 5.1. Then, we outline each OpenMP target region as a CUDA kernel function that will be fed to another module of Figure 5.1, called WHIRL2CUDA, in order to generate CUDA source code. The WHIRL2CUDA module is a modified module derived from WHIRL2C module which enables converting IR tree to corresponding C codes. Finally, we replace the original outlined target region with a runtime function call that is used to launch the kernel. Note that during the outlining step, we perform a one-to-one mapping of parallel loops to the threading architectures; we detail this mapping in the following chapters.

Since our translation is based on source-to-source translation to handle target regions, we use the CUDA SDK environment to translate the CUDA kernel functions into NVIDIA GPU assembly code, also called PTX code. The source-to-source translation plan make it more flexible to track each optimization phase in the back-end, it also extends the ability to adapt itself to new hardware architecture changes with minimum modifications.

The Pre-OPT phase and LNO phase contain a variety of optimizations for OpenUHs offloading backend. Live variable analysis [17] is applied and an extended version for the whole computer region is implemented in the back-end. For the data transferring constructs, this liveness analysis is used to assist variable allocation on the device.

The loop transformation strategy which converts associated loops into corresponding CUDA thread scheduling codes is included in the LNO phase. Our OpenMP 4.x implementation has already made use of those optimization strategies in the back-end.

5.3 Runtime Support

The compiler back-end will generate CUDA C kernel codes for each offloading region, and the host IR contains device function calls. The runtime system is responsible for launching device function calls as well as data transferring management. As we mentioned in the previous sections that the data management constructs is used to create device data environments, the runtime support is responsible for data mapping. As a source-to-source translation plan for OpenMP target model, the core data management routines in the runtime system is based on CUDA driver APIs with additional data handlers.

As shown in figure 5.2, at the entry of the data region, the device version of the data pointer referred inside the target region will be created and a hash map will be used to store the host-device pointer pairs. This hash map is called Data Present Table (DPT) [27]. The host pointer is used as the hash key of the table and the value represents the corresponding device pointer. The DPT is useful for data address retrieving on the device to reduce the extra data transferring cost. At the entry of each compute region, the runtime will search for the data pointers referred in the compute region. If the data pointer exists in the DPT, the runtime will use its

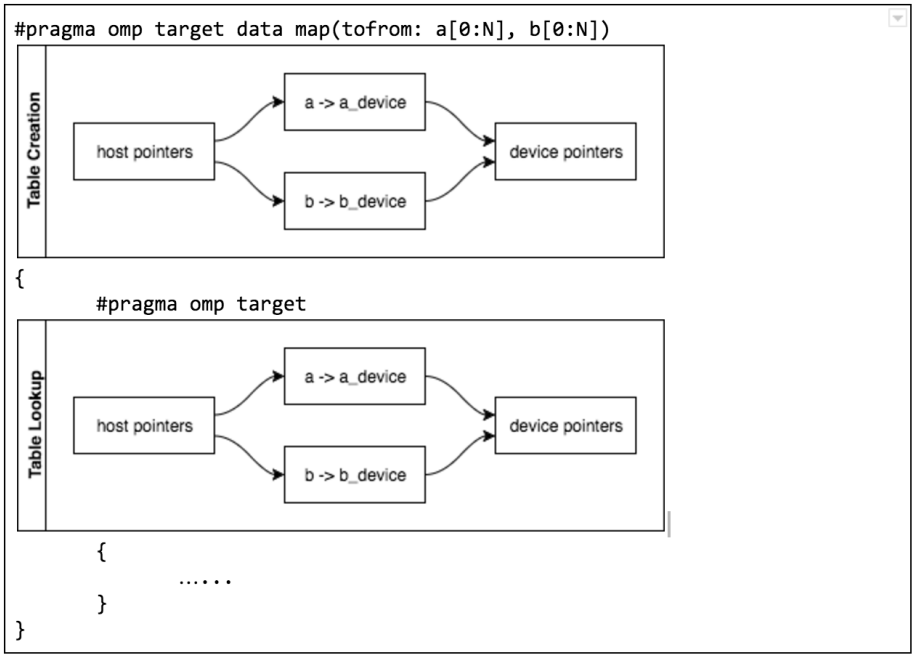

Figure 5.2: Runtime support for OpenMP target model of OpenUH

corresponding device pointer to handle the data. If the data pointer does not exist in the DPT, the device data will be allocated and initialized.

Chapter 6

One-to-One Parallelism Levels Mapping for OpenMP in OpenUH

OpenMP 4.x contains three levels of parallelism: teams, parallel for and simd. With the traditional CPU architecture and in CPU-like accelerators, the mapping of these three levels of parallelism is straightforward. In the latest Intel 72-Core Knights-Landing (KNL) accelerator, for example, each core has two 512-bit vector units and supports four threads. In this case, we might, e.g., create 72 teams containing 4 threads each. Each thread can then perform 512-bit SIMD operations. The typical GPU architecture does not permit this straightforward approach, and thus designing an efficient implementation of OpenMP 4.x on the GPU is not a trivial task. In this work, we introduce, and subsequently assess, an approach based on a one-to-one mapping of the OpenMP levels of parallelism to the levels of parallelism found in CUDA, i.e, grid, thread block, and warp [13].

6.1 Mapping Strategy Overview

Our main goal is to follow the OpenMP target model specification while attaining high performance on the GPUs as well as CPU-like accelerators despite the significant differences in their architectures. The individual cores in CPU-like accelerators contain SIMD units. Their ability to issue SIMD instructions makes it very easy for the cores to exploit the SIMD directives in OpenMP, which are important for performance on such platforms. In contrast, GPUs are highly suitable for massively parallel processing. Their thread hierarchy is flatter than on CPUs. The warps are basic scheduling units on GPUs and this makes it challenging to map different levels of parallelism.

Performance portability, where a single source code runs well across different target platforms, is increasingly demanded. Progress toward this goal in the context of OpenMP requires that accelerator directives be implemented efficiently on GPUs as well as CPU-like accelerators. In order to achieve this, it is necessary to effectively exploit each level of architectural parallelism, between and within groups of threads, and to map all three levels of OpenMP 4.x parallelism onto the GPU. Mapping the simd construct of OpenMP is critical to fully exploit the corresponding computing capabilities. Table 6.1 shows the mapping of OpenMP parallelism concepts to CUDA parallelism levels that was used in this work.

Table 6.1: OpenUH Loop Mapping in Terms of CUDA Terminologies

OpenMP Abstraction	CUDA Abstraction
teams	thread blocks within the grid
parallel	warps within a thread block
simd	32 threads within a warp

6.2 Mapping Three Levels of Parallelism

How to distribute loops among threads for thread blocks is an very important topic. Due to the hardware architecture differences and limitations, the compiler should be able to use an effective way to handle loop distribution, especially for programs with multiple levels of parallelism. The NVIDIA GPUs have a flat organization of CUDA cores in hardware level, the compute cores are organized as SMX (Streaming Multiprocessor). The grids, thread blocks and threads are representations of abstracted thread hierarchy from the hardware. Each grid can be composed of multidimensional thread blocks, each thread block can also be composed of multidimensional threads. The thread hierarchy in CUDA provides the possibility for mapping OpenMP device threads to CUDA threads. In OpenMP target model, each team contains multiple threads and each thread has vectorization capability using SIMD instructions.

The `teams` construct is translated directly into thread blocks in CUDA because the thread blocks are independent of one another and consist of a group of threads that run on the same execution unit (SMX). In the OpenMP `target` region, the environment of the compute kernel is initialized, which includes device data creation and CUDA kernel generation. At runtime, target data will be transferred between host and device according to the information from `map` clause, then the CUDA kernel will be launched.

When the program encounters the `teams` construct, a league of thread teams will be created. This mapping strategy will basically create a grid that consist of thread blocks; then the computation load will be taken by each thread block. Strictly

following the specification of OpenMP, the workload will be taken by the master threads of thread blocks. Note that all master threads of thread blocks will do the same calculation at this time, the workload will not be distributed across those master threads until the `distribute` construct has been specified. As each thread block represents a team in our implementation, there will be no synchronization among teams, the thread synchronization mechanism are only effective within a team.

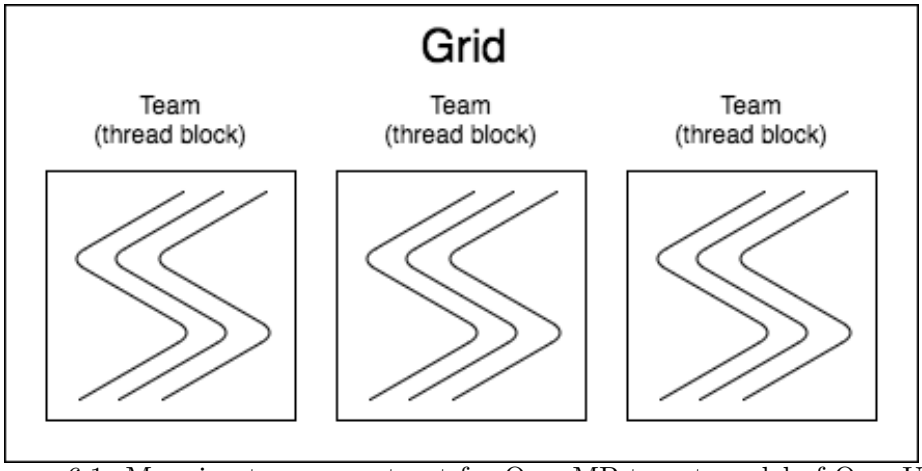

Figure 6.1: Mapping teams construct for OpenMP target model of OpenUH

In OpenUHs implementation, thread teams are configured as the x-dimension of the grid, each thread block represents the target thread team in OpenMP. There is only one grid created during every kernel launch.

The implementation of the `omp parallel for` directive inside the `omp target` region is based on a single thread block. A thread block will get a chunk of data and distribute the associated loop across threads. In OpenUH, the distribution is actually accomplished among warps within a thread block, and the master thread in each warp finishes the workload and waits at a synchronize point. This kind of mapping strategy makes it possible to utilize shared memory resources in each thread

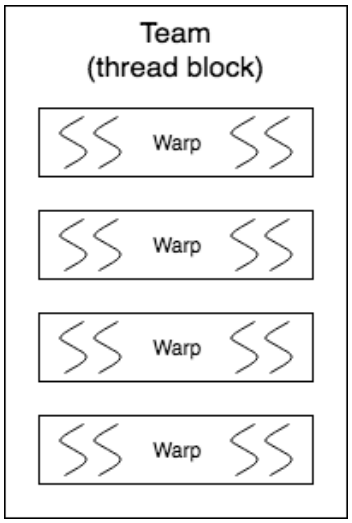

Figure 6.2: Mapping parallel loop construct for OpenMP target model of OpenUH

team on GPU-like accelerators.

Each warp consists of 32 threads; the first thread of each warp corresponds to the thread with the smallest thread-id. A parallel region is executed by each first thread in all the warps within the same thread block. If `for` is combined with `parallel`, these first threads will workshare the `parallel` loop region.

The GPU architecture supports the SIMT approach by executing the same instruction on several elements concurrently, within a warp. Therefore, a single instruction that is executed by a warp can be considered to be a SIMD operation as defined by OpenMP standard. Thus, in our implementation of OpenMP offloading for GPUs, we interpret the simd construct as the execution unit of a warp. Specifically, the iterations of a loop that is annotated by the simd directive are distributed or vectorized among threads within the same warp. The SIMT approach is much more flexible than SIMD. In addition to traditional SIMD operations, SIMT provides multiple addresses, registers and flow paths using a single instruction. In OpenUH,

the compiler backend is able to utilize some SIMT exclusive features to accomplish corresponding SIMD operations in programms.

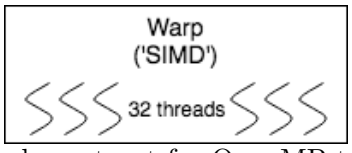

Figure 6.3: Mapping simd construct for OpenMP target model of OpenUH

When a loop has a `simd` hint, each thread in the warp may contain some thread private variables copied from outside the kernel region, those variables include pointers and scalar types, this situation results in redundant data items and a waste of register files. While for accelerators which support the exact definition of SIMD, the array and scalar pointers will be kept in different registers. The SIMD vector type register has the ability to accommodate a variety of data types, for example, a 256 bit length register can hold either 8 4 bytes integers or 4 8 bytes double precision floats. While for SIMT, each thread holds its own data elements and does the calculation regardless of the data length.

In a kernel region with simd hints, the compiler should generate codes for loading data to vector registers on CPU-like accelerators, and for loading data from device global memory to thread local registers on GPUs. In both cases, the program might need some random access of data elements. For SIMT architecture, the memory access happens at certain memory transaction cycles, random access will take more cycles for data loading operations. Although modern GPUs have shared memory for threads within a thread block to share resources, bank conflicts could be a common problem depending on applications. The SIMD model also needs extra instructional level support for random access of data elements.

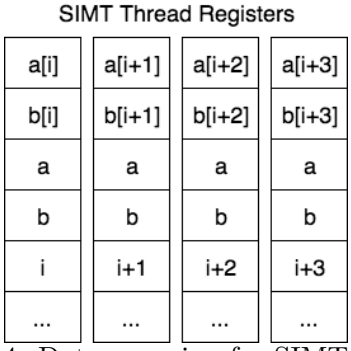

Figure 6.4: Data mapping for SIMT registers

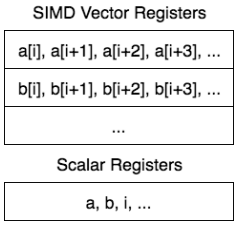

Figure 6.5: Data mapping for SIMD registers

The SIMT architecture on GPUs makes it more convenient to deal with control flows compared with SIMD model. This is especially convenient for our source-to-source translation plan. As we have described in the previous chapters that SIMD always use masking operations to deal with the branch divergence. OpenUH is using a similar approach to deal with control flows through invoking NVCC compiler. The SIMT implementation on NVIDIA Kepler architecture bundles 32 threads into a single warp, and the hardware level warp scheduler is used to issue several warps concurrently [22], thus different warps can not be guaranteed to be executed at the same time. If the control flows span multiple warps and each warp has a single flow path, there is no need to engage extra branching instructions since different warps can execute different instructions. However, this is not the case that the `simd` construct could be applied to this loop because the threads are not launched concurrently.

Placing a simd construct before a loop with control flow inside occurs frequently. In such scenario, warp divergence will happen. The warp divergence means that threads within a warp will go to different paths. The NVCC compiler invoked in OpenUH provides some prediction mechanism in some cases which the branching variable can be determined at compile time. In other cases, all threads will do all paths and then the correct results will be picked based on the branching variable. The branching cost using `simd` construct could be pretty high if each thread executes a particular path.

In OpenUH, the `parallel for` construct in the `target` region is mapped as the y-dimension threads in a thread block, the simd construct is mapped as the x-dimension threads in a thread block. Our compiler is using the coalesced memory access pattern to distribute loop iterations across threads. The coalesced access pattern make the consecutive threads have access to consecutive memory addresses, and during each memory transaction, those memory accesses will be bundled together [20].

Figure 6.6 is using a simple example to demonstrate how the memory coalescing is performed by OpenUH. The elements of the vector is stored in consecutive memory addresses, the GPU threads have been created in the compute kernel and each thread a unique thread ID. In the upper example of the figure, each thread is responsible to have access to a small group of consecutive memory addresses, in this case, memory access is not coalesced, the GPU hardware will typically use more memory transactions to load all elements. This situation could result in much lower memory bandwidth usage in GPU global memory. In the lower example of the figure, the first thread will load the first element of each sub-vector, the second thread will

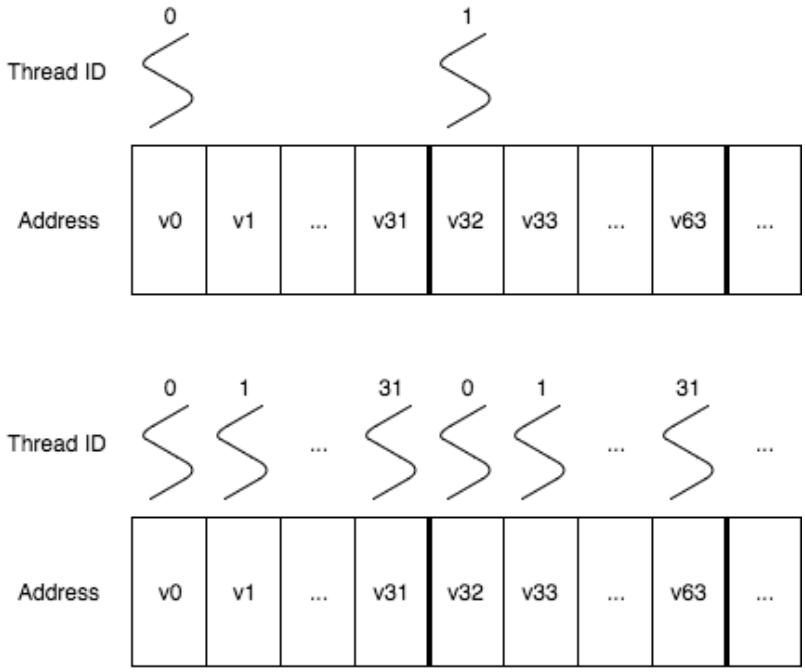

Figure 6.6: Memory coalescing for OpenMP target model of OpenUH

load the second element of each sub-vector and so on. In this case, each thread has a fixed step size for accessing memory addresses. Note that in some programs, inner loop interchange will be needed to help the OpenMP directives to enable memory coalescing.

The arrangement of GPU threads in OpenUH target model is based on this coalescing plan to better make use of GPU global memory bandwidth. We expect our loop transformation could cover most of the cases, some programs still need to be modified to adapt to memory coalescing. Since there are three levels of parallelism in OpenMP target model, there could be a combination of device constructs for each compute kernel, the step size for each thread to load data elements becomes more complex. The step size is determined by the current combination of constructs used in the compute kernel.

Table 6.2: Step size for GPU thread arrangement of OpenUH target model

Target Constructs	Step Size
teams	gridDim.x
parallel for	blockDim.y
simd	blockDim.x
teams + parallel for	gridDim.x * blockDim.y
parallel for + simd	gridDim.x * blockDim.x
teams + parallel for + simd	gridDim.x * blockDim.y * blockDim.x

Table 6.2 is using CUDA predefined variable to describe the step size for each combination of OpenMP target construct. Note that this thread arrangement plan is targeting on GPU-like accelerators, since the CPU-like accelerators have quite different memory accessing patterns in hardware level, this plan may not be as efficient as on GPUs.

6.3 Mapping Strategy Analysis

Figure 6.7 shows an example code that contains three levels of OpenMP offloading parallelism in a triple-nested loop.

We demonstrate the one-to-one mapping of this nested loop to the thread hierarchy. In this triple-nested loop, three thread blocks are generated and each thread block contains multiple warps, as we have the `simd` hint before the innermost loop, the computing can be distributed among threads inside warps with coalesced memory access model. This is a typical case that will fully utilize the three levels of parallelism of OpenMP target model, and it can also make full use of GPU threads resources. It is important that all threads within a warp are involved in the compute

```
#pragma omp target teams distribute
   for (i=1;i<=isize-1;i++)
   #pragma omp parallel for
      for (j=1;j<=jsize;j++)
      #pragma omp simd
         for (k=1;k<=ksize;k++)
            temp1 = dt * tx1;
            temp2 = dt * tx2;
            lhsX[i][j][k] = ...
```

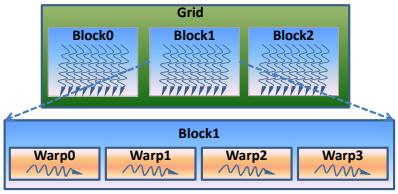

Figure 6.7: One-to-One Mapping of the Three Levels (top) of OpenMP Parallelism into a GPU (bottom)

task, otherwise there will be very high thread synchronization overhead which will harm performance.

Figure 6.8 shows an example code that contains two levels of OpenMP offloading parallelism in a double-nested loop. The innermost loop is at thread level and no further vectorization is required. For this code snippet, on a CPU-like accelerator, using `parallel for` is better than using `simd` for the inner loop because the inner loop control flow could be costly, SIMD on CPUs should identical instructions; if we use the `simd` construct instead of parallel for, the inner loop may be less efficient because of the divergent branching. CPU cores are typically much more powerful than GPU cores. On a GPU-like platform, using `parallel for` will make only one thread inside the warp in the computation which results in poor performance. On the programmers perspective, the same code should obtain desired performance on

different hardware platforms, this requires the compiler to optimize its implementation based on different accelerators types. In the OpenUH implementation, we have automatically attached the `simd` construct after the innermost loop to obtain reasonable performance. This transformation still spreads computation load across thread teams, but in a more efficient way. Since this transformation happens in the innermost loop, it is semantically legal. The performance of the code is heavily dependent on the target accelerator type. In OpenUH, the current compilation strategy was designed to take advantage of the warp-wise execution model, which assumes that the workload of the innermost loop always be spread across threads within warps.

```
#pragma omp target teams distribute
   for (i=1;i<=isize-1;i++)
   #pragma omp parallel for
      for (j=1;j<=jsize;j++)
         temp1 = dt * tx1;
         temp2 = dt * tx2;
         lhsX[i][j][k] = ...
         if ()
         else
```

Figure 6.8: One-to-One Mapping of the Two Levels of OpenMP Parallelism into a GPU

```
#pragma omp target
   #pragma omp parallel for
   for (i=1;i<=isize-1;i++)
       a[i] = ...
       if ()
       else
```

Figure 6.9: One-to-One Mapping of the One Level of OpenMP Parallelism into a GPU

Figure 6.9 shows an example code that contains just a single level of OpenMP offloading parallelism in a single loop. Since we do not have a `teams` construct, only one thread block is created and the workload is shared among the threads inside this

thread block. As described above, in a naive implementation only the master thread in each warp is working and the other threads simply wait at the synchronization point. There is a large overhead for synchronizing threads, which further degrades performance. In this kind of scenario, OpenUHs implementation will automatically spread the workload to warp threads which has an effect similar to that of using a combined parallel for `simd` construct. Note that this approach, and the previous insertion of `simd` constructs, does not affect the semantics of the program.

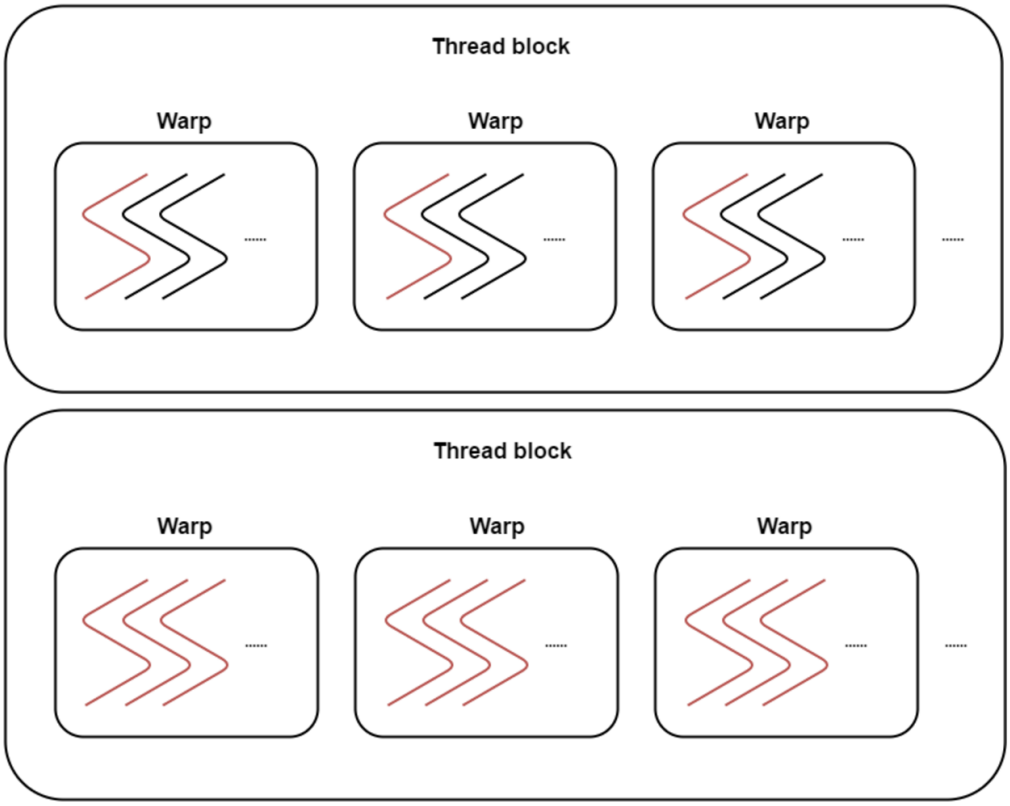

Figure 6.10: Improved warp-wise execution plan for OpenMP target model of OpenUH

Figure 6.10 provide an intuitive description of our mapping strategy for the inner-most loops. Threads marked red are master threads. The native scheme will let only the master threads involved in the execution, other threads will wait for the master

threads to finish and then be synchronized. Our improved plan will further share workloads among all threads within each warp. The hardware level GPU thread scheduler use warp as a basic scheduling unit, it is better to make full use of threads since the scheduling cost for an entire warp and a single thread of that warp are nearly identical.

Chapter 7

Performance Evaluation and Discussion

In this chapter, we present the results of experiments that used our implementation of the OpenMP 4.5 offloading model to translate the NAS parallel benchmarks (NPBs) for a GPU. We also compare our implementation with that of LLVM. We will first describe the test environments including the hardware platform and test kernels. Then we will compare the performance with LLVM. Finally, we will discuss about the performance issues based on the test results.

7.1 Test Environments

We will describe the hardware platform and give a brief introduction of the test codes.

7.1.1 Compilers and Test Beds

The NVIDIA GPU used for this evaluation includes a host with 2 Intel Xeon E5-2640 CPUs (16 threads per CPU) and 32GB main memory; the attached GPU is a K20XM with 5GB global memory. CUDA 6.5 is used for the OpenUH backend GPU code compilation with `-O3` optimization. For the comparative analysis, we used the Clang 3.8 implementation of OpenMP that is available at `https://github.com/clang-omp/libomptarget`. We use -fopenmp -omptargets=nxptx64sm_35-nvidia-linux -O3 for the LLVM compiler options. To obtain reliable results, all experiments were performed five times and then the average performance was computed. We also compare the GPU execution with 2 Intel Xeon E5-2640 CPUs.

7.1.2 NAS Parallel Benchmarks for OpenMP 4.x

In order to assess the performance of the OpenUH implementation of the OpenMP offloading model, we modified the NAS parallel benchmarks (NPB) [3] to make use of OpenMP offloading directives, including data movement and parallel region constructs with the corresponding clauses. The NAS parallel benchmarks contains several test kernels and some of the kernels come from computational fluid dynamics applications. Since those tests have been adjusted for large scale compute platforms consist of multi-core/many-core processors, the benchmarks result is valuable and comparable to real world applications.

The NAS benchmarks have an OpenACC version [29]. The OpenACC version has made some changes on the original codes to optimize the performance on GPUs. Our

modified OpenMP 4.x version NPB has adopted some of the optimization changes because our implementation is also targeting on GPUs, especially for the loop interchanges which help the directives to better utilize coalesced memory access pattern.

We used seven tests in NPB, for each test the modified version contains some compute kernels. We used two different data size options (Class B, Class C). The details of each test are described as follows: the BT test stands for Block Tridiagonal Solver and contains 46 compute kernels; the CG test stands for Conjugate Gradient and contains 17 compute kernels; the FT test stands for Fast Fourier Transform and contains 13 compute kernels; the EP test stands for Embarrassingly Parallel and contains 5 compute kernels; the LU test stands for Lower-Upper symmetric Gauss-Seidel and contains 56 kernels; the MG test stands for MultiGrid and contains 18 compute kernels; the SP test stands for Scalar Penta-diagonal solver and contains 65 compute kernels.

7.1.3 Common Compute Kernels

In order to measure the performance difference between OpenUH and LLVM, we used four small computer kernels: 2d-jacobi, matrix-matrix multiplication, gauss and laplacian. Those small kernels are more flexible and will make it easier to understand the performance gaps between different compiler implementations.

Figure 7.1 shows the code snippet of the 2d-jacobi kernel, it contains two nested loops. Note that this compute kernel is part of the whole jacobi kernel and will be launched multiple times, it is a good example for monitoring the kernel launch

```
#pragma omp target data map(to:temp_in[0:ni*nj], temp_out[0:ni*nj])
{
    #pragma omp target
    #pragma omp teams distribute
    for (j=1; j < nj-1; j++) {
        #pragma omp parallel for private(i00,im10,ip10,i0m1,i0p1,d2tdx2,d2tdy2)
        for (i=1; i < ni-1; i++) {
            i00 = I2D(ni, i, j);
            im10 = I2D(ni, i-1, j);
            ip10 = I2D(ni, i+1, j);
            i0m1 = I2D(ni, i, j-1);
            i0p1 = I2D(ni, i, j+1);

            d2tdx2 = temp_in[im10] - 2*temp_in[i00] + temp_in[ip10];
            d2tdy2 = temp_in[i0m1] - 2*temp_in[i00] + temp_in[i0p1];

            temp_out[i00] = temp_in[i00] + tfac*(d2tdx2 + d2tdy2);
        }
    }

}
```

Figure 7.1: Code snippet for 2d-jacobi kernel

overhead.

```
#pragma omp target data map(to: mat_a[0:dim*dim], mat_b[0:dim*dim]) map(tofrom: mat_c[0:dim*dim])
{
#pragma omp target
#pragma omp teams distribute
for (i = 0; i < dim; i++) {
#pragma omp parallel for
    for (j = 0; j < dim; j++) {
        mat_c[i * dim + j] = 0.0;
        for (k = 0; k < dim; k++) {
            mat_c[i * dim + j] += mat_a[i * dim + k] * mat_b[k * dim + j];
        }

    }
}
```

Figure 7.2: Code snippet for matrix-matrix multiplication kernel

The matrix-matrix multiplication kernel is shown in figure 7.2. There is dependency between the innermost loop and the outer loop, we can only use OpenMP target directives to parallelize the outer two loops. Each thread sharing the second loop need to calculate a complete loop.

The gauss kernel shown in figure 7.3 contains two nested loops. The inner loop does not contain any control flow but it has a lot of data loading operations. This

```
#pragma omp target data map(to: w0[0:szarray], w1[0:szarray])
{
  #pragma omp target
  {
    #pragma omp teams distribute private(j, i)
    for (j = 2; j < ny - 2; j++)
    {
        #pragma omp parallel for
        for (i = 2; i < nx - 2; i++)
        {
            _A(w1, i, j) = f * (
                s0 *  _A(w0, i,   j  ) +
                s1 * (_A(w0, i-1, j  ) + _A(w0, i+1, j  ) + _A(w0, i  , j-1) + _A(w0, i  , j+1)) +
                s2 * (_A(w0, i-1, j-1) + _A(w0, i+1, j-1) + _A(w0, i-1, j+1) + _A(w0, i+1, j+1)) +
                s4 * (_A(w0, i-2, j  ) + _A(w0, i+2, j  ) + _A(w0, i  , j-2) + _A(w0, i  , j+2)) +
                s5 * (_A(w0, i-2, j-1) + _A(w0, i-1, j-2) + _A(w0, i+1, j-2) + _A(w0, i+2, j-1)  +
                      _A(w0, i-2, j+1) + _A(w0, i-1, j+2) + _A(w0, i+1, j+2) + _A(w0, i+2, j+1)) +
                s8 * (_A(w0, i-2, j-2) + _A(w0, i+2, j-2) + _A(w0, i-2, j+2) + _A(w0, i+2, j+2)));
        }
    }
  }
}
```

Figure 7.3: Code snippet for gauss kernel

sample kernel could be used to measure the data loading efficiency on GPU threads.

```
#pragma omp target data map(to:w0[0:szarray], w1[0:szarray])
{
  #pragma omp target
  {
    #pragma omp teams distribute
    for (k = 1; k < ns - 1; k++)
    {
        #pragma omp parallel for
        for (j = 1; j < ny - 1; j++)
        {
            #pragma omp simd
            for (i = 1; i < nx - 1; i++)
            {
                _A(w1, i, j, k) = alpha * _A(w0, i, j, k) + beta * (

                    _A(w0, i+1, j, k) + _A(w0, i-1, j, k) +
                    _A(w0, i, j+1, k) + _A(w0, i, j-1, k) +
                    _A(w0, i, j, k+1) + _A(w0, i, j, k-1));
            }
        }
    }
  }
}
```

Figure 7.4: Code snippet for laplacian kernel

The laplacian kernel contains three levels of parallelism. It will fully utilize the three levels of OpenMP thread hierarchy. The implementation for OpenMP offloading on GPUs should be able to efficiently exploit thread resources. The `simd` construct should be mapped to the GPU threads in a way that benefit from GPU

hardware architecture.

7.2 Experimental Results and Analysis

7.2.1 Performance Comparison between GPU and CPU

We measured the execution time for data movement and for launching and running the kernels in NAS benchmarks. The data sizes used for the benchmark are CLASS B and CLASS C, the size of CLASS C is larger than CLASS B. The performance comparison is between CPU and GPU using OpenUH compiler. The execution time on CPU is based on our OpenMP 3.x implementation without GPU support. We enabled 16 threads on the CPU side.

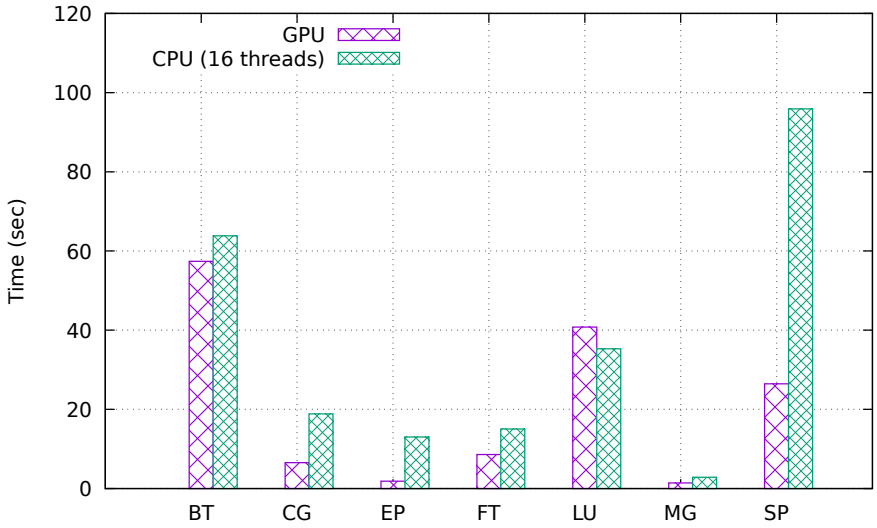

Figure 7.5: NPB-OpenMP offloading vs. NPB-OpenMP using OpenUH (Class B). BT denotes Block Tridiagonal Solver; CG denotes Conjugate Gradient; FT denotes Fourier Transform; EP denotes Embarrassingly Parallel; LU denotes Lower-Upper symmetric Gauss-Seidel; MG denotes Multi-Grid; SP denotes Scalar Penta-diagonal solver.

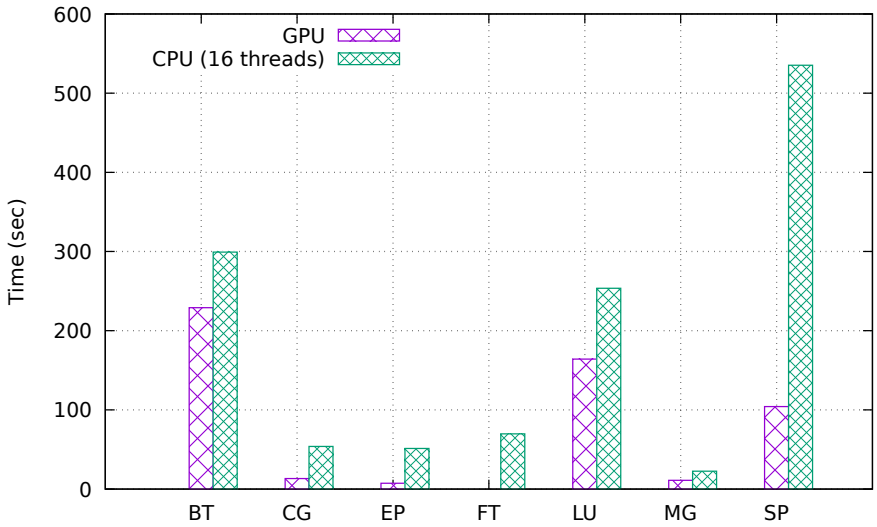

Figure 7.6: NPB-OpenMP offloading vs. NPB-OpenMP using OpenUH (Class C). BT denotes Block Tridiagonal Solver; CG denotes Conjugate Gradient; FT denotes Fourier Transform; EP denotes Embarrassingly Parallel; LU denotes Lower-Upper symmetric Gauss-Seidel; MG denotes Multi-Grid; SP denotes Scalar Penta-diagonal solver.

According to the test result, we can determine that the GPU version codes in general ran faster than the CPU version codes. However, we noticed that the LU kernel running on GPU using CLASS B data size has longer execution time than CPU, while for CLASS C, the execution time is smaller than CPU. Since the LU kernel contains a lot of data updating operations, the data transferring overhead is quite large, the data exchange cost dominates whole execution time for small data size. When the data size becomes larger, the relative performance difference between CPU and GPU becomes larger.

7.2.2 Performance Comparison between OpenUH and LLVM

We compared the performance of OpenUH with the LLVM implementation that used `libomptarget` as an offloading runtime. In all kernels, our implementation outperforms the LLVM implementation.

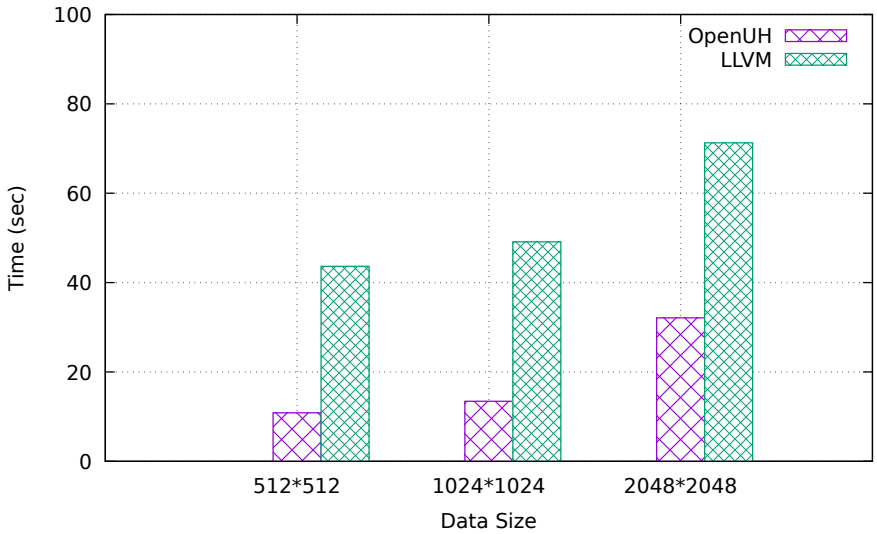

Figure 7.7: Performance comparison between OpenUH and LLVM on Jacobi-OpenMP4.x Kernel (using 20000 max-iterations)

For the matrix-matrix multiplication kernel, the performance gap between OpenUH and LLVM is huge (notice the logarithmic scale on the y axis) for very large data sizes. This is because Clang maps the loop hierarchy to the GPU threads in an unefficient way.

In fact, LLVM does not fully utilize GPU warps as we believe it should be handled while executing loops with a `parallel for` clause ahead. Indeed, LLVM mainly addresses two levels of parallelism on the GPU: `distribute` and `parallel`. In the current implementations [6, 5], the `teams` construct is mapped to the x dimension of

the thread blocks in the grid. The `parallel` directive is mapped to the x dimension of threads within each thread block. This indicates that LLVM does not handle the innermost loop efficiently; yet when only master threads in warps are executing, a large overhead for thread synchronization is incurred as described in the previous section.

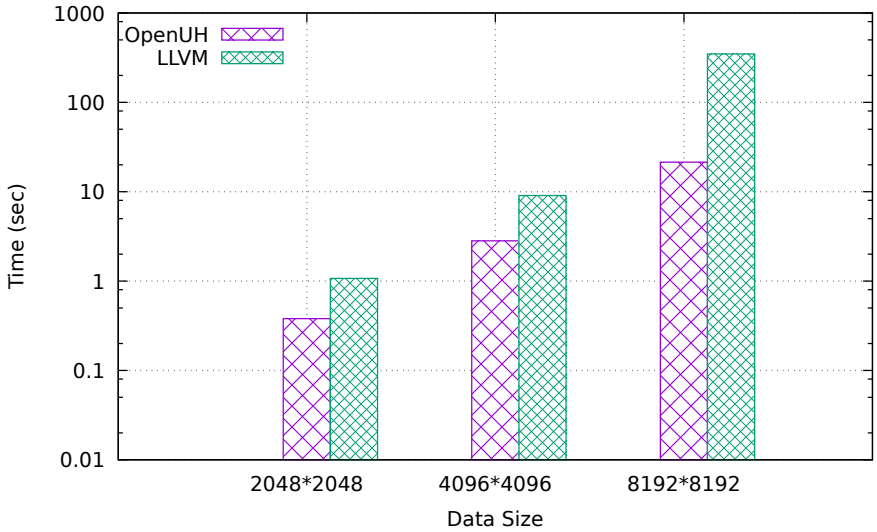

Figure 7.8: Performance Comparison between OpenUH and LLVM on Matrix Multiplication Kernel

This explains why in all tests with only two nested loops, OpenUH's performance is far better than LLVM, especially in matrix-matrix multiplication and gauss kernel. Both of the two kernels contains complex inner loops. The second loop of matrix-matrix multiplication kernel has a complete loop inside, even if the second loop has been shared by a bunch of GPU threads, each thread need to complete the entire innermost loop. If there is only one master thread working in a warp, the execution threads can not be bundled together, it will be a huge waste of GPU thread resources.

For the laplacian kernel, the performance gap between OpenUH and LLVM is

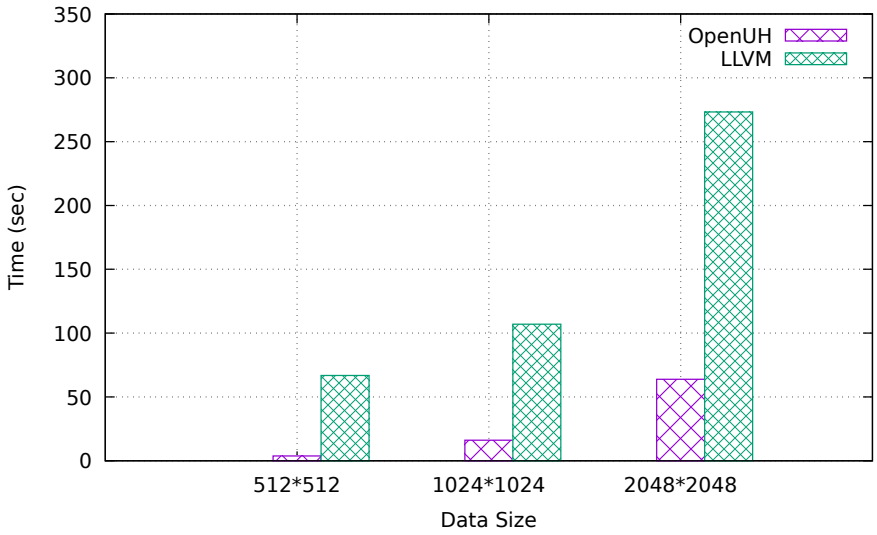

Figure 7.9: Performance Comparison between OpenUH and LLVM on Gauss Kernel

smaller than for the other three kernels. The laplacian has three levels of parallelism: `distribute`, `parallel` and `simd`, in this case, LLVM could make use of GPU warps for the innermost loop, which could then be spread across threads within each warp, resulting in much better performance. This example explains indirectly why making GPU warps work under full load is so important for offloading to GPU-like accelerators.

In OpenUH, the fact that we implicitly use GPU warps to handle innermost loop execution leads to a much better performance than LLVM's approach in general. Therefore, we believe the one-to-one mapping introduced in this paper should be used to fully exploit the thread hierarchy and to get better thread occupancy within GPUs.

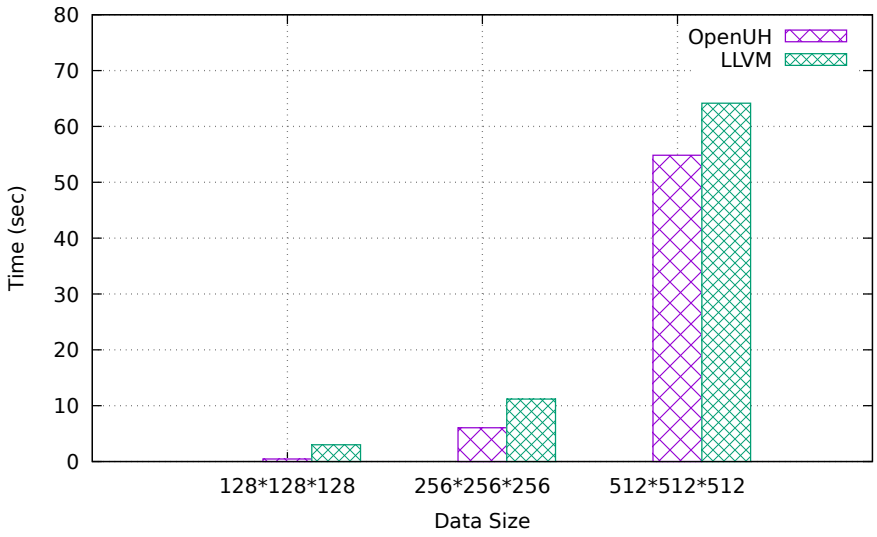

Figure 7.10: Performance Comparison between OpenUH and LLVM on Laplacian Kernel

Chapter 8

Conclusion

8.1 Conclusion

In this thesis, we describe our design and implementation of a compilation scheme based on a one-to-one mapping of the loop hierarchy parallelism available in OpenMP 4.x to the thread hierarchy found in GPUs. The implementation was carried out in the OpenUH compiler, which is well suited for prototyping novel techniques to support parallel languages and new compiler optimizations. The mapping strategy proposed in our work is valuable for offloading model for accelerators especially for GPU-like accelerators. The warp-based inner loop mapping plan was applied, we illustrated the efficient ways for inner loop transformations using OpenMP device directives. A set of benchmarks were employed to assess how this mapping, especially the use of GPU warps to handle innermost loop execution, impacts the performance

of GPU execution. The benchmarks include new versions of the NAS parallel benchmarks that we specifically developed for this research; we also used four commonly used kernels to help understand the performance gap between OpenUH and LLVM. We show that our one-to-one mapping technique significantly outperforms the LLVM implementation.

8.2 Future Work

We have built an usable and efficient implementation of OpenMP target model in OpenUH. There are several directions that is worth to extend the ability and improve the performance in our infrastructure:

- Modern GPUs contains some on-chip memory which is physically very close the compute cores and thus has high bandwidth and low latency. For example, the NVIDIA GPUs has the shared memory which can be shared among threads in each thread block. Those on-chip memory can be used to enchance the data locality optimization for GPU memory. However, the on-chip memory has very limited size. In order to utilize the limited resource for large data sets, it is important to design a robust and efficient method in OpenMP to make better use of the on-chip memory.

- In many real-world applications, deep copy appears very frequently. Users sometimes need to define their own data typed which may contain a complex form of data members. Those data types may contain dynamically allocated

arrays as well as array pointers which pointing to another type of structure. The shape of the complex data structure requires to be explicitly described in the corresponding directives. In order to use OpenMP directives to assist the memory transferring between host and device, a lot of work need to be done in the compiler and runtime.

- Nowadays, scientific applications always run on clusters which contains a number of compute nodes, each node also contains more than one GPU device. We should not targeting our impementation only on a single GPU. Multiple device support in OpenMP target model is needed to take advantage of multi-GPU systems. Data synchronization and distribution are an important issues that appear in porting applications into multi-GPU sytems. The communication overhead should be carefully studied. The modern NVIDIA GPUs have built-in multi-GPU support using CUDA. Our current infrastructure can be extended to incorporate this feature in our compiler framework.

Bibliography

[1] CUDA C PROGRAMMING GUIDE. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, September 2015.

[2] B. M. C. Ahmad Qawasmeh, Abid M. Malik. OpenMP Task Scheduling Analysis via OpenMP Runtime API and Tool Visualization. In *Parallel & Distributed Processing Symposium Workshops, 2014 IEEE International*. IEEE, 2014.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[4] J. Barker and J. Bowden. *Manycore Parallelism through OpenMP*, pages 45–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[5] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien. Integrating GPU Support for OpenMP Offloading Directives into Clang. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 5:1–5:11, New York, NY, USA, 2015. ACM.

[6] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave. Coordinating gpu threads for openmp 4.0 in llvm. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 12–21, Piscataway, NJ, USA, 2014. IEEE Press.

[7] B. C. Besar Wicaksono, Ramachandra C. Nanjegowda. A Dynamic Optimization Framework for OpenMP. In *7th International Workshop on OpenMP, IWOMP 2011, Chicago, IL, USA, June 13-15, 2011. Proceedings*. Springer, 2011.

[8] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski. OpenMP for Accelerators. In *OpenMP in the Petascale Era*, pages 108–121. Springer, 2011.

[9] O. A. R. Board. OpenMP 4.5 Specification. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`, 2015.

[10] B. Chapman, D. Eachempati, and O. Hernandez. Experiences Developing the OpenUH Compiler and Runtime Infrastructure. *International Journal of Parallel Programming*, pages 1–30, 2012.

[11] B. Chapman, O. Hernandez, L. Huang, T. hsiung Weng, Z. Liu, L. Adhianto, and Y. Wen. Dragon: an open64-based interactive program analysis tool for large applications. In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 792–796, Aug 2003.

[12] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[13] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[14] D. Eachempati, A. Richardson, T. Liao, H. Calandra, and B. Chapman. A Coarray Fortran Implementation to Support Data-Intensive Application Development. In *The International Workshop on Data-Intensive Scalable Computing Systems (DISCS), in conjunction with SC'12*, November 2012.

[15] P. Ghosh, Y. Yan, D. Eachempati, and B. Chapman. A Prototype Implementation of OpenMP Task Dependency Support. In A. Rendell, B. Chapman, and M. Mller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 128–140. Springer Berlin Heidelberg, 2013.

[16] L. Huang, H. Jin, L. Yi, and B. Chapman. Enabling locality-aware computations in openmp. *Sci. Program.*, 18(3-4):169–181, Aug. 2010.

[17] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[18] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[19] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.

[20] S. S. Mafi R. GPU-based acceleration of computations in nonlinear finite element deformation analysis. *International Journal for Numerical Methods in Biomedical Engineering*, 2014.

[21] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalán, M. González, and J. Labarta. Thread fork/join techniques for multi-level parallelism exploitation in numa multiprocessors. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 294–301, New York, NY, USA, 1999. ACM.

[22] NVIDIA. Kepler GK110 whitepaper, 2012.

[23] OpenACC. `http://www.openacc-standard.org`, 2015.

[24] OpenCL Standard. `http://www.khronos.org/opencl`, 2015.

[25] S. Pophale, O. Hernandez, S. Poole, and B. Chapman. Extending the openshmem analyzer to perform synchronization and multi-valued analysis. In S. Poole, O. Hernandez, and P. Shamis, editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, volume 8356 of *Lecture Notes in Computer Science*, pages 134–148. Springer International Publishing, 2014.

[26] C. Shen, X. Tian, D. Khaldi, and B. Chapman. Assessing one-to-one parallelism levels mapping for openmp offloading to gpus. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'17, pages 68–73, New York, NY, USA, 2017. ACM.

[27] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. *Compiling a High-Level Directive-Based Programming Model for GPGPUs*, pages 105–120. Springer International Publishing, Cham, 2014.

[28] Intel Xeon Phi Coprocessor Architecture Overview. `https://software.intel.com/sites/default/files/Intel%C2%AE_Xeon_Phi%E2%84%A2_Coprocessor_Architecture_Overview.pdf`, 2013.

[29] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman. *NAS Parallel Benchmarks for GPGPUs Using a Directive-Based Programming Model*, pages 67–81. Springer International Publishing, Cham, 2015.