OPTIMIZING THE PERFORMANCE OF DIRECTIVE-BASED PROGRAMMING MODEL FOR GPGPUS

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Rengan Xu

May 2016

OPTIMIZING THE PERFORMANCE OF DIRECTIVE-BASED PROGRAMMING MODEL FOR GPGPUS

Rengan Xu

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science at University of Houston

Dr. Christoph F. Eick
Dept. of Computer Science at University of Houston

Dr. Shishir Shah
Dept. of Computer Science at University of Houston

Dr. Jaspal Subhlok
Dept. of Computer Science at University of Houston

Dr. Henri Calandra
TOTAL E&P Research and Technology USA

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I would like to sincerely express my gratitude to my advisor, Dr. Barbara Chapman, for providing me an excellent research group to learn and improve. It is a great honor to work in her group to perform cutting edge research. I am grateful for her encouragement and support which inspired me new research ideas and provided me with opportunities to attend top tier conferences and workshops and interact with other researchers in this community. I would also like to thank my committee members, Dr. Henri Calandra, Dr. Christoph F. Eick, Dr. Shishir Shah, and Dr. Jaspal Subhlok, who spent their precious time to review my dissertation and provide valued comments.

My mentor Sunita Chandrasekaran helped me a lot for several projects she was in charge of. It would be impossible for me to complete those projects without her guide and support. I am also very grateful to other fellow members in HPCTools group: Tony Curtis, Yonghong Yan, Deepak Eachempati, Dounia Khaldi, Abid Malik, and Xiaonan Tian for their valuable discussion and help.

I had opportunities to do my internship in both Repsol and TOTAL. I appreciate Mauricio Araya from Repsol, Henri Calandra and Maxime Hugues from TOTAL for giving me generous help during my internships.

Last but not least, I cannot express my appreciation for endless love and support from my girl friend, my brother and my parents. Thank you for accompanying me all the time.

This research work is supported in part by the Office of the Assistant Secretary of Defense for Research and Engineering (OASD(R&E)) under agreement number

FA8750-15-2-0119. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Office of the Assistant Secretary of Defense for Research and Engineering (OASD(R&E)) or the U.S. Government.

OPTIMIZING THE PERFORMANCE OF DIRECTIVE-BASED PROGRAMMING MODEL FOR GPGPUS

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Rengan Xu

May 2016

Abstract

Accelerators have been deployed on most major HPC systems. They are considered to improve the performance of many applications. Accelerators such as GPUs have an immense potential in terms of high compute capacity but programming these devices is a challenge. OpenCL, CUDA and other vendor-specific models for accelerator programming definitely offer high performance, but these are low-level models that demand excellent programming skills; moreover, they are time consuming to write and debug. In order to simplify GPU programming, several directive-based programming models have been proposed, including HMPP, PGI accelerator model and OpenACC. OpenACC has now become established as the de facto standard. We evaluate and compare these models involving several scientific applications. To study the implementation challenges and the principles and techniques of directive-based models, we built an open source OpenACC compiler on top of a main stream compiler framework (OpenUH as a branch of Open64). In this dissertation, we present the required techniques to parallelize and optimize the applications ported with OpenACC programming model. We apply both user-level optimizations in the applications and compiler and runtime-driven optimizations. The compiler optimization focuses on the parallelization of reduction operations inside nested parallel loops. To fully utilize all GPU resources, we also extend the OpenACC model to support multiple GPUs in a single node. Our application porting experience also revealed the challenge of choosing good loop schedules. The default loop schedule chosen by the compiler may not produce the best performance, so the user has to manually try different loop schedules to improve the performance. To solve this issue, we

developed a locality-aware auto-tuning framework which is based on the proposed memory access cost model to help the compiler choose optimal loop schedules and guide the user to choose appropriate loop schedules.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Recent years have seen a rise of massively-parallel supercomputing systems that are based on heterogeneous architectures combining multi-core CPUs with General-Purpose Graphic Processing Units (GPGPUs). While such systems offer a promising performance with reasonable power consumption, programming accelerators in an efficient manner is still a challenge. The existing low-level APIs such as CUDA and OpenCL usually require users to be expert programmers and restructure the code largely. Optimized kernels are written that are usually coupled with specific devices. This leads to a less productive and more error prone software development process that is challenging to be adopted by the rapidly growing HPC market.

Recent approaches for programming accelerators include directive-based, high-level programming models for accelerators. It allows the users to insert non-executable pragmas and guide the compiler to handle low-level complexities of the system. The major advantage of the directive-based approach is that it offers a high-level programming abstraction thus simplifying the code maintenance and improving productivity.

As different directive-based programming models offer different feature sets, the code portability therefore becomes a major issue. As a joint standardization between CAPS, CRAY, PGI and NVIDIA, OpenACC was first released in November 2011, which aims to provide a directive-based portable programming model for accelerators. By using OpenACC, it allows the users to maintain a single code base that is compatible with various compilers, while on the other hand, the code is also portable across different possible types of platforms.

Our first attempt is to study the feasibility and applicability of these directive-based models when they are applied to scientific applications consisting of varied characteristics. These models allow programming without the need to explicitly manage the data transfer between CPU and GPU, device start-up and shut-down to name a few. We will explore three directive-based programming models, HMPP, PGI, and OpenACC and assess the models using scientific applications. We compare and contrast the performance achieved by these directives to that of the corresponding hand-written CUDA version of the application. Besides these mainstream directive-based models, there are some other research-based directive-based models including hiCUDA [35], CUDA-lite [66], Mint [67], and OpenMPC [48], but the compiler implementation for these models are not mature enough for evaluation.

Since OpenACC is a high-level directive-based model, the implementations of this model usually translates it to a low-level model such as CUDA or OpenCL. The low-level models like CUDA or OpenCL are language-based model, therefore it is flexible for the user to apply any optimization they want. But this requires the user to throughly understand the underlying architecture so that the applied optimizations can utilize the architecture efficiently. The high-level model OpenACC is directive-based, therefore it requires the compiler to apply those optimizations automatically. However, without enough information, the compiler is not able to do the optimizations as well as the user who is an expert in both the ported application and the architecture. Even though the compiler can apply some optimizations automatically, it may not achieve the expected speedup as the compiler does not have a full view of the whole application. Because of these reasons, the application ported with OpenACC and CUDA usually have a performance gap. The goal of our work is to apply both manual and automatic optimizations to optimize the performance of OpenACC programs.

To understand OpenACC model throughly, we start to use it to port large applications. We chose the NAS Parallel Benchmarks (NPB) which is a well recognized benchmark suite for evaluating current and emerging multi-core/many-core hardware architectures. We present a set of optimization techniques to tune the application performance. Besides those optimization techniques applied in the application level, we also present the optimizations in compiler and runtime level. The optimizations we focus on include the runtime library, reduction algorithm and multi-GPU. Finally, we focus on the loop scheduling auto-tuning optimization which aims to find

the optimal mapping from a nested loop to the GPU threads hierarchy.

1.2 Contributions and Dissertation Outline

In summary, to make the directive-based programming model for GPUs more mature and broaden its impact in both academia and industry, several research issues are addressed in this dissertation. Chapter 2 provides the background of the research work in this dissertation. The contributions of this dissertation are as follows:

- In Chapter 3, we compare the features of different high-level directive-based programming models for GPUs and the performance of applications using these models.

- In Chapter 4, to understand the root cause of the performance of directive-based model, we use NAS Parallel Benchmark (NPB) suite as example and present a set of optimizations to tune the application performance. Most of these optimizations are applied manually in the application level. We analyze a number of choices and combinations of optimization techniques and study their impact on application performance. We learn that poorly selected options or using system default options for optimizations may lead to significant performance degradation. We also compare the performance of OpenACC NPB with that of the well-tuned OpenCL and CUDA versions of the benchmarks to present the reasoning behind the performance gap.

- In Chapter 5, we present the optimizations applied automatically in the compiler and runtime for OpenACC applications. We demonstrate how the runtime tracks and manages all the data correctly and efficiently. We also propose and demonstrate new compiler algorithms to parallelize the comprehensive reduction operations within three levels of parallelism. Our implementation covers all possible reduction cases, reduction types and operand data types.

- In Chapter 6, we enable multi-GPU programming on single node by using two approaches. First we explore and evaluate the OpenMP & OpenACC hybrid model. Second, based on the disadvantages of the hybrid model approach, we propose a set of new directives to extend the OpenACC model to support multiple GPUs.

- In Chapter 7, we present a locality-aware auto-tuning framework for loop scheduling that uses an analytical model to find the optimal loop schedule which maps a nested loop to the GPU threads hierarchy. The framework extends the reuse distance model for GPU cache modeling.

In this dissertation, the related work is not in a separate chapter, it is discussed in each chapter. Chapter 8 concludes this dissertation.

Chapter 2

Background

In this chapter, we give the background of our research work. First, we give the overview of the GPU architecture which is the platform our research based on. Second, we give an overview of the directive-based programming models for GPUs, including Hybrid Multicore Parallel Programming workbench (HMPP) [1], PGI accelerator model [30] and OpenACC [10] which is the standard of high-level directive-based programming model for accelerators. Third, since we use OpenMP & OpenACC hybrid model in the multi-GPU research work, we also discuss the OpenMP programming model which is the de facto standard of shared-memory system programming.

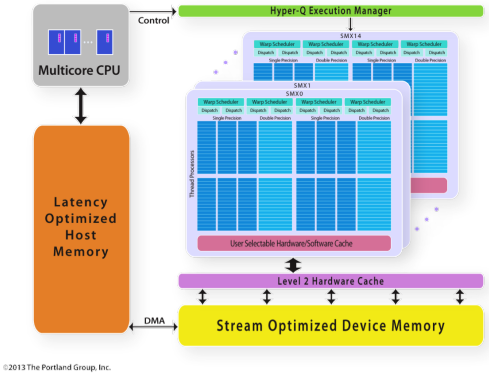


Figure 2.1: Nvidia Kepler GPU architecture[1]

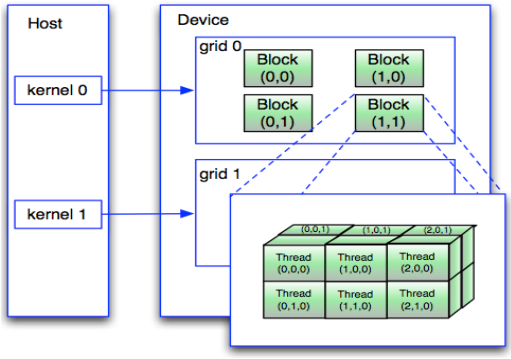


Figure 2.2: GPU threads hierarchy

2.1 Overview of GPU Architecture

GPU architectures differ significantly from that of traditional processors. We illustrate this using Nvidia Kepler GPU which is shown in Figure 2.1. Employing a Single Instruction Multiple Threads (SIMT) architecture, NVIDIA GPUs have hundreds of cores that can process thousands of software threads simultaneously. GPUs organize both hardware cores and software threads into two-level of parallelism. Hardware cores are organized into an array of Streaming Multiprocessors (SMs), each SM consisting of a number of cores named as Scalar Processors (SPs). An execution of a computational kernel, e.g. CUDA kernel, will launch a (software) thread grid. As shown in Figure 2.2, a grid consists of multiple thread blocks whose shape could be 1D, 2D, or 3D. And each thread block consists of multiple threads whose shape could also be 1D, 2D, or 3D. The GPU is connected to the CPU by a PCI-E bus. We also refer ***host*** to CPU and ***device*** to GPU. Since the host and the device are

[1]https://www.pgroup.com/lit/articles/insider/v2n1a5.htm

separate, in the GPU execution model, the CPU needs to first move the input data from CPU to GPU, then perform the computation within GPU, finally move the output result back to CPU. When CPU moves data to GPU, it can only move to the GPU DRAM as it is the only memory that can communicate with CPU. DRAM is the largest (usually GB) but also the slowest memory in GPU. Like CPU, GPU also has caches. Each SM has its own L1 cache (e.g. 16KB) and shared memory (e.g. 48KB) which are only accessible for the threads in this SM. All SMs share a unified L2 cache (e.g. 1.5MB). The shared memory is a user programmable cache while L1 and L2 are cached automatically by the GPU system. Each GPU thread has its own registers.

For programmers, the challenges to efficiently utilize the massive parallel capabilities of GPUs are to map the algorithms onto thread hierarchy, and to lay out data on both the global memory and shared memory to maximize coalesced memory access for the threads. Using low-level programming models such as CUDA and OpenCL to do this has been known as not only time consuming but also the software created are not identical to its original algorithms significantly decreasing code readability.

Table 2.1: Major features comparison among different models

Features		OpenACC	HMPP	PGI
Data	memory allocation	acc_malloc	allocate	acc_malloc
	memory free	acc_free	release	acc_free
	data movement	copyin copyout copy	args[*].io= in,out,inout	copyin copyout copy
	synchronization	update device update host	advancedload delegatedstore	update device update host
Computation	kernels offloading	parallel kernels	codelet&callsite	region
	loop scheduling	gang worker vector	gridify	parallel vector
	loop optimization	collapse tile auto	permute distribute fuse,unroll jam,tile	unroll

2.2 Overview of High-level Directive-based Programming Models

In this section, we provide details about three directive-based models that are being evaluated in this dissertation. The major features among these models are summarized in Table 2.1.

2.2.1 Hybrid Multicore Parallel Programming workbench (HMPP)

HMPP is a directive-based programming model used to build parallel applications running on manycore systems. It is a source-to-source compiler that can translate directive-associated functions or code portions into CUDA or OpenCL kernels.

In HMPP, the two most important concepts are "*codelet*" and "*callsite*" [1]. The "*codelet*" concept represents the function that will be offloaded to the accelerator, and "*callsite*" is the place to call the "*codelet*". It is the programmer's responsibility to annotate the code by identifying the codelets and inform the compiler about the codelets and where to call the same. In the steps of compilation, the annotated code is parsed by the HMPP preprocessor to extract the codelets and to translate the directives into runtime calls. The preprocessed code is then compiled and linked to HMPP runtime with a general-purpose host compiler. If the accelerator is not found or not available, the program execution can fall back to the original sequential version. HMPP also supports the "*region*" directive which only offloads part of a function into the accelerator and the "*region*" is a merge of *codelet*/*callsite* directives. The main issue with programming accelerators is the data transfer between the accelerator and the host. HMPP offers many data transfer policies as part of the optimization strategies. The user can manually control the data transfer, i.e. transfer the data every time the codelet is called or transfer the data only during the first time when the codelet is called. It can also be automatically decided by the compiler.

HMPP also provides a set of directives to improve the performance by enhancing the code generation. In the codelet, the user can put the read-only data into constant memory, preload the frequently-used data into shared memory, or explicitly specify the grid size in NVIDIA architecture. If the loop is so complex that the compiler is not able to parse, the user can give some hints to the compiler that all iterations in the loop are independent.

2.2.2 PGI Accelerator Model

PGI accelerator programming model contains a set of directives, runtime-library routines and environment variables [30]. The directives include data directives, compute directives and loop directives. The compute directive specifies a portion of the program to be offloaded to the accelerator. There is an implicit data region surrounding the compute region, which means data will be transferred from the host to the accelerator before the compute region and be transferred back from the accelerator to the host at the exit of compute region. Data directives allow the programmer to manually control where to transfer the data other than the boundaries of compute region. The loop directives enable the programmer to control how to map loop parallelism in a fine-grained manner. The user can add these directives incrementally so that the original code structure is preserved. The compiler maps loop parallelism onto the hardware parallelism using the *planner* [71]. PGI optimizes the data transfer by "*data region*" directive and its clauses and be able to remove unnecessary data copies. Using the loop scheduling directive, the user can add the data in the highest level of the data cache by using "*cache*" clause and this helps in improving the data-access speed.

2.2.3 OpenACC Programming Model

OpenACC is an emerging GPU-based programming model that is working towards establishing a standard in directive-based accelerator programming. As a joint standardization between CAPS, CRAY, PGI, and NVIDIA, OpenACC was first released

in November 2011, which aims to provide a directive-based portable programming model for accelerators. By using OpenACC, it allows the users to maintain a single code base that is compatible with various compilers, while on the other hand, the code is also portable across different possible types of platforms.

OpenACC is based on the use of pragmas or directives that allow the application developers to mark regions of code for acceleration in a vendor-neutral manner. It builds on top of prior efforts by several vendors (notably PGI and CAPS Enterprise) to provide parallel-programming interface for heterogeneous systems, with a particular emphasis on platforms that are comprised of multicore processors as well as GPUs. Among others, OpenACC is intended for use on the nodes of large-scale platforms such as the Titan system at ORNL, where CPUs and NVIDIA GPUs are used in concert to solve some of the nations most urgent scientific problems.

The OpenACC model is based on the PGI model, hence the former inherits most of the concepts from the latter. However some of the differences are: unlike PGI's single "*region*" compute directive, OpenACC offers two types of compute directives "*parallel*" and "*kernels*". The directive "*kernels*" is similar to PGI's "*region*" that surrounds the loops to execute on the accelerator device. With the "*parallel*" directive, however, if there is any loop inside the following code block and the user does not specify any loop scheduling technique, all the threads will execute the full loop. OpenACC supports three levels parallelism: gang, worker and vector, while PGI only defines two levels of parallelism: parallel and vector. Both OpenACC and PGI models allow the compute region to use the "*async*" clause to execute asynchronously with the host computation and the user can synchronize these asynchronous activities

with the "*wait*" directive. They also have a similar set of runtime library routines, including getting the total number of accelerators available, getting & setting the device type and number, checking & synchronizing the asynchronous activities and starting up & shutting down the accelerator. Unlike PGI, OpenACC can allocate and free a part of accelerator memory using *acc_malloc()*and *acc_free()* functions.

The OpenACC feature set includes pragmas, or directives, that can be used in conjunction with C, C++, and Fortran code to program accelerator boards. OpenACC can work with OpenMP to provide a portable programming interface that addresses the parallelism in a shared memory multicore system as well as accelerators. A key element of the interface is the parallel construct that launches gangs that will execute in parallel. Each of the gangs may support multiple workers that execute vector or SIMD constructs. A variety of clauses are provided that enables conditional execution, controls the number of threads, specifies the scope of the data accessed in the accelerator parallel region, and determines if the host CPU should wait for the region to complete before proceeding with other work.

Suitable placement of data and careful management of required data transfer between host and accelerator is critical for application performance on the emerging heterogeneous platforms. Accordingly, there are a variety of features in OpenACC that enables the application developer to allocate data and determine whether data needs to be transferred between the configured devices. The features also enable control this transfer, including the values to be updated on the host/accelerator by copying current data values on the accelerator/host, respectively. These features are complemented by a set of library routines to obtain device information or set device

types, test for completion of asynchronous activities, as well as a few environment variables to identify the devices that will be used.

OpenACC standard gives great flexibility to the compiler implementation. For instance, different compilers can have different interpretation of OpenACC three level parallelisms: coarse grain parallelism "gang", fine grain parallelism "worker", and vector parallelism "vector". On an NVIDIA GPU, PGI maps each gang to a thread block, and vector to threads in a block and it just ignores worker; CAPS maps gang to the x-dimension of a grid block, worker to the y-dimension of a thread block, and vector to the x-dimension of a thread block; Cray maps each gang to a thread block, worker to warp, and vector to SIMT group of threads.

2.2.4 OpenMP Programming Model

OpenMP is a high-level directive-based programming model for shared memory multi-core platforms. The model consists of a set of directives, runtime library routines, and environment variables. The user just needs to simply insert the directives into the existing sequential code, with minor changes or no changes to the code. OpenMP adopts the fork-join model. The model begins with an initial main thread, then a team of threads will be forked when the program encounters a parallel construct, and all other threads will join the main thread at the end of the parallel construct. In the parallel region, each thread has its own private variables and does the work on its own piece of data. The communication between different

threads is performed via shared variables. In the case of a data race condition, different threads will update the shared variable atomically. Starting from 3.0, OpenMP introduced *task* concept [17] that can effectively express and solve the irregular parallelism problems such as unbounded loops and recursive algorithms. To make the task implementation efficient, the runtime needs to consider the task creation, task scheduling, task switching, and task synchronization, etc. OpenMP 4.0 released in 2013 includes support for accelerators.

2.3 Overview of Low-level Language-based Programming Models

2.3.1 CUDA

CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and a language-based programming model specific for NVIDIA GPUs. A CUDA program is a unified source code including both host and device code. The host code is pure ANSI C code and the device code is the extension of ANSI C which provides some keyword for labeling data parallel functions called *kernels*. The host code and device code are compiled separately. The host code is compiled by the host's standard C compiler and the device code is compiled by Nvidia compiler nvcc. The host code is executed on the host and offloads the device code to be executed on the device. CUDA provides both a low level driver API and a high level runtime API. The programmer can use these APIs to manage the execution context environment,

the device memory allocation and deallocation, the data movement between CPU and GPU, the asynchronous data movement and kernel execution, etc. Since CUDA is proprietary to NVIDIA, hence the performance of CUDA program is optimized for NVIDIA GPU, but the disadvantage is that the same program cannot be migrated to other vendor's GPUs which has less portability.

2.3.2 OpenCL

OpenCL, which stands for Open Computing Language, is a language-based programming model for heterogeneous platforms including CPUs, GPUs, digital-signal processors (DSPs) and field-programmable gate arrays (FPGAs), etc. OpenCL is an extension of ISO C99 and provides an API to control the platform and program execution on the compute devices. Since OpenCL is an open standard, the benefit of OpenCL is cross-vendor and cross-platform software portability. Unlike CUDA model that has only one implementation in NVIDIA, OpenCL has implementations in multiple vendors such as AMD, Apple, IBM, Intel, Nvidia, Qualcomm, and Samsung.

2.4 Summary

In this chapter, we give an overview of the GPU architecture and both the high-level directive-based programming models and low-level language-based programming models for GPUs. The high-level models are able to simplify the GPU programming but at the cost of performance loss because it is not as flexible as the low-level models. The low-level models allow the users to take advantage of the hardware features by communicating to the driver and runtime directly. Therefore they are more flexible for the users and provide more opportunities to manually tune applications, but at the cost of steep learning curve and time-consuming development. The following chapters try to evaluate the performance gap between the directive-based model OpenACC and the low-level model CUDA and apply optimizations within OpenACC to decrease the performance gap. Some research issue like the loop scheduling tuning that exists in both OpenACC and CUDA will also be addressed.

Chapter 3

Experimental Analysis of Porting Applications to GPU

In this chapter, we assess and evaluate three directive-based programming models, HMPP, PGI, and OpenACC with respect to their run-time performance, program complexity, and ease of use. We compare the performance obtained of the three programming models with that of the native CUDA and the sequential version. The OpenACC model is very attractive as it standardizes programming. To the best of our knowledge, this is one of the very few work that systematically evaluates the models. This work is one of the first to compare OpenACC with other programming models. We have explored two vendor compilers supporting the OpenACC model. To maintain confidentiality we will be referring to the 2 vendor compilers in this chapter as OpenACC_Compiler_A and OpenACC_Compiler_B.

In this section, we evaluate three directive-based programming models using three

scientific applications. The experimental platform is a server machine that is a multicore system consisting of two NVIDIA C2075 GPUs. Configuration details are shown in Table 3.1. We use the most recent versions for all the compilers being discussed in this chapter. We use GCC 4.4.7 for all the sequential versions of the programs as well as for the HMPP host compiler. We use -O3 as the compilation flag for optimization purposes. As part of the evaluation process, we highlight several features of the programming models, that is best suited for the characteristics of an application. We compare the performances achieved by each of the models with that of the sequential and CUDA versions of the code. We consider wall-clock time as the evaluation measurement.

Table 3.1: Specification of experiment machine

Item	Description
Architecture	Intel Xeon x86_64
CPU socket	2
Core(s) per socket	8
CPU frequency	2.27GHz
Main memory	32GB
GPU Model	Tesla C2075
GPU cores	448
GPU clock rate	1.15GHz
GPU global memory	5375MB

3.1 2D-Heat Equation (Stencil domain)

The formula to represent 2D-heat equation is explained in [59] and is given as follows:

$$\frac{\partial T}{\partial t} = \alpha(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2})$$

where T is temperature, t is time, α is the thermal diffusivity, and x and y are points in a grid. To solve this problem, one possible finite difference approximation is:

$$\frac{\Delta T}{\Delta t} = \alpha[\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}]$$

where ΔT is the temperature change over time Δt and i, j are indices in a grid. At the beginning, there is a grid that has boundary points with initial temperature and the inner points that need to update their temperature. Then each inner point updates its temperature by using the previous temperature of its neighboring points and itself. The temperature updating operation for all inner points in a grid needs to last long enough which means many iterations is required to get the final stable temperatures. In our program, the number of iterations is 20000, and we increase the grid size gradually from 128*128 to 4096*4096. Figure 3.1 contains the code for temperature updating kernel and the steps to call this kernel in the main program.

It is possible to parallelize the 2D heat equation using the most basic directives from OpenACC, HMPP, and PGI. For example, we use the OpenACC model and insert "*#pragma acc kernels loop independent*" before the nested loop inside the temperature updating kernel. The main point to note with this algorithm is performance. By profiling the basic implementation, we found that the data is transferred back and forth in every main iteration step. The cost of data transfer is so expensive that the parallelized code is even slower than the original native version. The challenging task is executing the pointer-swapping operation. In iteration i, *temp1* is the input and *temp2* stores the output data. Before proceeding to iteration $i + 1$, these two pointers are swapped so that *temp1* holds the output data in iteration i while *temp2* will wait to store the output data in iteration $i + 1$. An intermediate pointer *temp*

is needed to swap these two pointers. Since *temp* resides on the host while *temp1* and *temp2* reside on the accelerator, they cannot be swapped directly. The key is how to swap these pointers inside the accelerator internally so that unnecessary data transfer is removed.

While using the OpenACC model, we use the *deviceptr* data clause to specify *temp* as a device pointer, i.e, the pointer will always remain on the accelerator side. To avoid transferring the data during each step, *data* directive needs to be added so that the data is transferred only before and after the main loops. After all the iterations are completed we need to transfer data in *temp1* instead of *temp2* from GPU to CPU since the pointers of *temp1* and *temp2* have been swapped earlier. Inside the temperature-updating kernel, the nested loop is collapsed because every iteration is independent. Using HMPP, we have considered copying the input and output data just once by setting the data transfer policy of *temp_in* and *temp_out* to *manual*. To ensure that the correct data is being accessed, we used HMPP's data mirroring directive so that we refer to arguments *temp1* and *temp2* with their host addresses. Data mirroring requires data mirrors to be declared and allocated before being used. HMPP could collapse the nested loop in the kernel by using "*gridify(j,i)*" code generation directive. Figure 3.1 and Figure 3.2 show the code snippet using both HMPP and OpenACC model.

Note that this application is sensitive to floating-point operations. We found that the precision of the floating-point values of the final output temperature on the GPU is different to the values on the CPU. This is due to Fused Multiply-Add (FMA) [70], here the computation rn(X * Y + Z) occurs in a single step and rounded

```
#pragma hmpp heat codelet, target=CUDA &
#pragma hmpp & , args[temp_in].io=in &
#pragma hmpp & , args[temp_out].io=inout &
#pragma hmpp & , args[temp_in,temp_out].mirror &
#pragma hmpp & , args[temp_in, temp_out].transfer=manual
void step_kernel(int ni, int nj, float tfac,
                 float *temp_in, float *temp_out) {

    // loop over all points in domain (except boundary)
    #pragma hmppcg gridify(j,i)
    for (j=1; j < nj-1; j++) {
        for (i=1; i < ni-1; i++) {
            // find indices into linear memory
            // for central point and neighbours
            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);

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

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

 #pragma hmpp heat allocate, data["temp1"], size={ni*nj}
 #pragma hmpp heat advancedload, data["temp1"]

 #pragma hmpp heat allocate, data["temp2"], size={ni*nj}
 #pragma hmpp heat advancedload, data["temp2"]

 // main iteration loop
 for (istep=0; istep < nstep; istep++) {
     #pragma hmpp heat callsite
     step_kernel(ni, nj, tfac, temp1, temp2);
     // swap the temp pointers
     temp = temp1;
     temp1 = temp2;
     temp2 = temp;
    }

 #pragma hmpp heat delegatedstore, data[temp1]
 #pragma hmpp heat release
```

Figure 3.1: HMPP implementation of 2D-Heat Equation

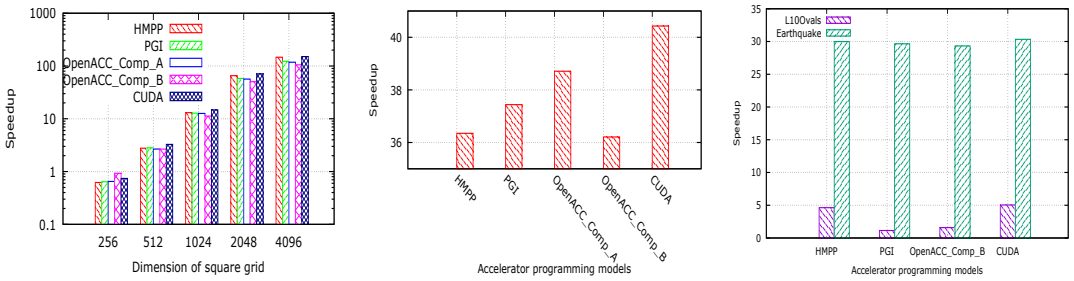
```
void step_kernel(...)
{
  #pragma acc kernels present(temp_in[0:ni*nj], temp_out[0:ni*nj])
  {
    // loop over all points in domain (except boundary)
    #pragma acc loop collapse(2) independent
    for (j=1; j < nj-1; j++) {
        for (i=1; i < ni-1; i++) {
            // find indices into linear memory
            // for central point and neighbours
            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);

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

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

#pragma acc data copyin(ni, nj, tfac) copy(temp1[0:ni*nj]) \
copyin(temp2[0:ni*nj]) deviceptr(temp)
{
    for (istep=0; istep < nstep; istep++) {
        step_kernel(ni, nj, tfac, temp1, temp2);
        // swap the temp pointers
        temp = temp1; temp1 = temp2; temp2 = temp;
    }
}
```

Figure 3.2: OpenACC implementation of 2D-Heat Equation



(a) 2D-Heat Equation (b) FDK (c) CLEVER

Figure 3.3: Application speedup with different models

once. Without FMA, rn(rn(X * Y) + Z) is composed of two steps and rounded twice. So their results would be slightly different. In 2D-heat equation, this kind of numerical difference will propagate with more iterations. To disable FMA, we used the options "–nvcc-options -fmad=false" in HMPP and "-ta=nofma" in PGI. When we used these compilation flags, the results of both HMPP and PGI are exactly the same as that of the CPU. Figure 3.3 (a) shows performance results when different programming models are applied on this application.

Let us consider the baseline (sequential) speedup to be 1 as shown in the Figure 3.3 (a). We see that for almost all the grid sizes, HMPP and PGI models perform as close as 80% to that of the CUDA version. The CUDA code has been considered from [59]. OpenACC_Compiler_A and OpenACC_Compiler_B perform approximately 80% and 70% respectively to that of the CUDA version. For the smallest grid size 256*256 considered, we see that neither the directive-based approaches nor the CUDA version was able to perform better than the sequential version. This is due to the fact that GPU is only suitable for massive computation purposes. For example, we see that for the grid size 4096*4096, almost all of the directive-based approaches

and CUDA model seem to achieve more than 100% to that of the sequential version.

3.2 Feldkamp-Davis-Kress (FDK) Algorithm (Image Processing domain)

Computed Tomography (CT) has been widely used in medical industry to produce tomographic images of specific areas of the body. It uses reconstruction technique that reconstructs an image of the original 3D-object from a large series of two dimensional X-ray images. As a set of rays pass through an object around a single axis of rotation, the produced projection data is captured by an array of detectors, from which a Filtered Back-Projection method based on the Fourier Slice Theorem is typically used to reconstruct the original object. Among various filtered back-projection algorithms, the FDK algorithm [31] is mathematically straightforward and easy to implement. It is important that the acquired reconstruction effect is good, the goal of this work is to speed up the reconstruction using directive-based programming models.

Algorithm 1 shows the pseudo-code of the FDK algorithm, this is comprised of three main steps: Weighting - calculate the projected data; Filtering - filter the weighted projection by multiplying their Fourier transforms; Back-projection - back-project the filtered projection over the 3D-reconstruction mesh. The reconstruction algorithm is computationally intensive and it has biquadratic complexity ($O(N^4)$), where N is the number of detector pixels in one dimension. The most time-consuming

step of this algorithm is back-projection which takes more than 95% of the whole algorithm. So we will concentrate on parallelizing the back-projection step.

Algorithm 1: Pseudo-code of FDK algorithm

```
Initialization;
foreach 2D image in detected images do
    foreach pixel in image do
        Pre-weight and ramp-filter the projection;
    end
end
foreach 2D image in detected images do
    foreach voxel in 3D reconstruct volume do
        Calculate projected coordinate;
        Sum the contribution to the object from all tilted fan beams;
    end
end
```

We follow the approach from [44] for implementation purposes. The back-projection has four loops. The three outermost loops will loop over each dimension of the output the 3D object, and the innermost loop will access each of 2D-detected image slices. First the code is restructured so that the three outermost loops are tightly nested and then we can apply *collapse* clause from PGI and OpenACC and use *gridify* clause from HMPP. The innermost loop is sequentially executed by every thread. All detected images are transferred from CPU to GPU by using the *copyin* clause, and the output 3D object (actually many 2D-image slices) are copied from GPU to CPU using the *copyout* clause. In HMPP, the input/output property of detected images is set as *in* and output object is set as *out*. To evaluate our implementations, we use the 3D Shepp-Logan head phantom data which has 300 detected images and the resolution of each image is 200*200. The algorithm produces 200*200*200 reconstructed

cube. Note that this algorithm also has the same issue as 2D-heat conduction i.e. the algorithm is sensitive to floating-point operations, so we disabled the FMA using the compilation flags. This does not mean that the results using FMA are incorrect, but just that we explore different implementation techniques while maintaining the same computation strategy, such that the results are consistent; we will also be able to make a fair comparison of results in this case. Figure 3.3 (b) shows the speedup for different accelerator programming models compared to that of the sequential version. A point to note is that the interval between the points in the Y-axis is small, therefore their performances are actually close to each other. The performance of all directive-based models (HMPP, PGI and OpenACC) are close to 90% of the CUDA code, that was written by us from scratch.

3.3 CLEVER Clustering (Data Mining domain)

In this section, we will parallelize a clustering algorithm called CLEVER (CLustEring using representatiVEs and Randomized-hill climbing) [29]. CLEVER is a prototype-based clustering algorithm that seeks for clusters maximizing a plug-in fitness function. Prototype-based clustering algorithms construct clusters by seeking an 'optimal' set of representatives-one for each cluster; clusters are then created by assigning objects in the dataset to the closet cluster representatives. Like K-means [53], it forms clusters by assigning the objects to a cluster with the closest representative. CLEVER uses a randomized-hill climbing to seek a good clustering, i.e. It samples p solutions in the neighborhood of the current solution and continues this

process until no better solutions can be found. Algorithm 2 shows the pseudo-code of CLEVER program code. It starts with randomly selecting *k'* representatives from the dataset O where *k'* is provided by the user. CLEVER samples p solutions in the neighborhood of the current solution and chooses the solution s with the maximum fitness value for *q(s)* as the new current solution provided there is an improvement in the fitness value. New neighboring solutions of the current solution are created by three operators: "Insert" which inserts a new representative into current solution; "Delete" which deletes an existing representative from current solution, and "Replace" which replaces an existing representative with a non-representative. Each operator is selected at a certain probability and the representatives to be manipulated are chosen at random. To prevent premature convergence, CLEVER will resample $p * q$ more solutions in the neighborhood before terminating, where q is the resampling rate. The description of CLEVER parameters are as follows [24]:

1. *k'*: initial number of clusters
2. *neighborhood-size*: maximum numbers of operators applied to generate a solution in the neighborhood
3. p: sampling rate, number of samples that is randomly selected from the neighborhood
4. q: resampling rate. If the algorithm fails to improve fitness with p and then $2 * p$ solutions, then sampling size in the neighborhood would be increased by factor $q - 2$
5. $imax$: maximum number of iterations in the algorithm

Algorithm 2: Pseudo-code of CLEVER algorithm

```
Input: Dataset O, k', neighborhood-size, p, q,imax
Output: Clustering X, fitness q(X), rewards for clusters in X

Current solution ← randomly selecting k' representatives from O ;
while iterations ≤ imax do
    Create neighbors of the current solution randomly using the given
    neighborhood definition, and calculate their respective fitness;
    if The best neighbor improved fitness then
        Current solution ← best neighbor;
    else
        Neighborhood of current solution is re-sampled by generating more
        neighbors;
        if re-sampling leads to better solution then
            Current solution ← best solution found by re-sampling;
        else
            Terminate returning the current solution;
        end
    end
end
```

We profile the CLEVER algorithm using GNU profiler before parallelizing the same. Statistical information gathered shows that the most time-consuming portion of the algorithm is the function that assigns objects to the closest representative which computes and compares a lot of distances. The original code is written in C++, this needed to be converted to C so that the algorithm could be supported by PGI, HMPP, and OpenACC model. (A point to note is that HMPP recognizes C++ code to an extent).

Since the data structure of the dataset is user-defined and the pointer operation is quite complicated, the accelerator region cannot be parsed by the compiler. Hence, the code is restructured so that it is relatively easier to be parsed by the compiler. The directives also give hints to the compiler such that all the iterations in the loop

are independent. Since the whole dataset is read-only, it is transferred to accelerator before the kernel is called. We can achieve this using both PGI and OpenACC model by using *copyin()* clause of data construct. We use the *region* directive of HMPP model and set the data transfer policy of the dataset as *atfirstcall* so that it is copied from the host to the accelerator only once. This will enable the dataset to stay in the global memory of the accelerator even if the kernel is called many times.

We evaluated the three directive-based models on two datasets called L10Ovals (Large 10Ovals) and Earthquake. The characteristics of these two datasets are shown in Table 3.2 and Table 3.3, respectively. The L10Ovals dataset has natural clusters representing L10Ovals containing 335,900 objects.

Earthquake dataset contains 330,561 earthquakes which are characterized by latitude, longitude, and depth of the earthquake. The goal is to find clusters where the variance of the earthquake depth is high; that is where shallow earthquakes are co-located with deep earthquakes.

Figure 3.3 (c) shows the speedup of how the four different models react to these two datasets. OpenACC_Compiler_A required a very long time to execute this algorithm, hence we do not include the speedup in the graph. The reason may be that the model is yet to provide an effective support to deal with pointer operations.

For L10Ovals dataset HMPP showed a speedup of 4.63x, which is very close to that of the CUDA version, 5.04x. We have considered the CUDA version of the algorithm from [24]. We noticed that among the OpenACC models, OpenACC_Compiler_B showed a speedup of 1.58x, performing poorer to CUDA. For

Table 3.2: L10Ovals dataset characteristics

Item	Description
Data size	335,900 objects
Attributes	<x, y, class label>
Distance Function	Euclidean Distance
Plug-in Fitness Function	Purity: Percentage of objects belonging to the majority class of the cluster

Table 3.3: Earthquake dataset characteristics

Item	Description
Data size	330,561 objects
Attributes	<latitude, longitude, depth >
Distance Function	Euclidean Distance
Plug-in Fitness Function	High Variance: Measures how far the objects in in the cluster are spread out with respect to earthquake depth

Earthquake dataset, the speedup achieved by HMPP, PGI, OpenACC_Compiler_B were 29.99x, 29.65x and 29.32x, respectively, these are almost the same as that of CUDA, which showed a speedup of 30.33x. Although L10Ovals and Earthquake have almost the same number of objects in their datasets we still see a significant difference in performance. This is primarily due to the characteristics of each of these datasets. L10Ovals has well defined and separated clusters therefore converges more quickly. The L10Ovals clustering task takes 21 iterations whereas the earthquake dataset clustering takes 216 iterations. Moreover the number of clusters searched in the earthquake clustering experiment is much higher than the number of clusters in L10Ovals experiment making it more time consuming to assign objects to clusters.

Table 3.4 shows the execution time consumed in seconds by the different directive-based models for three case studies. Results for the application 2D-Heat Equation shows that the time consumed by directive-based models were significantly lower than the time consumed by the serial version of the code. However the execution time for the CUDA code still appears to be the best. We see that HMPP model performs better than that of PGI and also that of OpenACC models. It could be due to slightly better optimization strategies offered by HMPP compiler implementations. Among the two OpenACC models, OpenACC_Compiler_A seems to perform better than OpenACC_Compiler_B. With respect to FDK algorithm, we notice that almost all the models perform similar to each other and in fact close to that of the CUDA code. CLEVER application shows that HMPP model performs the best compared to all other models. As mentioned earlier, we suspect that PGI compiler cannot handle pointer operations very efficiently yet. OpenACC_Compiler_A model may also have issues with the implementation of pointer operations. Hence the execution time seems to be long.

To summarize, directive-based models have indeed shown promising results compared to that of the CUDA version. In-depth research analysis of these models would lead to better performance. An important point to note is that the OpenACC model is still being constructed and the technical details may require fine tuning before we could actually make a deeper comparative analysis.

Table 3.4: Time(in sec) consumed by Serial, CUDA, HMPP, PGI and OpenACC versions of the code, only for most time-consuming dataset

Applications	Serial	CUDA	HMPP	PGI	OpenACC	
					A	B
2D Heat	8922.81	59.13	60.78	72.74	75.65	84.76
FDK	363.50	8.99	10.40	9.71	9.39	10.04
CLEVER	116.15	23.04	25.08	101.51	-	73.31

3.4 Summary

This chapter evaluates some of the prominent directive-based GPU programming models for three applications with different characteristics. We compare each of these models and tabulate the performances achieved by these models. We see that the performance is highly dependent on the application characteristics. The high-level models also provide a high-level abstraction by hiding most of the low-level complexities of the GPU platform. This makes programming easier leading to programmer productivity. We noticed that all the directive-based models performed much better than that of the serial versions of the applications being evaluated. We also observed that these models demonstrated performance results close enough to that of the CUDA version of the applications. However, we had to write almost different versions of the code before we could use a particular programming model, especially while using HMPP and PGI. OpenACC solved this portability issue by providing a single standard to be used to program GPUs.

Chapter 4

Parallelization and Optimization Strategies using Directive-based Model

In this chapter, we discuss the parallelization strategies to port NAS Parallel Benchmarks (NPB) [18] to GPGPUs using high-level compiler directives, OpenACC. NPB are well recognized for evaluating current and emerging multi-core/many-core hardware architectures, characterizing parallel programming models, and testing compiler implementations. The suite consists of five parallel kernels (IS, EP, CG, MG, and FT) and three simulated computational fluid dynamics (CFD) applications (LU, SP, and BT) derived from important classes of aerophysics applications. Together they mimic the computation and data move- ment characteristics of large scale computational CFD applications [18]. This is one of the standard benchmarks that is close

to real world applications. We believe that the OpenACC programming techniques used in this chapter can be applicable to other models such as OpenMP. Based on the application requirements, we will analyze the applicability of optimization strategies such as array privatization, memory coalescing, and cache optimization [79]. With vigorous experimental analysis, we will then analyze how the performance can be incrementally tuned. To the best of our knowledge, we are the first group to create an OpenACC benchmark suite for the C programs in NPB. Four benchmarks in this suite have been contributed to SPEC benchmark [43].

The performance of NPB benchmarks are well studied for conventional multi-core processor based clusters. Results in [42] show that OpenMP achieves good performance for a shared memory multi-processor. Other related works also include NPB implementations of High-Performance Fortran (HPF) [32], Unified Parallel C (UPC) [4] and OpenCL [60]. With high performance computing systems rapidly growing, hybrid-programming models become a natural programming paradigm for developers to exploit hardware characteristics. Wu et al. [72] discussed a hybrid OpenMP + MPI version of SP and BT benchmarks. Pennycook et al. [57] described the MPI+CUDA implementation of LU benchmark. The hybrid implementations commonly yield better performance if communication overhead is significant for MPI implementation and if computation for a single node is well parallelized with OpenMP. NAS-BT multi-zone benchmark was evaluated in [19] using OpenACC and OpenSHMEM hybrid model.

Grewe et al. [34] presented a compiler-based approach that automatically translate OpenMP program to optimized OpenCL code for GPUs and they evaluated

all benchmarks in NPB suite. Lee et al. [50] parallelized EP and CG from NPB suite using OpenACC, HMPP, CUDA, and other models and compared the performance differences. But our implementation is different from theirs for these two benchmarks.

4.1 Strategies

One of the main benefits of programming using a directive-based programming model is achieving performance by adding directives incrementally and creating portable modifications to an existing code. We consider the OpenMP version of NPB benchmarks as the starting point. Steps to parallelize legacy code using OpenACC are:

1) Profile the application to find the compute intensive parts, which are usually loops.

2) Determine whether the compute intensive loops can be executed in parallel. If not, perform necessary code transformations to make the loops parallelizable, if possible.

3) Prepend `parallel/kernels` directives to these loops. The `kernels` directive indicates that the loop needs to be executed on the accelerator. Using the `parallel` directive alone will cause the threads to run the annotated code block redundantly, until a `loop` directive is encountered. The `parallel` directive is mostly effective for non-loop statements.

4) Add `data` directives for data movement between the host and the device. This directive should be used with care to avoid redundant data movement, e.g. putting

`data` directives across multiple compute regions. Inside the data region, if the host or device needs some data at the end of one compute region, `update host` directive could be used to synchronize the corresponding data from the device to host, or `update device` directive is used if the device needs some data from the host.

5) Optimize data structures and array access pattern to efficiently use the device memory. For instance, accessing data in the global memory in a coalesced way, i.e. consecutive threads should access consecutive memory address. This may require some loop optimizations like loop permutation, or transforming the data layout that will change the memory access pattern.

6) Apply loop-scheduling tuning. Most of the OpenACC compilers provide some feedback during compilation informing users about how a loop is scheduled. If the user finds the default loop scheduling not optimal, the user should optimize the loop manually by adding more `loop` directives. This should lead to improvement in speedup.

7) Use other advanced optimizations such as the `cache` directive, which defines the variables to be cached by the kernel. Usage of the `async` clause will initiate data movement operations and kernel execution as asynchronous activities, thus enabling an overlap with continuous execution by the host CPU.

Some of the above steps need to be applied repeatedly along with profiling and feedback information provided by compiler and profilers. The practices and optimization techniques applied vary depending on the original parallel pattern and code structures of an application. Some of those techniques are summarized in the following sections. While these techniques have been used for optimizing parallel

```
for(k=0; k<N; k++){
    for(j=0; j<N; j++){
        for(i=0; i<N; i++){
            A[j][i] = ...
        }
    }
}
```

```
for(k=0; k<N; k++){
    for(j=0; j<N; j++){
        for(i=0; i<N; i++){
            AX[k][j][i] = ...
        }
    }
}
```

Figure 4.1: Array privatization example

program on CPUs, applying them on GPUs pose different challenges, particularly when using them in large code bases.

4.1.1 Array Privatization

Array privatization makes different threads access distinct memory addresses, so that different threads do not access the same memory address. It is a technique of taking some data that is common or shared among parallel tasks and duplicating it so that different parallel tasks can have a private copy to operate. Figure 4.1 shows an example for array privatization. If we parallelize the triple-nested loop on the left side of the figure using OpenMP for CPU and only parallelize the outermost loop, each thread handles the inner two loops. The array A could be annotated as OpenMP `private` clause to each thread, thus no modification is required to keep the memory usage minimal and improve the cache performance. However, this is not the case with OpenACC. In OpenACC, if the compiler still only parallelizes the outermost loop, multiple threads will be reading and writing to the same elements of the array A. This will cause data-race conditions, incorrect results, and potential crashes. An option here is to use the OpenACC `private` clause which is described in [7]. However, if the number of threads are very large, as typically in GPUs, it is very easy that all copies

of the array exceed the total memory available. Even though sometimes the required memory does not exceed the available device memory, it is possible that the assigned number of threads is larger than the number of loop iterations, and in this case some of the device memory will be wasted since some threads are idle. Also the life time of variable within a `private` clause is only for a single kernel instance. This limits our choice to apply loop-scheduling techniques since only the outermost loop can be parallelized. If the triple nested loop can be parallelized and each thread executes the innermost statements, thousands of threads still need to be created. Keeping the array A private for each thread will easily cause an overflow of memory available on the accelerator device. The right side of Figure 4.1 shows the array privatized code that addresses this issue. This solution added another dimension to the original array so that all threads can access different memory addresses of the data and no data race will happen.

4.1.2 Loop Scheduling Tuning

When parallelizing loops using OpenACC, `parallel/kernels` directives are inserted around the loop region. With the `parallel` directive, the user can explicitly specify how the loop is scheduled by setting whether the loop is scheduled in the level of `gang`, `worker`, or `vector`. With the `kernels` directive, however, loop scheduling is usually left to the compiler's discretion. Ideally, the compiler performs loop analysis and determines an optimal loop scheduling strategy. Our simple experiments show that, when using the `kernels` directive, the compiler makes good choices most of the times. But the compiler often opts for the less-efficient loop scheduling when the

```
     #pragma acc kernels loop gang
80:  for (k = 0; k <= grid_points[2]-1; k++) {
       #pragma acc loop worker
81:    for (j = 0; j <= grid_points[1]-1; j++) {
         #pragma acc loop vector
82:      for (i = 0; i <= grid_points[0]-1; i++) {
83:        for (m = 0; m < 5; m++) {
84:            rhs[m][k][j][i] = forcing[m][k][j][i];
85:         }
86:      }
87:    }
88:  }
```

Figure 4.2: Loop scheduling example

loop level is more than three. Figure 4.2 shows one of the scheduling techniques that delivers efficient loop scheduling. However the default scheduling by some compiler only applies to the loops in lines 82 and 83. The loops in line 80 and 81 are executed sequentially. This default option is very inefficient since the two outer most loops are not parallelized. Work in [65] discusses other loop scheduling mechanisms that could be applied in this context.

4.1.3 Memory Coalescing Optimization

The speedup from the parallel processing capability of GPU can be tremendous if memory coalescing is efficiently achieved. GPU has faster memory with unique data-fetching and locality mechanisms. In CPU, only one thread fetches consecutive memory data into the cache line, so the data locality is limited to only one thread. In GPU, however, consecutive threads fetch consecutive memory data into the cache line allowing better data locality. For instance, the code in Figure 4.2 is already optimized for memory coalescing. The i loop is vectorized with the rightmost dimension of *rhs* and *forcing* is i. In the original serial-code version, the memory-access pattern of *rhs*

and *forcing* are rhs[k][j][i][m] and forcing[k][j][i][m], respectively. But for memory-coalescing purposes, we need to reorganize the data layout so that the dimension "m" is not on the farther right. Since C language is row-major, the right-most dimension is contiguous in memory. We need the threads to access (i.e. the vector loop) the right-most dimension. So after data-layout reorganization, the memory access-pattern becomes rhs[m][k][j][i] and forcing[m][k][j][i].

4.1.4 Data Motion Optimization

Data transfer overhead is one of the important factors to consider when determining whether it is worthwhile to accelerate a workload on accelerators. Most of the NPB benchmarks consist of many global variables that persist throughout the entire program. An option to reduce data transfer will be to allocate the memory for those global variables at the beginning of the program so that those data reside on the device until the end of the program. Since some portion of the code cannot be ported to the device, we could use `update` directive to synchronize the data between the host and device.

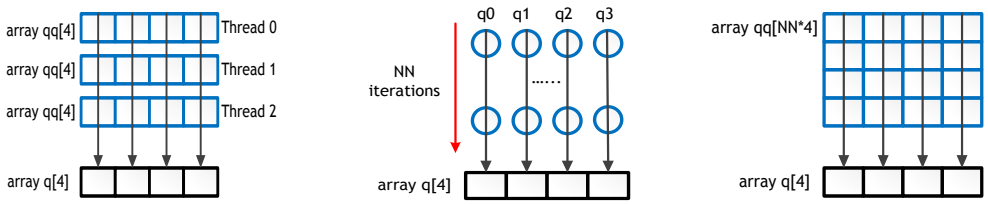
4.1.5 Cache Optimization

NVIDIA Kepler GPU memory hierarchy has several levels of memory, including global memory, then L2 cache for all SMs and the registers, L1 cache, shared memory, and read-only data cache for each SM. In Kepler GPU, L1 cache is reserved only for local memory accesses such as register spilling and stack data. Global loads are

cached in L2 cache only [8]. Here the usage of both L1 and L2 is controlled by the hardware and they are not manageable by the programmer. The shared memory can be utilized by the `cache` directive in OpenACC. Although the read-only data cache is also controlled by the hardware, the programmer can give some hints in the CUDA kernel file to tell the compiler what the read-only data list is. Since the read-only data cache is a device-specific memory, OpenACC does not have any directive to utilize this cache. However, when the user specifies the device type when using OpenACC, the compiler can perform some optimizations specific to the specified device. We implemented this optimization in the compiler used so that the compiler can automatically determine the read-only data in a kernel by scanning all data in that kernel and then add "const __restrict__" for all read-only data and add "__restrict__" for other data that has no pointer alias issue. These prefix are required in CUDA if the user wants the hardware to cache the read-only data [8]. This compiler optimization can improve the performance significantly if the read-only data is heavily reused.

4.1.6 Array Reduction Optimization

Array reduction means every element of an array needs to do reduction. This is supported in OpenACC specification which only supports scalar reduction. Different programming models solve this issue differently. As seen in Figure 4.3, there are several ways to solve the array reduction in array q in EP benchmark. In the OpenMP version, each thread has its own private array qq to store the partial count of q, for the purpose of reducing the overhead of atomic update of shared variables. Thus,



(a) OpenMP solution (b) OpenACC solution 1 (c) OpenACC solution 2

Figure 4.3: Solutions of array reduction in EP benchmark.

each thread only needs to perform an atomic update on q with its own partial sum qq. Since OpenACC does not support array reduction, Lee et al. [50] decomposed the array reduction into a set of scalar reductions which is seen in Figure 4.3 (b). This implementation is not scalable as it cannot handle large array reduction, and the size of the result array must be known at compile time. Our solution, as seen in Figure 4.3 (c), uses the array privatization technique to make a copy of q and expand it by another dimension with size NN (declared as new variable qq). In this way, each thread does its own work independently and writes the result into its own portion of the global memory. Finally, each element of q can be obtained by doing reduction just once with qq.

4.1.7 Scan Operation Optimization

The NAS IS benchmark has both inclusive and exclusive prefix-sum/scan operations. The inclusive scan takes a binary operator $\bigoplus$ and an array of N elements $[A_0, A_1, ..., A_{N-1}]$ and returns the array $[A_0, (A_0 \bigoplus A_1), ..., (A_0 \bigoplus A_1 \bigoplus ... \bigoplus A_{N-1})]$. Exclusive scan is defined similarly, but shifts the output and uses an identity value I

as the first element. The output array is $[I, A_0, (A_0 \bigoplus A_1), ..., (A_0 \bigoplus A_1 \bigoplus ... \bigoplus A_{N-2})]$. In scan loop, an element in the output array depends on its previous element, and because of such data dependence, it cannot be parallelized by the `loop` directive in OpenACC. To overcome such limitations, we provided some extensions to the OpenACC standard. We introduced a new `scan` clause to the `loop` directive followed by usage of recursive algorithm in [37] to handle the scan operation for arbitrary input array size. The proposed new directive format is `#pragma acc loop scan(operator:in-var,out-var,identity-var,count-var)`. We implemented this optimization in OpenUH compiler. Our implementation also supports in-place scan in which the input and output array are the same. By default, however, the scan is not in-place which means the input array and output array are different. For inclusive scan, the identity value is ignored. For exclusive scan, the user has to specify the identity value. For in-place inclusive scan, the user must pass IN_PLACE in in-var For in-place exclusive scan, the user must pass IN_PLACE in in-var and specify the identity value. The identity value must be the same as the first value of the provided array.

4.2 Performance Evaluation

The experimental setup is a machine with 16 cores Intel Xeon x86_64 CPU with 32GB main memory, and an NVIDIA Kepler GPU card (K20) with 5GB global memory. We use OpenUH compiler to evaluate the performance of C programs of NPB on GPUs. This open source compiler provides support for OpenACC 1.0 at the time

Table 4.1: Comparing elapsed time for NPB-ACC, NPB-SER, NPB-OCL, and NPB-CUDA (time in seconds), "-" implies no result due to "out of memory" issue. For NPB-CUDA, only LU, BT, and SP are accessible. Data size increases from A to C, ∼16x size increase from each of the previous classes. The "Techniques Applied" numbers refer to the optimizations described in corresponding sections. Other than listed techniques, we have optimized all of our OpenACC implementations including using data-motion optimizations as well.

Benchmark	EP			CG			FT			IS		
Data Size	A	B	C	A	B	C	A	B	C	A	B	C
NPB-SER	46.56	187.02	752.03	2.04	101.80	269.96	6.97	79.42	390.35	0.99	4.04	17.00
NPB-OCL	0.27	0.82	2.73	0.36	13.42	35.39	1.49	32.55	-	0.04	0.35	1.74
NPB-ACC	0.49	1.96	7.85	0.36	9.51	21.28	1.18	9.20	-	0.06	0.23	1.94
Techniques Applied	3.1, 3.3, 3.6			3.5			3.1, 3.3			3.7		
Benchmark	MG			LU			BT			SP		
Data Size	A	B	C	A	B	C	A	B	C	A	B	C
NPB-SER	2.57	11.48	99.39	60.38	264.71	1178.97	93.14	387.51	1626.33	52.17	225.26	929.85
NPB-OCL	0.13	0.61	5.48	5.32	16.70	54.88	46.12	167.48	-	11.84	54.35	288.40
NPB-ACC	0.24	1.12	7.55	6.64	26.12	103.97	15.25	63.61	226.70	3.45	15.90	57.46
NPB-CUDA	-	-	-	5.79	19.58	75.06	13.08	53.46	216.98	2.47	11.17	43.16
Techniques Applied	3.1, 3.2, 3.5			3,1, 3.3, 3.5			3,1, 3,2, 3.3, 3.5			3,1, 3.2, 3.3		

of writing this chapter. Although implementations for OpenACC 2.0 are beginning to exist, they are not robust enough to be used to evaluate NPB-type benchmarks. For evaluation purposes, we compare the performances of our OpenACC programs with serial and third-party well-tuned OpenCL [60] and CUDA programs [3] (that we had access to) of the NAS benchmarks. All OpenCL benchmarks run on GPU rather than CPU. We used GCC 4.4.7 and -O3 flag for optimization purposes. The CUDA version used by the OpenACC compiler is CUDA 5.5. The OpenCL codes are compiled by GCC compiler and link to CUDA OpenCL library.

Table 4.1 shows the execution time taken by NPB-SER, NPB-OCL, and NPB-ACC, which are the serial, OpenCL and OpenACC versions of the NPB benchmarks, respectively. For the FT benchmark, OpenCL, and OpenACC codes could not execute for problem size Class C. The reason being, FT is memory limited; the Kepler

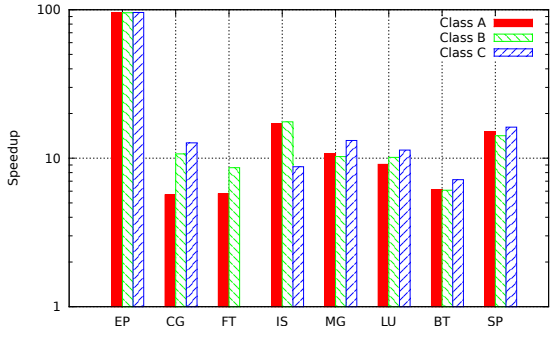


Figure 4.4: NPB-ACC speedup over NPB-SER

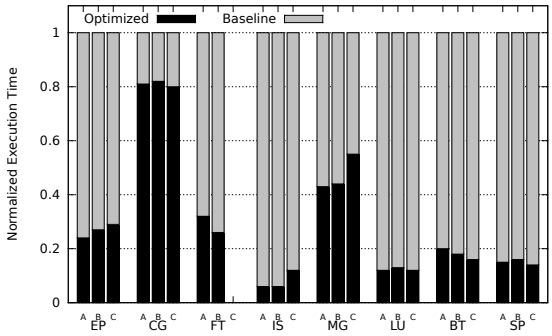


Figure 4.5: NPB-ACC performance improvement after optimization

card in use ran out of memory. Same to do with the OpenCL program for BT benchmark. However, this was not the case with OpenACC. The reason being: OpenCL allocated the device memory for all the data needed in the beginning of the application. With OpenACC program, different solver routines have different memory coalescing requirements, as a result, different routines have different data layout. For those data, OpenACC program only allocates the device memory in the beginning of the solver routines and frees the device memory before exiting these routines. This explains that the data in the OpenCL program are active throughout the full application, but for the OpenACC program, some data is only active in some of the routines, hence saving the requirement for the total-memory at a given time.

Figure 4.4 shows the speedup of NPB-ACC over NPB-SER for the benchmarks that have been optimized. It is quite clear that all the benchmarks show significant speedup, especially EP. This is because EP is an embarrasingly parallel benchmark that has only few data transfers and our optimization technique enabled all the memory accesses to be nicely coalesced. Most of the benchmarks observed increase-in-speedup as the data problem size increased, except IS. This is because, IS uses

buckets to sort an input integer array, the value of the bucket size is fixed as defined in the benchmark, no matter what the data size is. As a result, when the data size becomes larger, the contention to each bucket becomes more intense decreasing the performance to quite an extent. However this does not affect the numerical correctness due to atomic operations in place to prevent data races.

We measure the effectiveness of the potential optimizations applied in Figure 4.5 by comparing the baseline and the optimized versions of the benchmarks. The baseline versions use only array privatization in order to parallelize the code and data motion optimization to eliminate unnecessary data transfer overhead and not any other optimizations discussed. The optimized versions exploit the optimizations discussed earlier.

IS benchmark demonstrates much improvement from the baseline version. This is due to the *scan* operation discussed earlier. CG mainly benefits from cache optimization, the rest of the optimizations all seem to have a major impact on the benchmark's performance. FT benchmark shows improvement due to Array of Structure (AoS) to Structure of Array (SoA) transformation since the memory is not coalesced in AoS data layout but coalesced in SoA data layout. Note that the execution time of the three pseudo application benchmarks LU, BT, and SP are even less than 20% of the time taken by the baseline version. LU and BT observed over ∼50% and ∼13% of performance improvement using cache optimization, since both the benchmarks extensively use read-only data.

LU, BT, and SP benchmarks benefit significantly from memory coalescing optimizations since in the serial code the memory access is not coalesced at all for GPU

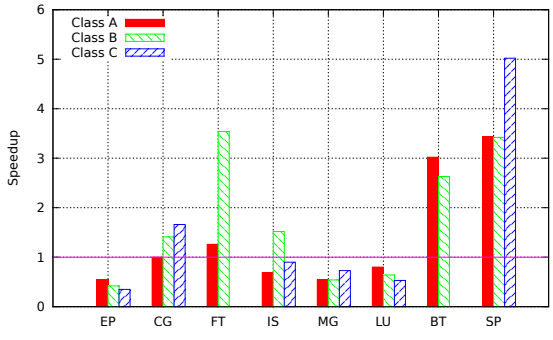


Figure 4.6: NPB-ACC speedup over NPB-OCL

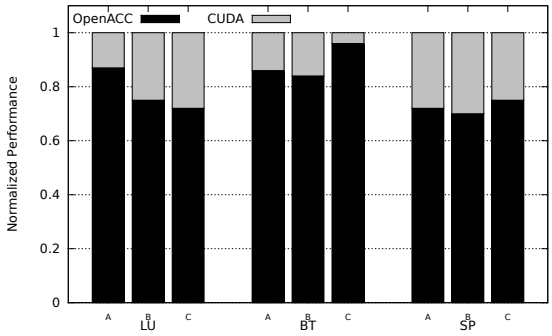


Figure 4.7: NPB-ACC performance comparison with NPB-CUDA

architecture. Memory coalescing requires explicit data layout transformation. We observed that tuning loop scheduling is very crucial for MG, BT, and SP benchmarks since these benchmarks have three or more levels of nested loops. The compiler could not always identify the best loop scheduling option, requiring the user to intervene.

These analysis of benchmark results indicate that it is insufficient to simply insert directives to an application no matter how simple or complex it is. It is highly essential to explore optimization techniques, several of those discussed in this chapter, to not only give the compiler adequate hints to perform the necessary transformations but also perform transformations that can exploit the target hardware efficiently.

To evaluate our optimizations further, we compare the NPB-ACC with well-tuned code written with the low-level languages OpenCL (NPB-OCL) and CUDA (NPB-CUDA). Figure 4.6 and Figure 4.7 show the corresponding results. Figure 4.6 shows that the EP program using OpenACC is around 50% slower than that of the OpenCL program. This is because the OpenACC version uses array privatization, which increases the device memory in turn exceeding the available memory limit.

Therefore we use the blocking algorithm to move data chunk by chunk into the device. We launch the same kernel multiple times to process each data chunk. The OpenCL program, however, uses the shared memory in GPU and does not need to use array privatization to increase the GPU device memory, therefore it only needs to launch the kernel once. Faster memory access through shared memory and reduced overhead due to less number of kernel launches improved the results for OpenCL. Although OpenACC provides a `cache` directive that has similar functionalities to CUDA's shared memory, the implementation of this directive within OpenACC compiler is not technically mature enough yet. This is one of the potential areas where support in OpenACC can be improved.

Performance of OpenACC programs of benchmarks BT and SP are much better than that of the OpenCL programs. The reason is two-fold. First up, the OpenCL program does not apply the memory coalescing optimization; memory accesses are highly uncoalesced. Secondly, the program does not apply loop-fission optimization; there are very large kernels. Although the large kernel contains many parallelizable loops, they are only executed sequentially inside the large kernel. On the contrary, the OpenACC program uses loop fission, thus breaking the large kernel into multiple smaller kernels and therefore exposing more parallelism.

The OpenACC program for benchmark MG appears to be slower than that of the OpenCL program. This is because former program uses array privatization, which needs to allocate the device memory dynamically in some routines, however the latter uses shared memory, which has faster memory access and no memory allocation overhead. The OpenACC program for benchmark FT is faster than OpenCL, since

OpenACC transforms the AoS to SoA data layout to enable memory coalescing. The OpenACC program for benchmark LU is slower than OpenCL since the former privatizes small arrays into the GPU global memory, but OpenCL uses the small array inside the kernel as in they will be allocated in registers or possibly spilled to L1 cache. The memory access from either register or L1 cache is much faster than that from the global memory as used by OpenACC.

Figure 4.7 shows the normalized performance of NPB-ACC and NPB-CUDA. We found CUDA programs for only the pseudo applications, i.e. LU, BT, and SP, hence we have only compared OpenACC results of these applications with CUDA. The result shows that OpenACC programs for LU, BT and SP benchmarks achieve 72%∼87%, 86%∼96% and 72%∼75% to that of the CUDA programs, respectively. The range denotes results for problem sizes from CLASS A to C. We see that the performance gap between CUDA and OpenCL is quite small. The reasoning for the small performance gap is the same as that we have explained for the OpenCL LU benchmark. It is quite evident that careful choice of optimization techniques for high-level programming models can result in reaching performance very close to that of a well hand-written CUDA code. We believe that as the OpenACC standard and its implementation evolve, we might even be able to obtain better performance than CUDA. Thus successfully achieving portability as well.

4.3 Discussion

4.3.1 Programmability

Programming heterogeneous systems can be simplified using OpenACC-like directive-based approaches. An expected advantage is that they help maintain a single code base catering to multiple targets, leading to considerably lesser code maintenance. However, in order to achieve good performance, it is insufficient to simply insert annotations. The user's intervention is required to manually apply certain code transformations. This is because the compiler is not intelligent enough to determine the optimal loop scheduling for accelerated kernels and optimize the data movement automatically. With respect to memory-coalescing requirement, currently there is no efficient mechanism to maintain different data layout for different devices, the user has to change the data layout. There is no compiler support that can effectively utilize the registers and shared memory in GPU that play an important role in GPUs. Data movement is one of the most important optimization techniques. So far it has been the user's responsibility to choose the necessary data clause and to move data around in order to get the best performance. If the compiler provides suitable hints, this technique can prove to be quite useful.

4.3.2 Performance Portability

Achieving performance portability can be quite tricky. Different architectures demand different programming requirements. Merely considering a CPU and a GPU;

obtaining optimal performance from CPU largely depends on locality of references. This holds good for GPUs as well, but the locality mechanism of the two architectures are different. The amount of computation that a CPU and a GPU can handle also differs significantly. It is not possible to maintain a single code base for two different architectures unless the compiler automatically handles most of the optimizations internally. Performance portability is not only an issue with just the architecture, but also an issue that different compilers can provide a different implementation for a directive/clause. Moreover the quality of the compilation matters significantly. For example, the OpenACC standard allows the user to use either `parallel` or `kernels` in the compute region. The `kernels` directive allows the compiler to choose the loop-scheduling technique to be applied i.e. analyze and schedule each loop level to `gang/worker/vector`. A compiler can use its own technique to schedule the loop nest to nested gang, worker and vector; this is typically not part of the programming model standard. As a result, the performance obtained using the `kernels` directive is different for different compilers. On the contrary, the code that uses `parallel loop` directive is more portable since this allows the user to have control over explicitly adopting the loop scheduling. Also the transformations of the `parallel` directive by most of the OpenACC compilers are similar.

Chapter 5

Compiler and Runtime Driven Optimizations

In Chapter 4 we have presented several parallelization and optimization techniques that are required to obtain the high performance when using the directive-based programming model OpenACC for GPUs. However, most of those techniques are applied by the user. In the remaining chapters, we focus on the optimizations that are applied automatically by the compiler and runtime. Since OpenACC is still in its early stages, most of the existing OpenACC compilers are commercial ones such as PGI and Cray compilers. The challenge with these commercial compilers is that it is not straightforward to deal with the compile-time and runtime errors and explain varying performance numbers between different compilers and even between different versions of the same compiler [69]. So To study the implementation challenges and the principles and techniques of directive-based model, we built an open source

OpenACC compiler in a main stream compiler framework (OpenUH as a branch of Open64). In this chapter we will present how the runtime is designed for OpenACC directive-based programming model, and how the reduction algorithm is optimized in the compiler [80].

5.1 Runtime

The OpenACC annotated source code is parsed by the compiler to extract the device kernels and translate the OpenACC directives into runtime calls. Then two parts of the code are generated: one part is the host code compiled by the host compiler, another part is the kernel code compiled by the accelerator compiler. The runtime is responsible for handling data movement and managing the execution of kernels from the host side.

5.1.1 Runtime Library Components

The runtime library consists of three modules: context module, memory manager, and kernel loader. The context module is in charge of creating and managing the virtual execution environment. This execution environment is maintained along the liftime of all OpenACC directives. All context and device related runtimes, such as `acc_init()` and `acc_shutdown()`, are managed by this module.

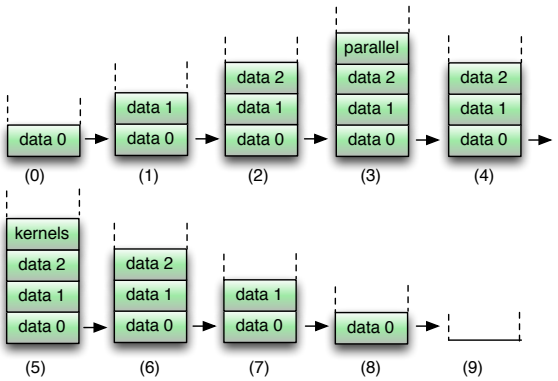
The memory manager helps to control the data movement between the host and device. The compiler translates the clauses in `data` and `update` directives into

corresponding runtime calls in this module. OpenACC provides a `present` clause that indicates the corresponding data list are already on the device, in order to avoid unnecessary data movement. To implement this feature, the runtime creates a global hash map that stores all the device data information. Whenever a compiler parses a `present` clause, it will translate this clause to the runtime call to check if the data list in the `present` clause are in the map. If the data exists in the map, then there is no need for data movement. If the data does not exist in the map, the compiler will issue a compilation error. Note that the data allocated from `acc_malloc()` and the data in the `deviceptr` clause do not have a corresponding host address since they are only allowed to use on the device.

The global hash map maintained in the runtime can be used for both structured data region and unstructured data directives. Each entry in the hash map includes the host address, device address and the data size so that we can find the device address given a host address or vice versa. The runtime maintains a region stack to track the region chain and the new data created within each region. Figure 5.1 (a) shows an example OpenACC code and (b) shows the structure of the region stack. The region stack can guarantee that the data list created at the entry of a region (either data region or compute region) will be freed at the exit of the same region. Whenever a new data region is encountered, a region pointer is pushed into the region stack. If the regions are not nested, then they are pushed into the stack in sequence. All the newly created data in the region level i are appended to a linked list and then inserted into the hash map. The device memory is allocated for these data and copied to the device as necessary (for `copy` and `copyin` clauses). At the

```
#pragma acc data create(...) //data 0
{ //(0)
  #pragma acc data create(...) //data 1
  { //(1)
    #pragma acc data create(...) //data 2
    { //(2)
      #pragma acc parallel create(...)
      { //(3)
      } //(4)
      #pragma acc kernels create(...)
      { //(5)
      } //(6)
    } //(7)
  } //(8)
} //(9)
```

(a) OpenACC code example



(b) Region stack flow for the code in (a)

Figure 5.1: Runtime region stack structure

exit of a region, the runtime will pop the region pointer from the region stack, copy the data from the device address to the host address (for `copy` and `copyout` clauses) and free the data list created in that region.

The purpose of kernel loader module is to launch the specified kernel from the host. After the kernel file is compiled by the accelerator compiler, the runtime loads the generated file, setups the threads topology and pushes the corresponding arguments list into the kernel parameter stack space, then launch the specified kernel. Since different kernels have different number of parameters, a vector data structure is created to store the kernel arguments to guarantee that the kernel argument size is dynamic. Another work to do before launching a kernel is to specify the threads topology. The compiler parses the loop mapping strategy and then generates the corresponding thread topology. The recommended threads value in the topology is described in section 5.1.2.

5.1.2 Launch Configuration Setting

The launch configuration (or threads topology) is an important factor affecting application performance. Since we map gangs to blocks in grid and vector into threads within each block, the values of blocks and threads need to be chosen carefully. Too many blocks and threads may generate potential scheduling overhead, and too few threads and blocks cannot take advantage of the whole GPU hardware resources such as cache and registers. The threads topology setting should consider exposing enough parallelism in each multiprocessor and balancing the workload across all multiprocessors. Different launch configurations affect the performance differently. How to choose the appropriate launch configuration automatically by the compiler is discussed in Chapter 7. In OpenUH, if the user did not specify the gang and vector number, the default value will be used. The default vector size is 128 because the Kepler architecture has quad warp scheduler that allows to issue and execute four warps (32 threads) simultaneously. The default gang number is 224 since Kepler 20 GPU allows up to 16 thread blocks per SM and there are 14 SMs.

5.1.3 Execution Flow in Runtime

Figure 5.2 gives a big picture of the execution flow at runtime and Figure 5.3 shows an example. In the beginning, `acc_init()` is called to setup the execution context. This routine can be either called explicitly by the user or implicitly generated by the compiler. Next the data clauses will be processed. There are different kinds of data clauses (e.g. `copyin`, `copyout`, and `copy`) and these data clauses may be in either

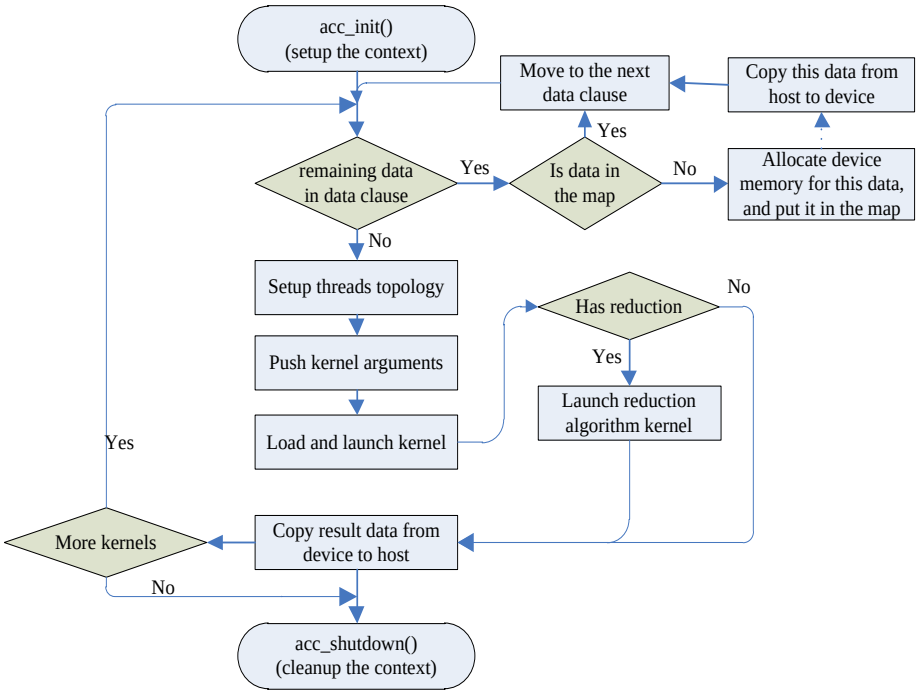


Figure 5.2: Execution flow with OpenACC runtime library

of `data`, `parallel` or `kernels` directive. The data list in the first data clause is scanned and they are checked if in the global hash map. If they are not in the map, then the device memory will be allocated for them and put them in the map. If the data needs to be accessed from the device, for instance those in `copyin` or `copy` or `update device` clauses, then they are transferred from the host to device. These data clauses will be scanned and processed. The purpose of this step is to make the data ready before launching the kernels. After the data is ready, we will setup the threads topology and push the corresponding arguments to the kernel. So far everything is ready and we can safely load and launch the kernel. If the kernel needs to do some reduction operation, after this kernel is finished a separate reduction algorithm kernel will be launched. The result data, for instance those in `copyout` or

```
#pragma acc data copyin(A,B) copyout(C)
{
   #pragma acc kernels loop
   for(i=0; i<N; i++)
   {
       C[i] = A[i]+B[i];
   }
}
```

(a) OpenACC code example

```
acc_init();
is_present=accr_in_hashmap(B,&d_B,0,N,4);
if(is_present == 0){
    accr_malloc_on_device(B,&d_B,N*4);
    accr_memin_h2d(B, d_B, N*4);
}
is_present=accr_in_hashmap(A,&d_A,0,N,4);
if(is_present == 0){
    accr_malloc_on_device(A, &d_A, N*4);
    accr_memin_h2d(A, d_A, N*4);
}
if(is_present == 0)
    accr_malloc_on_device(C, &d_C, N*4);
is_present=accr_in_hashmap(&N,&d_N,0,1,4)
    ;
if(is_present == 0){
    accr_malloc_on_device(&N, &d_N, 4);
    accr_memin_h2d(&N, d_N, 4);
}
accr_set_default_gang_vector ();
accr_push_kernel_param_pointer(&d_C);
accr_push_kernel_param_pointer(&d_B);
accr_push_kernel_param_pointer(&d_A);
accr_push_kernel_param_pointer(&d_N);
accr_launchkernel("__accrg_vecadd", "
    vecadd.ptx");
accr_memout_d2h(d_C, C, N*4);
acc_shutdown();
```

(b) Translated runtime calls by compiler

Figure 5.3: OpenACC execution flow example

`copy` or `update host` clauses, will be transferred from the device to host. Finally `acc_shutdown()` is called to release all the resources and destroy the context.

5.2 Reduction Algorithm

5.2.1 Related Work

Reduction is a well known topic, in this section, we will discuss some of the existing implementations of reduction operation in different programming models. Performing reduction is a challenge, different models may adopt different implementation strategies.

Reduction in OpenMP: In OpenMP programming model, the threads are one dimensional, hence there are not many use cases for reduction. Liao et al. [52] implemented the OpenMP reduction in two steps in the OpenUH compiler. In the first step, a reduction variable is substituted with a local copy in each thread to participate in the reduction operation computation. In the second step, values of local copies are combined into the original reduction variable protected by a critical section. Before the critical section, a barrier is inserted to ensure that those private reduction values are all ready. After the critical section, another barrier is inserted to ensure the final reduction value is available after that barrier. The barrier here is an expensive operation. So Nanjegowda et al. [55] tried to use tree barrier and tournament barrier algorithms to improve the barrier performance. Their barrier algorithms have complex control flows and therefore are not suitable for GPU architecture. Reduction in GCC compiler [6] is implemented by creating an array of the type of the reduction variable so that it can be indexed by the thread id, then each thread stores its reduction value into the array, finally the master thread iterates over the array to collect all private reduction values and generate the final reduction

value. These approaches cannot be applied to OpenACC since OpenMP has only one level of parallelism but OpenACC has three levels of parallelism, and OpenACC has no support for critical section.

Reduction in CUDA and OpenCL: Harris et al. [36] discussed seven different reduction algorithms in CUDA. Their optimizations include global memory coalescing to reduce memory divergence, shared-memory optimization avoiding shared memory bank conflict, partial- and full-loop unrolling and algorithm cascading. They analyzed the cost of each algorithm and compared the performance of all reduction algorithms and achieved bandwidth close to the theoretical bandwidth. We have leveraged some of these algorithms in our work. OpenCL uses different reduction strategies. Catanzaro [5] took advantage of the associativity and commutativity properties of reduction operation to restructure a sequential loop into reduction trees and then used several strategies for building efficient reduction trees. They observed that most of their parallel-reduction trees are very inefficient because of a number of communication and synchronization required among threads. Better performance can be achieved if much of the reduction is done serially. In both CUDA and OpenCL, the reduction happens in at most two levels of parallelism: thread blocks and threads within a block, but OpenACC reduction can happen in three levels of parallelism: `gang`, `worker`, and `vector`, so these algorithms cannot be directly applied in OpenACC reduction algorithm.

Reduction in OpenACC: Komoda et al. [45] proposed a new directive in OpenACC to solve the array reduction problem and to overcome the limitation of

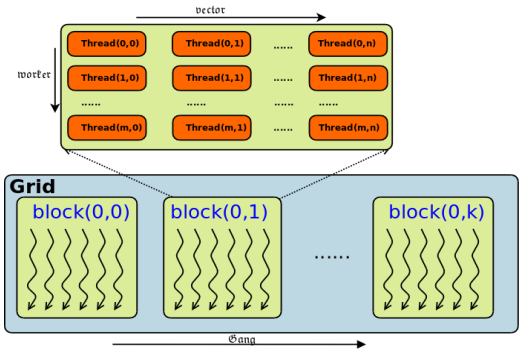


Figure 5.4: GPGPU thread block hierarchy

supporting only scalar reduction in current OpenACC specification. The array reduction means that every element of an array needs to do reduction. Their array reduction implementation can work on both single GPU and multi-GPU platform. However, they do not mention the complexity of scalar reduction in OpenACC and all the possible reduction usages. Our work in this dissertation focuses on the scalar reduction in OpenACC standard.

5.2.2 Parallelism Mapping

OpenACC supports three levels of parallelism: coarse-grained parallelism "gang", fine-grained parallelism "worker" and vector parallelism "vector". The programmer can create several gangs and a single gang may contain several workers and a single worker may contain several vector threads. The iterations of a loop can be executed in parallel by distributing the iterations among one or multiple levels of parallelism of GPGPU architectures. Mapping loops to the hardware in OpenACC is implementation dependent. Figure 5.5(a) shows a triple nested loop example that exhibits all the three levels of parallelism. In this example, assume each loop can be executed in

```
#pragma acc loop gang
for(k=k_start; k<k_end; k++){
  #pragma acc loop worker
  for(j=j_start; j<j_end; j++){
    #pragma acc loop vector
    for(i=i_start; i<i_end; i++){
        ...
    }
  }
}
```

(a) Loop nest example

```
k = blockIdx.x + k_start;
for(k=k; k<k_end; k+=gridDim.x){
  j = threadIdx.y + j_start;
  for(j=j; j<j_end; j+=blockDim.y){
    i = threadIdx.x + i_start;
    for(i=i; i<i_end; i+=blockDim.x){
      ...
    }
  }
}
```

(b) Compiler implementation

Figure 5.5: Loop nest example with OpenACC parallelisms

parallel, then k loop is distributed across all gangs, j loop is distributed across all workers in a single gang, and i loop is distributed across all vector threads of one worker.

Our OpenACC implementation is built on top of the SIMT execution model of CUDA. Table 5.1 shows the CUDA terminologies that is used in our OpenACC implementation. In OpenUH compiler, `gang` maps to a thread block, `worker` maps to the Y-dimension of a thread block and `vector` maps to the X-dimension of a thread block. Based on these definitions, the implementation details for the loop nest discussed in Figure 5.5(a) is shown in Figure 5.5(b). Here we add the start offset to the index of each level of threads so that the working threads start from 0 that effectively reduce the thread divergence. Another possible implementation is to get the thread id and then determine whether the id is greater than the start position and lesser than the end position in each loop level. In that case, the threads whose ids are lesser than the start position will not participate in the computation and will be idle all the time leading to thread divergence. In our implementation, the threads in each loop level increase along with their own stride size, so that each

Table 5.1: CUDA terminology in OpenACC implementation

Term	Description
threadIdx.x	thread index in X dimension of a thread block
threadIdx.y	thread index in Y dimension of a thread block
blockDim.x	no. of threads in X dimension of a thread block
blockDim.y	no. of threads in Y dimension of a thread block
blockIdx.x	block index in X dimension of the grid
gridDim.x	no. of blocks in X dimension of the grid

thread processes multiple elements of the input data. This solves the issue of limited number of threads availability in the hardware platform. Our implementation is designed in a way that it is independent of the number of threads used in each loop level [64]. However, the appropriate number of threads may enable coalesced memory access and improve performance.

In the loop nest example, we assume that all the iterations are independent, but most time the loop nest may contain reduction operation and the reduction may appear anywhere on the loop nest. In the next section, we will discuss such cases and how they are implemented in our OpenUH compiler.

5.2.3 Parallelization of Reduction Operations for GPGPUs

The reduction operation applied to a parallel loop uses a binary operator to operate on an input array and generates a single output value for that array. Each thread has its own local segments copy of the input array when the loop is distributed among threads. The operation that consolidates the results from the thread-local copies of the segments using the reduction operation is the issue that we are addressing in this

dissertation. The approach to performing parallel reductions depends on how the loop nests are mapped to the GPGPU's thread hierarchy. Moreover, reduction operation always implies a barrier synchronization, this may introduce runtime overhead, hence we need to be cautious to only include the synchronization when necessary.

Although most reduction operations are inherently not parallel, for those that have the properties of associativity and commutativity [5], we are able to apply the divide and conquer method to achieve parallel execution. That is, let us assume there are three input variables, *a1*, *a2*, and *a3*, and the reduction operator 'sum' does $a1 + a2 + a3$. The associativity of a binary operator is a property that determines how operators of the same order of operations are grouped without using parentheses. Since each reduction uses only one operator and this operator has equal precedence, the operation can be grouped differently. For instance, $(a1+a2)+a3$ and $a1+(a2+a3)$ will deliver the same output as $a1+a2+a3$. The commutativity of a binary operator is a property that changes the order of operations and does not change the result. For instance, $a3+a1+a2$ and $a2+a3+a1$ will deliver the same output as $a1+a2+a3$. All of the OpenACC reduction operators satisfy both associativity and commutativity properties. So the reduction operations can be performed in any order as long as it uses a single operator and includes all of the input data. These are the vital properties that we will be applying in our implementation.

```
#pragma acc parallel \
            copyin(input) \
            copyout(temp)
{
  #pragma acc loop gang
  for(k=0; k<NK; k++){
    #pragma acc loop worker
    for(j=0; j<NJ; j++){
      int i_sum = j;
      #pragma acc loop vector \
              reduction(+:i_sum)
      for(i=0; i<NI; i++)
        i_sum+=input[k][j][i];
      temp[k][j][0] = i_sum;
    }
  }
}
```

(a) OpenACC code

```
k = blockIdx.x;
for(k=k; k<NK; k+=gridDim.x){
  j = threadIdx.y;
  for(j=j; j<NJ; j+=blockDim.y){
    i = threadIdx.x;
    int i_sum = j;
   /* private for each vector */
    int i_sum_priv = 0;
    for(i=i; i<NI; i+=blockDim.x)
      i_sum_priv += input[k][j][i];
    sbuf[threadIdx.x+threadIdx.y*blockDim.
        x]=i_sum_priv;
    __syncthreads();
    i_sum = reduce_vector(sbuf, j);
    temp[k][j][0] = i_sum;
  }
}
```

(b) Compiler implementation

Figure 5.6: Reduction in vector

5.2.3.1 Reduction in Single-level Thread Parallelism

The loop nest where a reduction operation is applied could be mapped to one or multiple-level thread hierarchy. For example, OpenACC includes three-level of parallelisms: gang, worker, and vector, reduction can appear within any of these levels. Let us first discuss the case where the reduction operation appears in only one level of the parallelism.

Reduction only in vector Figure 5.6 shows an example of reduction occurring only in vector, where the worker and gang loops (k and j) can be executed in parallel, whereas the vector loop (i) needs to perform reduction. There are different strategies to parallelize this case, as seen in Figure 5.9. Figure 5.9(a) shows the data and worker and vector threads laid out in one gang before doing reduction. Each row is one worker that includes multiple vector threads. Since vector reduction happens in each worker, each row needs to do reduction and finally each worker should have one

```
#pragma acc parallel \
            copyin(input) \
            copyout(temp)
{
  #pragma acc loop gang
  for(k=0; k<NK; k++){
    int j_sum = k;
    #pragma acc loop worker \
            reduction(+:j_sum)
    for(j=0; j<NJ; j++){
      #pragma acc loop vector
      for(i=0; i<NI; i++)
        temp[k][j][i]=input[k][j][i];
      j_sum += temp[k][j][0];
    }
    temp[k][0][0] = j_sum;
  }
}
```

(a) OpenACC code

```
k = blockIdx.x;
for(k=k;  k<NK; k+=gridDim.x){
  j = threadIdx.y;
  int j_sum = k;
 /* private for each worker */
  int j_sum_priv = 0;
  for(j=j; j<NJ; j+=blockDim.y){
    i = threadIdx.x;
    for(i=i; i<NI; i+=blockDim.x){
      temp[k][j][i]=input[k][j][i];
    }
    j_sum_priv += temp[k][j][0];
  }
  if(threadIdx.x == 0)
    sbuf[threadIdx.y] = j_sum_priv;
  __syncthreads();
  j_sum = reduce_worker(sbuf, k);
}
```

(b) Compiler implementation

Figure 5.7: Reduction in worker

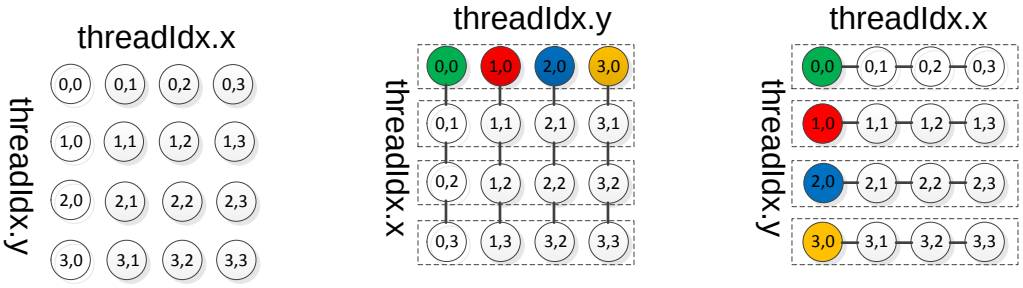
```
sum = 0;
#pragma acc parallel \
            copyin(input) \
            create(temp)
{
  #pragma acc loop gang \
          reduction(+:sum)
  for(k=0; k<NK; k++){
    #pragma acc loop worker
    for(j=0; j<NJ; j++){
      #pragma acc loop vector
      for(i=0; i<NI; i++)
        temp[k][j][i]=input[k][j][i];
    }
    sum += temp[k][0][0];
  }
}
```

(a) OpenACC code

```
k = blockIdx.x;
sum = 0;
/* private for each gang */
int sum_priv = 0;
for(k=k; k<NK; k+=gridDim.x){
  j = threadIdx.y;
  for(j=j; j<NJ; j+=blockDim.y){
    i = threadIdx.x;
    for(i=i; i<NI; i+=blockDim.x){
      temp[k][j][i]=input[k][j][i];
    }
  }
  sum_priv += temp[k][0][0];
}
if(threadIdx.x == 0 &&
   threadIdx.y == 0)
  partial[blockIdx.x]=sum_priv;
```

(b) Compiler implementation

Figure 5.8: Reduction in gang



(a) Data and threads layout in global memory

(b) One type data and threads layout in shared memory

(c) Another type data and threads layout in shared memory

Figure 5.9: Parallelization comparison for vector reduction. (a) includes the original threads layout in a thread block. In (b), each vector thread works on each column data and the reduction results are stored in the first row. In (c), each vector thread works on each row data and the reduction results are stored in the first column. The data inside the dashed rectangle are contiguous in memory.

reduction result. In this example, there are 4 workers and each worker has 4 vector threads, so four vector reduction results should be generated. Since NVIDIA GPU provides very low-latency shared memory, the reduction can be moved to the shared memory to reduce the memory-access latency. We present two different implementation strategies in Figure 5.9(b) and (c). In both these strategies, each vector thread first creates a private variable and does the partial reduction itself and then all the partial private reduction values are stored into the shared memory. But how these data are stored and which thread works on which data can have a significant impact on the performance.

Figure 5.9(b) uses a strategy where the data and the threads layout are transposed in the shared memory, so the reduction in every row of the original thread block becomes the reduction in every column of the thread block and the final four

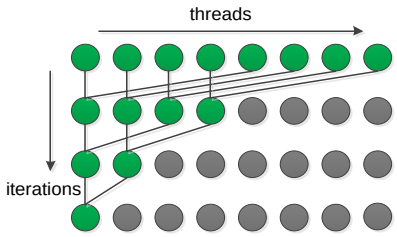
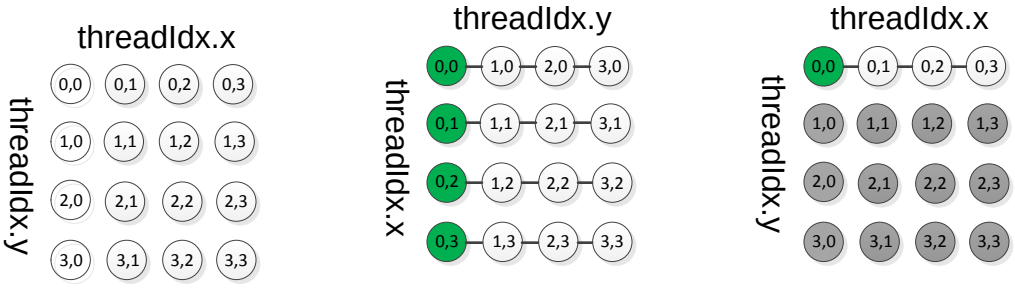


Figure 5.10: Interleaved log-step reduction. Synchronization is inserted after each iteration and before the next iteration. Green represents active threads while the grey represents inactive threads in each iteration.

reduction results are stored in the first row. This approach increases memory divergence since the data that needed to use reduction are not stored contiguously in the shared memory. This is because a warp is the smallest execution unit for GPGPUs and instructions are SIMD-synchronous within a warp.

Figure 5.9(c) shows yet another approach which is implemented in OpenUH. In this approach, the thread layout is the same as the data in the global memory and the layout of the threads working on these data still keep the same. Therefore the vector reduction happens in each row and the final reduction values are stored in the first column of the shared memory. Thus, the data that needs to be reduced are stored contiguously in the shared memory. Although memory divergence may still happen, this issue could be solved by unrolling the last 6 iterations where the reduction data size are 64, 32, 16, 8, 4, and 2. Actually in our implementation, we unroll all iterations since the thread block size is limited to less or equal to 1024 threads in the hardware we are using. Vector size should be a multiple of the warp size.

For both Figure 5.9(b) and (c), the vector threads in the shared memory do the



(a) Data and threads layout in global memory

(b) One type data and threads layout in shared memory

(c) Another type data and threads layout in shared memory

Figure 5.11: Parallelization comparison for worker reduction. In (a), Threads in each row is one worker, so four workers need to do reduction. In (b), four worker values are in every row, so four rows have duplicate values and the final reduction is stored in the first column. In (c), four worker values are only in the first row and following three row threads are inactive, the final reduction is stored in the first element of the first row.

reduction using the interleaved log-step reduction algorithm [36] seen in Figure 5.10.

A point to note is that the initial value of the variable that needs to be reduced may have a different value for the private copy of that variable. For example, the initial value of *i_sum* in Figure 5.6(a) is *j*, but the initial value for the private copy of the variable *i_sum_priv* for each thread is 0 (seen in Figure 5.6(b)). In most of the implementations, the initial value is processed after the vector reduction algorithm is done. For instance, the initial value is summed for + reduction or multiplied for * reduction.

Reduction only in worker Figure 5.7(a) shows an example of reduction occurring only in worker, where the gang and vector loops (k and i) can be executed in parallel, whereas the worker loop (j loop) has to do the reduction. Again we present

two parallelization strategies in Figure 5.11(b) and (c). In both implementations, each worker creates a private variable and does the private partial reduction first, then all the private reduction data computed by all workers will be stored into the shared memory for the final reduction. But the next step they go to different paths.

In Figure 5.11(b), the original vertically layout workers are placed into the shared memory horizontally. That is, the transposed threads work on the transposed data elements in the shared memory. Only the first row in the shared memory contains the useful data while the other rows have duplicate data as the first row. All rows need to do the same interleaved log-step reduction algorithm as vector reduction, final worker reduction results are stored in the first column of the shared memory. Since all rows are the same, actually only the first element of the first row has the useful final reduction result. The advantage of this approach is that the implementation follows the worker reduction concept very strictly. But the disadvantage is that it consumes a lot of shared memory which is a scarce resource in the GPGPU architecture, and it needs to insert synchronization between each iteration in the reduction algorithm because the workers are not consecutive or they are not warp threads.

The OpenUH compiler, however, employs another different strategy seen in Figure 5.11(c). First, the workers also need to do partial reduction by creating private variable. Next the first vector thread of each worker stores the partial reduction into the shared memory, then all the original vector threads of workers use the interleaved log-step reduction algorithm to generate the final reduction result. Note that the threads working on the shared memory data are the same as the threads working on the global memory data, so no transpose happens here. Using this approach requires

less threads and less shared memory so that we can leave more shared memory for other more important computations. For instance, in the example of Figure 5.11(c), only 4 threads are required to do the reduction and only the first row of the shared memory is occupied. Also the advantage of this approach is that the vector threads are warp threads, so we do not need synchronization in the last 6 iterations when only the last warp threads need to the reduction.

Reduction only in gang The example of reduction only in gang is shown in Figure 5.8(a), where the inner worker and vector loops (j and i) are executed in parallel while the gang loop (k loop) has reduction. Since we map each gang to each thread block in CUDA and there is no synchronization mechanism to synchronize all thread blocks, the strategy of OpenUH is to create a temporary buffer (*partial* in Figure 5.8(b)) with the size equal to the number of gangs, each block writes its partial reduction into the specific entry of the buffer, then another kernel (the same reduction kernel as the one in vector addition) is launched to do the reduction within only one block. How to implement the partial reduction within each gang may be different in different compilers. One approach is to divide the iterations all gangs need to compute among all gangs equally, then each gang works on the assigned iterations. OpenUH does not use such blocking algorithm, instead OpenUH considers all gangs as a window, and this window slides through the iteration space. This is similar to the round-robin scheduling algorithm. Essentially there is no difference between blocking algorithm and window-sliding techniques in gang-partial reduction, but the window-sliding technique is superior than blocking algorithm in vector-partial reduction since it can enable memory coalescing. Memory coalescing is impossible in worker- and

gang-partial reduction, but we still use window-sliding technique in both cases in order to make the implementation consistent.

5.2.3.2 Reduction across Multi-level Thread Parallelism (RMP)

Section 5.2.3.1 focused on reduction occurring only in single-level parallelism. Although we discuss each of the cases individually, there could be several combinations of these cases, so some or all of these single cases could be grouped together. For instance, in a triple nested loop, the outermost, the middle and the innermost loops use gang, worker and vector reduction, respectively. Reduction can also occur on different variables within different levels of parallelism. Multiple levels of parallelism can happen within different loops or within the same loop. Next we will discuss all these possibilities in detail.

```
#pragma acc parallel copyin(input) copyout(temp)
{
  #pragma acc loop gang
  for(k=0; k<NK; k++){
    int j_sum = k;
    #pragma acc loop worker reduction(+:j_sum)
    for(j=0; j<NJ; j++){
      #pragma acc loop vector
      for(i=0; i<NI; i++)
        j_sum += input[k][j]i];
    }
    temp[k] = j_sum;
  }
}
```

Figure 5.12: Example of RMP in different loops

RMP in Different Loops Figure 5.12 shows an example of the same reduction spanning across different levels of parallelism in different loops. In this example, the *j_sum* needs to perform reduction on both the worker and vector loops. The CAPS compiler adds the reduction clause to both the worker and vector loops, failing which

incorrect result is generated. This is also a favorable step since reduction occurs in both worker- and vector-level parallelisms. The OpenUH compiler, however, is smarter since it can automatically detect the position of the reduction variable and the user just needs to add the reduction clause to the loop that is the closest to the next use of that reduction variable. In this case, *j_sum* is assigned to *temp[k]* after the worker loop, so we add the reduction in the worker loop. If *j_sum* is used after the vector loop and inside the worker loop, then we add the reduction clause in vector loop. CAPS compiler at times, also just needs to add the reduction clause to the outer most loop, but only when all the inner loops are sequential. With respect to the implementation, OpenUH compiler creates a buffer with the size equal to the number of all threads that needs to do the reduction (workers * vector threads in this example) and the buffer is stored in the shared memory. Each thread writes its own partial reduction result into this buffer and continues the reduction operation in the shared memory. The multi-levels of parallelisms can be of three scenarios: gang & worker, gang & worker & vector, and worker & vector. For the former two cases, a temporary buffer is created and all threads performing reduction operation write their own private reduction into this buffer based on the unique id of each thread. The buffer is allocated in the global memory since the reduction spans across gangs and all gangs do not have the mechanism to synchronize. Another kernel that takes this temporary buffer as input is launched and this kernel performs the vector reduction to generate the final reduction value. Note that the reduction cannot span across gang & vector without going through the worker.

An alternative approach is to perform the reduction in multi-level parallelism

following the order of the parallelisms the reduction appear in. In the example of Figure 5.12, each vector thread performs partial reduction and populates its own private variable, then all vector threads perform the vector reduction with their private reduction values. As a next step, each worker does the partial reduction and populate its own private variable, then all workers do the worker reduction with their private reduction values. Each worker's private reduction value is the reduction value of all vector threads within that worker. The final worker reduction value is the private value for each gang. Since each gang has multiple workers, the final reduction value would be different for each gang. Using this approach, the private variables are different in worker and vector, the vector and worker reduction algorithms could reuse the algorithms discussed in Section 5.2.3.1. This implementation converts the same reduction in different loop levels into different reductions in different loop levels so that the algorithms in Section 5.2.3.1 can be reused. OpenUH does not use this implementation since this approach needs to perform reduction in multiple times and therefore more synchronizations are required.

```
sum = 0;
#pragma acc parallel copyin(input)
{
  #pragma acc loop gang worker vector \
              reduction(+:sum)
  for(i=0; i<NI; i++)
    sum += input[i];
}
```

Figure 5.13: Example of RMP in the same loop

RMP in the Same Loop Figure 5.13 shows an example of reduction across multi-level parallelism in the same loop. In the implementation, OpenUH creates a buffer with the size equal to the size of the all threads that need to reduction (gangs * workers * vector threads in this example), then each thread does its own partial

reduction, and finally launch another kernel to do the reduction for all values in the buffer. Again whether the buffer is stored in global memory or shared memory depends on whether the reduction happens in gang parallelism. As long as gang parallelism is involved, the buffer must be in global memory.

5.2.3.3 Special Reduction Considerations

Apart from the cases listed in Section 5.2.3.1 and Section 5.2.3.2, there are some other special reduction cases. One of them is that the same reduction clause includes multiple-reduction variables and these variables have different data types (e.g. int and float). In this case, one way is to create a large shared memory space and different sections of the shared memory are reserved for different reduction data types. This implementation may face the shared memory size issue since too many reduction variables may required more shared memory than the hardware limit. OpenUH compiler, however, creates a shared-memory space with the size the same as the largest required shared memory for a particular data type. For instance, if there are "int" type reduction and the "double" type reduction in the same reduction clause, then we just need to create a shared memory for the double-type reduction because the required int-type reduction memory space is smaller than the required shared memory for double-type reduction and these two reduction can share the same shared memory space.

We implemented the different cases of reduction operation in both global and shared memory. Although the implementation in global memory is similar to that of the shared memory, the main difference is the memory-access latency. We created

an implementation in the global memory primarily because the shared memory is sometimes reserved for other computation, therefore there is not enough memory space for performing reduction operations. Take the blocked matrix multiplication for example, the matrix is divided into multiple blocks and the computation for each block occurs inside the shared memory. Therefore if a reduction has to happen at the same time, then we would need to move the reduction operation to the global memory.

Another issue is the size of the iteration space and the size of threads. The algorithm in [36] requires that both the iteration space and thread size be power of 2. We remove such a restriction in OpenUH. The restriction of the iteration space size is removed in the algorithm as shown in Figure 5.5, because the threads window slides through the iteration space. Although when the iteration space is not power of 2, there will be some memory divergence within the iterations in the last window. Therefore, the iteration space size in the interleaved log-step reduction algorithm is decided by the thread size rather than the original iteration space size itself. Because the log-step reduction algorithm inherently requires that the iteration space must be power of 2, we need some additional steps before we could consider the algorithm, when the threads size is not power of 2. For instance, if the threads size is 96, first we need to get the previous power-of-2 number 64, then the first 32 threads will do the reduction on the first 32 elements and the last 32 elements. Then the first 32 threads will work on the first 32 elements and the middle 32 elements which is 64 elements which has already satisfied the requirement of the log-step reduction algorithm. The recommended vector threads size is multiple of warp size (32). Although the vector

threads size also could not be multiple of 32, the correctness will not be affected but the performance will degrade significantly.

5.2.4 Performance Evaluation

The experimental platform has 24 Intel Xeon x86_64 cores with 32GB main memory, and an NVIDIA Kepler GPU card (K20c) with 5GB global memory. We use OpenUH compiler to evaluate our OpenACC reduction implementation. For a comparative analysis, we also uses commercial OpenACC compilers CAPS 3.4.0 and PGI 13.10 compilers. CUDA 5.5 is used for all the three compilers. GCC 4.4.7 is used as the host compiler for CAPS compiler. To easily compare the CPU result and GPU result, we disable Fused Multiply Add (FMA) [70]. We use "-O3, -acc -ta=nvidia,cc35,nofma" for the PGI compiler and "–nvcc-options -Xptxas=-v,-arch,sm_35,-fmad=false gcc -O3" for the CAPS compiler. OpenUH compiler uses "-fopenacc" flag to compile the given OpenACC program. And since OpenUH uses a source-to-source technique, CUDA nvcc compiler is used to compile the generated kernel files and the flag passed to nvcc compiler is "-arch=sm_35 -fmad=false". The number of vector size is set to 128 since Kepler architecture has quad warp scheduler that allows to simultaneously issue and execute four warps (32 threads). Threads within a thread block is limited to 1024 threads due to which the number of workers is set to 8. All thread blocks are scheduled on all streaming multiprocessors (SMs). Kepler has 13 SMs and one of them is likely to be disabled [27], also each SM can support at most 16 thread blocks. To keep all SMs busy we choose the number of gangs to be 12x16=192. We can set the gang, worker, and vector using `num_gangs`, `num_workers`, and `vector_length`

clauses in OpenACC.

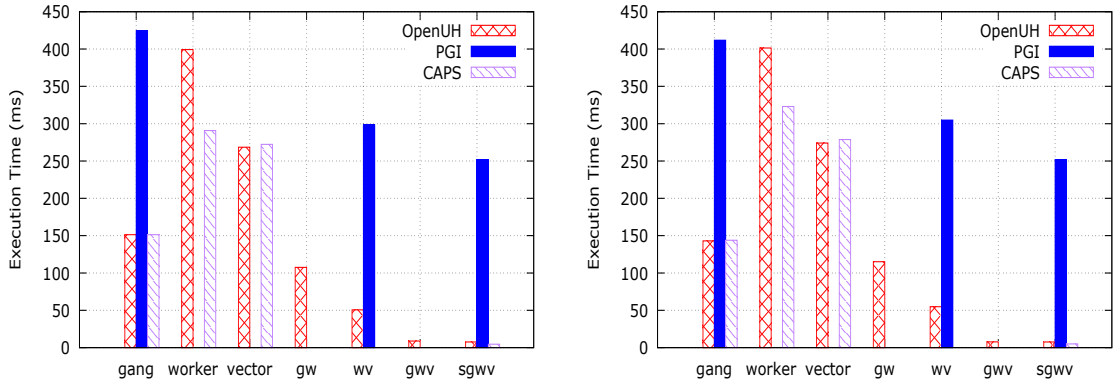
Since there are no existing benchmarks that could cover all the reduction cases, we have designed and implemented a test suite to validate all possible cases of reduction including different reduction data types and reduction operations. The test suite will check if a given reduction implementation passed or failed by verifying the OpenACC result with the CPU result. If the values do not match, it implies there is an implementation issue. We also measure the execution time of each of the reduction cases, so if the compilers under evaluation can pass the test, we compare their performances too. For all the test cases, we perform reduction using OpenACC first and then on the CPU side, after which we compare if their results are the same. When reduction occurs in one of the levels of parallelism, the other levels of parallelisms has instructions being executed in parallel. Only the RMP in the same loop uses one loop, the other reduction tests use triple nested loop. When one loop level needs to do reduction, that loop iteration size is up to 1M and the other two loops are 2 and 32 because of the memory limit of the hardware. Although we used triple nested loop in experiments, the user can use `collapse` clause in OpenACC if the loop level is more than three. We discuss the results of the most commonly used reduction operators "+" and "*"; the implementation of other reduction operators are almost the same. We also use a real world application to demonstrate "max" reduction intrinsic.

Table 5.2 discusses the performance results of OpenUH, PGI, and CAPS compilers while using the reduction test suite. We see that only OpenUH compiler passed all of the reduction tests. CAPS compiler failed some of the tests of RMP in different

loops. PGI compiler failed the summation reduction in worker, vector, and RMP in gang & worker. It even failed to compile the RMP in gang & worker & vector. Figure 5.14 shows the performance comparison of the three compilers. It is observed that in gang or vector reduction, the performance of OpenUH compiler was more or less the same as the CAPS compiler, and only in worker reduction it was slightly less efficient than CAPS compiler. The performance of OpenUH was better than PGI compiler for all reduction cases. We could not dive deeper into the analysis for the obvious reason that CAPS and PGI are commercial compilers and we do not have access to their underlying implementation details. Although the execution time here was only several hundred milliseconds, it can still have an impact on a real-world application. We discuss this later.

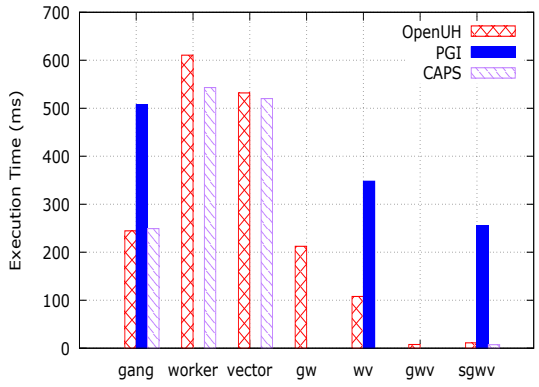
Although the test suite only included the reduction cases in triple nested loop and one loop, they can be used in any levels of the loop. Apart from the test suite, we also used some real-world benchmark applications: 2D-Heat Equation, matrix multiplication, and Monte Carlo PI. Let us look into the evaluation details.

2D-Heat Equation is a type of stencil computation. The formula to represent the 2D-heat equation is explained in [59]. In this application, there is a grid that has boundary points and inner points. Boundary points have an initial temperature and the temperature of the inner points need to be updated over iterations. Each inner point updates its neighboring points and itself. The temperature updatin!htbp operation for all inner points needs to last long enough to obtain the final stable temperature. We added the temperature converge code in [59] so that we can know when the convergence happens. The temperature was stable when the maximum

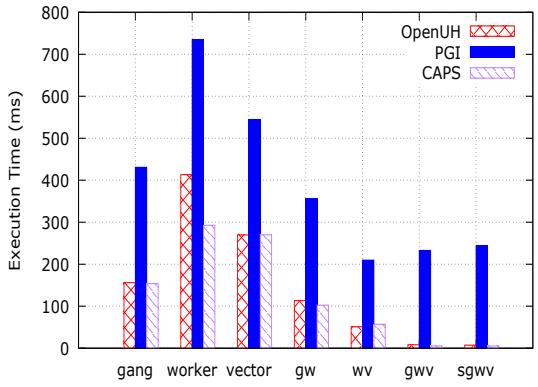


(a) Int [+]

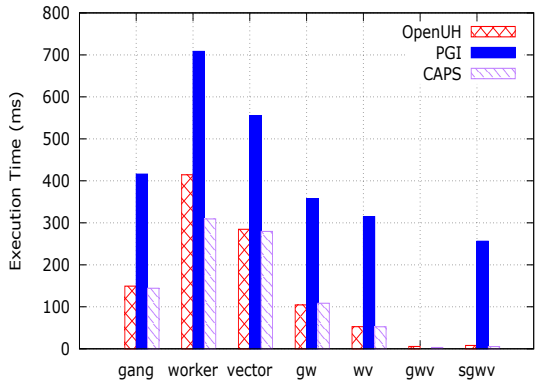
(b) Float [+]



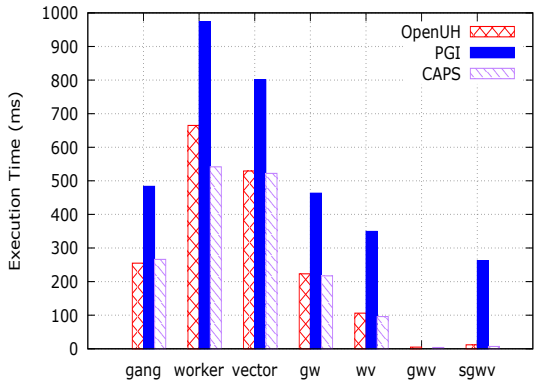
(c) Double [+]



(d) Int [*]



(e) Float [*]



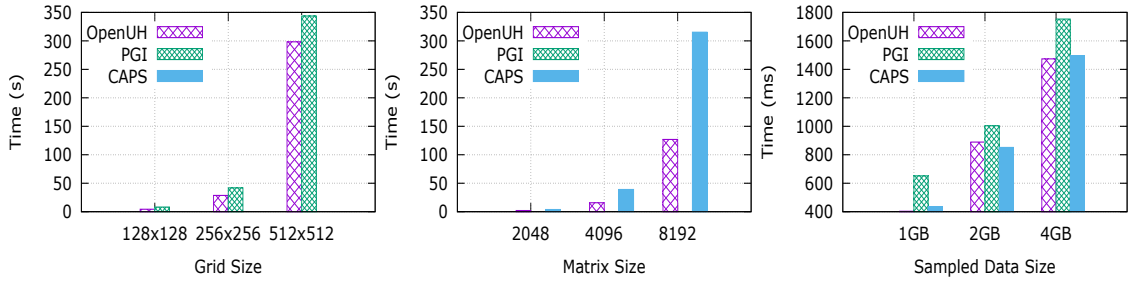
(f) Double [*]

Figure 5.14: Performance comparison of OpenACC compilers using reduction test suite. Missing bars imply that the test failed. The symbol in square brackets indicates the reduction operator. gw: gang worker; wv: worker vector; gwv: gang worker vector; sgwv: same line gang worker vector

Table 5.2: Performance results of OpenACC compilers using reduction test suite. Time is in milliseconds. "F" stands for test FAILED, and "CE" stands for compile time error. Except the "same line gang worker vector" uses only one loop, the other reduction tests used triple nested loop where the outermost, middle, and the innermost loops are gang, worker, and vector, respectively. The first column implies the reduction position. For instance, "gang worker" means gang and worker loops need to do reduction while the vector loop executes in parallel.

Reduction Position	Reduction Operator	Data Type								
		Int			Float			Double		
		OpenUH	PGI	CAPS	OpenUH	PGI	CAPS	OpenUH	PGI	CAPS
gang	+	151.27	424.64	151.29	142.99	411.76	143.78	244.61	507.02	249.13
	*	156.01	430.77	153.92	249.27	415.91	144.12	254.90	483.59	266.09
worker	+	399.25	F	290.83	401.33	F	322.97	610.61	F	543.20
	*	413.35	734.35	292.86	414.50	708.29	309.25	664.87	973.88	541.80
vector	+	268.47	F	272.32	274.06	F	278.74	532.01	F	520.14
	*	269.80	544.71	269.98	284.68	555.58	279.58	529.37	800.89	522.11
gang worker	+	107.49	F	F	115.10	F	F	212.31	F	F
	*	113.36	356.00	102.50	104.44	357.50	108.53	223.30	463.34	217.15
worker vector	+	50.58	298.33	F	54.85	304.82	F	107.75	347.90	F
	*	51.20	209.72	56.82	52.75	314.60	52.46	105.97	349.44	95.78
gang worker vector	+	8.77	CE	F	7.66	CE	F	7.65	CE	F
	*	8.15	232.84	5.50	5.61	CE	3.09	4.87	CE	3.82
same line gang worker vector	+	7.55	251.67	4.60	7.57	251.98	4.94	11.24	255.12	7.26
	*	7.25	243.63	5.21	7.86	256.18	5.361	11.90	262.49	6.90

temperature difference for all data points in the grid between two consecutive iterations gradually decreased from a large value to 0. So in every iteration, the program needed to compute the maximum difference for all data points in the current iteration and all data points in the previous iteration, which was a max reduction in OpenACC. The code snippet is seen in Figure 5.16 (a), where *temp1* is the temperature in the previous iteration and *temp2* is the temperature in the current iteration. All data points in the grid are traversed to get the maximum error. We have prior experience working on the parallelization of the temperature updating kernel [74], but in this chapter, we only focus on the maximum reduction. Figure 5.15 (a) shows



(a) 2D-Heat Equation [max] (b) Matrix Multiplication [+] (c) Monte Carlo PI [+]

Figure 5.15: Performance comparison for three applications. CAPS bar in (a) and PGI bar in (b) are missing because they failed. The symbol in square brackets is the reduction operator.

```
#pragma acc loop gang
reduction(max:error)
for (j=1; j < nj-1; j++)
{
 #pragma acc loop vector
 for (i=1; i < ni-1; i++)
     {
  i00 = j*ni + i;
  error = fmax(error,
  fabs(temp1[i00] - temp2[
      i00]));
 }
}
```

(a) 2D-Heat Equation

```
#pragma acc loop gang
for (i = 0; i < n; i++){
 #pragma acc loop worker
 for (j = 0; j < n; j++){
   c = 0.0;
   #pragma acc loop vector
        reduction(+:c)
   for (k = 0; k < n; k++)
     c+=A[i*n+k]*B[k*n +j
         ];
   C[i*n+j] = c;
 }
}
```

(b) Matrix Multiplication

```
#pragma acc loop gang
    vector reduction(+:m)
for(i=0; i<n; i++)
{
//x[i]=2.0*rand()/(
    RAND_MAX+1.0)-1.0;
//y[i]=2.0*rand()/(
    RAND_MAX+1.0)-1.0;
    if(x[i]*x[i] + y[i]*y[
        i] < 1.0)
        m++;
}
```

(c) Monte Carlo PI

Figure 5.16: Code snippet for three applications

the performance comparison between OpenUH and other two compilers. The performance of the CAPS compiler is missing since the temperature difference generated by this compiler increases gradually rather than a decrease, so the application can never converge. We increase the grid size from 128x128 to 512x512 and we find that OpenUH compiler is always better than PGI compiler. This demonstrates that the performance of the reduction implementation will accumulate in an iterative algorithm; we do not observe the signicant performance improvement in only one

iteration.

Matrix Multiplication is a classic example in parallel programming. We consider a naive matrix-multiplication case. Most developers usually only parallelize the outer two loops and lets the third loop sequentially execute since the third loop has data-dependence. However we can also parallelize the third loop because essentially it just includes the "sum" reduction operations. The code snippet is seen in Figure 5.16 (b) and the performance comparison in seen in Figure 5.15 (b). Different matrix sizes are chosen and the result shows that the performance of OpenUH is more than 2x better than CAPS compiler. Since the reduction here happens only in vector while PGI compiler failed, the vector reduction is seen in Figure 5.14 (c), the PGI performance bar is not shown.

Monte Carlo PI is another example of using reduction. PI (π) can be computed in different ways and one of them is to use the Monte Carlo statistical method. Given a circle of the radis 1 is inscribed inside a square with side length 2, then the area of the circle and the square are π and 4, respectively. Therefore the ratio of the area of the circle to the area of the square (ρ) is $\pi/4$. The program picks points within the square randomly and check whether the point is also inside the circle. This can be determined by the formula $x^2 + y^2 < 1$, where x and y are the coordinates of the point. Assume the number of data points within the circle is m and the number of data points within the square is n, then $\rho = m/n$ and we can obtain $\pi = 4.0 * m/n$. The more data points sampled within the square, the more accurate the value of π can be obtained. In the program, n is the total number of iterations of a loop and inside the loop the coordinates of a data point x and y are randomly generated

by calling rand() in C, and then checked whether $x^2 + y^2 < 1$. If yes, then m is increased by 1. Therefore the computation of m is actually a reduction operation. Since at the time of writing most compilers do not support function call inside an OpenACC kernel region , we pre-generated the x and y values on the host and then transferred them to the device to for m reducion. The code is in Figure 5.16 (c) where the loop is one level and the computation is distributed to gang and vector threads. For more accurate PI value, we try to sample as many points as we can. In our Kepler architecture, the maximum global memory is 5 GB, so we use different sampled data size 1 GB, 2 GB and 4 GB memory for this application. The results comparison among different compilers is shown in Figure 5.15 (c). It is observed that the performance of OpenUH is slightly better than CAPS compiler and much better than PGI compiler. This result is consistent with the performance difference while using the reduction test suite, although we just used gang and vector in one loop instead of using gang, worker and vector in the test suite.

5.3 Summary

In this chapter, we presented the efficient runtime design which shows how to manage the execution context, data management, and kernel launch in the runtime. We also present all possible reduction cases in OpenACC programming model, and the underlying parallelization implementations in an open-source OpenACC compiler OpenUH. We evaluated our implementation with a self-written reduction test suite

and three real-world applications. We observed competitive performance while comparing OpenUH with the other two commercial OpenACC compilers. Unlike one of the commercial compilers that needed to add the reduction clause in multiple-level parallelism, OpenUH could intelligently detect the position where the reduction had to occur and the user was only required to add the reduction clause once. A similar reduction methodology can also be applied to other programming models such as OpenMP 4.0. OpenMP demonstrates two levels of parallelism and it just needs to ignore the worker if our implementation strategy is used.

Chapter 6

Optimizations for Multiple GPUs

The previous chapters have demonstrated that using a single GPU can lead to obtaining significant performance gains. In this chapter, we discuss how to achieve further performance speedup if we use more than one GPU. Heterogeneous processors consisting of multiple CPUs and GPUs offer immense potential and are often considered as a leading candidate for porting complex scientific applications. Unfortunately programming heterogeneous systems requires more effort than what is required for traditional multicore systems. Directive-based programming approaches are being widely adopted since they make it easy to use/port/maintain application code. OpenMP and OpenACC are two popular models used to port applications to accelerators. However, neither of the models provides support for multiple GPUs. A plausible solution is to use combination of OpenMP and OpenACC that forms a hybrid model [73]; however, building this model has its own limitations due to a lack of necessary compilers' support. Moreover, the model also lacks support for

direct device-to-device communication. To overcome these limitations, an alternate strategy is to extend OpenACC by proposing and developing extensions that follow a task-based implementation for supporting multiple GPUs [78]. We critically analyze the applicability of the hybrid model approach and evaluate the proposed strategy using several case studies and demonstrate their effectiveness.

6.1 Related Work

A single programming model is insufficient to provide support for multiple accelerator devices; however, the user can apply hybrid model to achieve this goal. Hart et al. [38] used Co-Array Fortran (CAF) and OpenACC and Levesque et al. [51] used MPI and OpenACC to program multiple GPUs in a GPU cluster. The inter-GPU communication in these work are managed by the Distributed Shared Memory (DSM) model CAF or distributed-memory model MPI. Unlike these work, our work uses OpenMP and OpenACC hybrid model. We also developed an approach that extends the OpenACC model to support multiple devices in a single cluster node.

Other work that targets multiple devices include OmpSs [22] that allows the user to use their own unique directives in an application so that the program can run on multiple GPUs either on the shared-memory system or distributed-memory system. StarPU [16] is a runtime library that schedules tasks to multiple accelerators. However, the drawback of these approaches is that both OmpSs and StarPU require the user to manually write the kernel which is going to be offloaded to the accelerators. Moreover their approach is not part of any standard thus limiting the usability.

In this chapter, we have adopted a task-based concept by proposing extensions to the OpenACC model to support multiple accelerators. Related work that uses tasking concept for GPUs include Chatterjee et al. [25] who designed a runtime system that can schedule tasks onto different Stream Multiprocessors (SMs) in one device. In their system, at a specific time, the device can only execute the same number of thread blocks as the number of SMs (13 in Kepler 20c) thus limiting the performance. This is because their system was designed for tackling load balancing issues among all SMs primarily for irregular applications. Extensions to the OpenACC model was proposed by Komoda et al. [45] to support multiple GPUs. They proposed directives for the user to specify memory access pattern for each data in a compute region and the compiler identifies the workload partition. It is complicated if the user has to identify the memory-access pattern for all data, especially when the data is multi-dimensional or accesses multiple-index variables. They also did not explain clearly about how the data was managed when device-to-device communication is required. In our extensions to the OpenACC model, we allow the user to partition the workload thus further simplifying the application porting which makes the multi-GPU support general enough to cover most application cases.

6.2 Multi-GPU with OpenMP & OpenACC Hybrid Model

In this section, we will discuss strategies to explore programming multi-GPU using OpenMP & OpenACC hybrid model within a single node. We will evaluate our

strategies using three scientific applications. We study the impact of our approach by comparing and analyzing the performances achieved by the hybrid model (multi-GPU implementation) against that of a single GPU implementation.

The experimental platform is a multi-core server consisting of two NVIDIA Kepler 20Xm GPUs. The system itself has Intel Xeon x86_64 CPU with 24 cores (12 x 2 sockets), 2.5 GHz frequency and 62 GB main memory. Each GPU has 5 GB global memory. We used CAPS OpenACC for S3D and PGI OpenACC for matrix multiplication and 2D-heat equation. PGI compiler was not used for S3D since it did not compile successfully. The 2D-heat equation program compiled by CAPS compiler was extremely long so we do not show the result here. CAPS compiler does compile the matrix multiplication program but we leave till later the performance comparison with other compilers and only verify the feasibility of hybrid programming model. We used GCC 4.4.7 as CAPS host compiler for all C programs. For a Fortran program, we used PGI and CAPS (pgfortran as the host compiler of CAPS) to compile the OpenACC code. We used the latest versions of CAPS and PGI compilers; 3.4.1 and 14.2 respectively. CUDA 5.5 was used for our experiments. The CAPS compiler performs source-to-source translation of directives inserted code into CUDA code, and then calls *nvcc* to compile the generated CUDA code. The flags passed to CAPS compiler were "–nvcc-options -Xptxas=-v,-arch,sm_35,-fmad=false", and the flags passed to PGI compiler were "-O3 -mp -ta=nvidia,cc35,nofma". We consider wall-clock time as the evaluation measurement. We ran all experiments five times and then averaged the performance results. In the forthcoming subsections, we will discuss both single and multi-GPU implementations for the S3D Thermodynamics

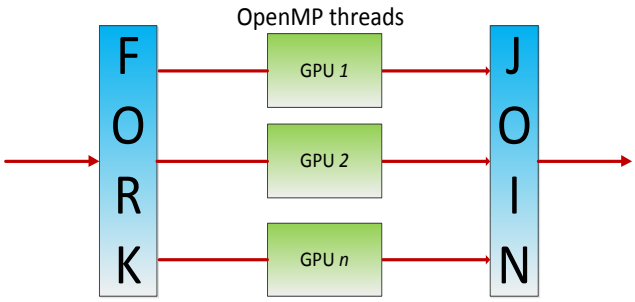


Figure 6.1: A multi-GPU solution using the hybrid OpenMP & OpenACC model. Each OpenMP thread is associated with one GPU

application kernel, matrix multiplication and 2D-heat equation.

OpenMP was fairly easy to use, since all that the programmer needs to do is to insert OpenMP directives in the appropriate places and if necessary, make minor modifications to the code. The general idea of an OpenMP & OpenACC hybrid model, as seen in Figure 6.1, is that we need to manually divide the problem among OpenMP threads, and then associate each thread to a particular GPU. The best case scenario is when the work in each GPU is independent of each other and does not involve communication among GPUs. But there may be cases where the GPUs will have to communicate with each other and this will involve the CPUs too. Different GPUs transfer their data to their corresponding host threads, these threads then communicate or exchange their data via shared variable, and finally the threads transfer the new data back to their associated GPUs. With the GPU Direct technique [9], it is also possible to transfer data between different GPUs directly without going through the host. This has not been plausible in OpenMP & OpenACC hybrid model so far, but in Section 6.3 we will propose some extensions to the OpenACC programming model to accommodate this feature.

6.2.1 S3D Thermodynamics Kernel

S3D [26] is a flow solver that performs direct numerical simulation of turbulent combustion. S3D solves fully compressible Navier-Stokes, total energy, species, and mass conservation equations coupled with detailed chemistry. Apart from the governing equations, there are additional constitutive relations, such as the ideal gas equation of state, models for chemical reaction rates, molecular transport, and thermodynamic properties. These relations and detailed chemical properties have been implemented as kernels or libraries suitable for GPU computing. Some research on S3D have been done in [62] [51], but the code used is not accessible. For the experimental purpose of our work, we only chose two separate and independent kernels of the large S3D application, discussed in detail in [39].

```
!$acc data copyout(c(1:np), h(1:np)) copyin(T(1:np),...)
do m = 1, MR
  call calc_mixcp(np, nslvs, T, midtemp, ... , c)
  call calc_mixenth(np, nslvs, T, midtemp, ... , h)
end do
!$acc end data
```

Figure 6.2: S3D thermodynamics kernel in single GPU

We observed that the two kernels of S3D have similar code structures and their input data are common. Figure 6.2 shows a code snippet of a single GPU implementation. Both the kernels, *calc_mixcp and calc_mixenth* are surrounded by a main loop with *MR* iterations. Each kernel produces its own output result, but their results are the same as that of the previous iteration. The two kernels can be executed in the same accelerator sequentially while sharing the common input data, which will stay on the GPU during the whole execution time. Alternatively, they can be

also executed on different GPUs simultaneously since they are totally independent kernels.

In order to use multi-GPU, we distribute the kernels to two OpenMP threads and associate each thread to one GPU. Since we have only two kernels, it is not necessary to use *omp for*, instead we use *omp sections* so that each kernel is located in one section. Each thread needs to set the device number using the runtime *acc_set_device_num(int devicenum, acc_device_t devicetype)*. Note that the device number starts from 1 in OpenACC, or the runtime behavior would be implementation-defined if the *devicenum* were to start from 0. To avoid setting the device number in each iteration and make the two kernels work independently, we apply loop fission and split the original loop into two loops. Finally, we replicate common data on both the GPUs. The code snippet in Figure 6.3 shows the implementation for multi-GPU. Although it is a multi-GPU implementation, the implementation in each kernel is still the same as that of a single GPU implementation. Figure 6.4 shows the performance results of using single GPU and two GPUs. It was observed, two GPUs almost always takes approximately half the time taken for a single GPU. This illustrates the performance advantage of using multiple GPUs over single GPU.

6.2.2 Matrix Multiplication

With S3D, we had distributed different kernels of one application to multiple GPUs. An alternate type of a case study would be where the workload of only one kernel is distributed to multiple GPUs, especially if the workload is very large. We will use

```
call omp_set_num_threads(2)
!$omp parallel private(m)
!$omp sections
!$omp section
call acc_set_device_num(1, acc_device_not_host)
!$acc data copyout(c(1:np)) copyin(T(1:np),...)
do m = 1, MR
  call calc_mixcp(np, nslvs, T, ... , c)
end do
!$acc end data
!$omp section
call acc_set_device_num(2, acc_device_not_host)
!$acc data copyout(h(1:np)) copyin(T(1:np),...)
do m = 1, MR
  call calc_mixenth(np, nslvs, T, ... , h)
end do
!$acc end data
!$omp end sections
!$omp end parallel
```

Figure 6.3: S3D thermodynamics kernel in multi-GPU using hybrid model

square-matrix multiplication as an illustration to explore this case study. We chose this application since this kernel has been extensively used in numerous scientific applications. This kernel does not comprise of complicated data movements and can be parallelized by simply distributing work to different thread. We also noticed a large computation to data movement ratio.

Typically matrix multiplication requires matrix A and matrix B as input, and produces matrix C as the output. When multiple GPUs are used, we will use the same number of threads as the number of GPUs on the host. For instance, if the system has 2 GPUs, then we will launch 2 host threads. Then we partition matrix A in block row-wise which means that each thread will obtain partial rows of matrix A. Every thread needs to read the whole matrix B and produce the corresponding partial result of matrix C. After partitioning the matrix, we use OpenACC to execute the computation of each partitioned segment on one GPU.

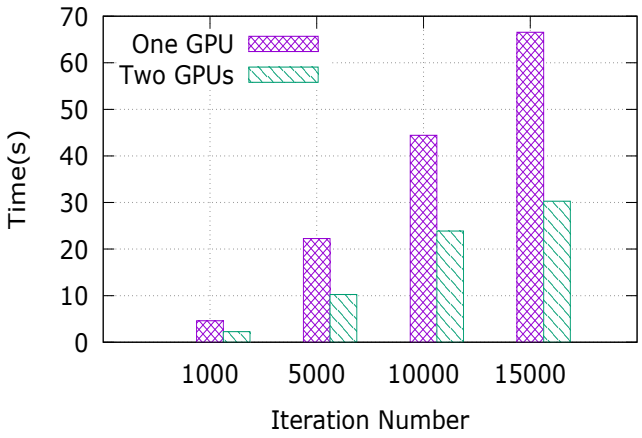


Figure 6.4: Performance comparison of S3D

Figure 6.5 shows a code snippet for the multi-GPU implementation for matrix multiplication. Here we assumed that the number of threads could be evenly divided by the square matrix row size. Since the outer two loops are totally independent, we distribute the i loop into all gangs and the j loop into all vector threads of one gang. We have used only 2 GPUs for this experiment, however more than 2 GPUs can be easily used as long as they are available in the platform. In this implementation, we assumed that the number of GPUs can be evenly divided by the number of threads. We used different workload sizes for our experiments. The matrix dimension ranges from 1024 to 8192. Figure 6.6 (a) shows the performance comparison while using one and two GPUs. For all data size except 1024, the execution time with 2 GPUs was almost half of that with only 1 GPU. For 1024*1024 as the matrix size, we barely saw any benefit using multiple GPUs. This is possibly due to the overhead incurred due to the creation of host threads and GPU context setup. Moreover the computation was not large enough for two GPUs. When the problem size was more than 1024, the multi-GPU implementation showed a significant increase in performance. In

these cases, the computation was so intensive that the aforementioned overheads were being ignored.

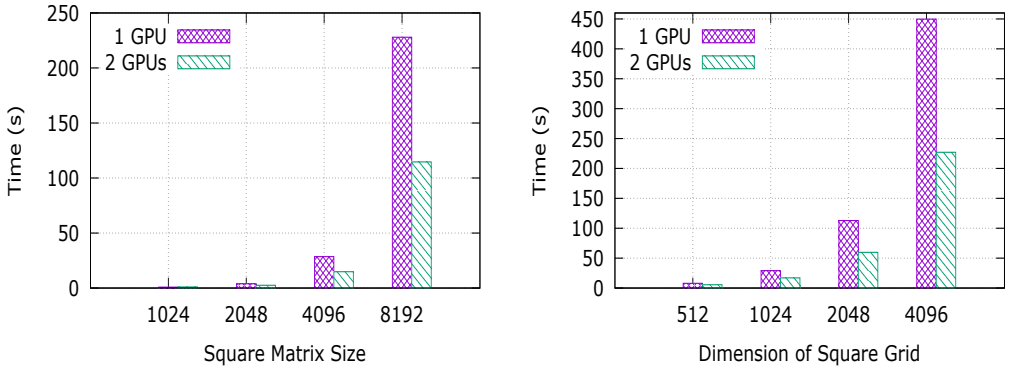
```
omp_set_num_threads(threads);
#pragma omp parallel
{
    int i, j, k;
    int id, blocks, start, end;
    id = omp_get_thread_num();
    blocks = n/threads;
    start = id*blocks;
    end = (id+1)*blocks;

    acc_set_device_num(id+1, acc_device_not_host);
    #pragma acc data copyin(A[start*n:blocks*n])\
                     copyin(B[0:n*n])\
                     copyout(C[start*n:blocks*n])
    {
      #pragma acc parallel num_gangs(32) vector_length(32)
       {
       #pragma acc loop gang
       for(i=start; i<end; i++){
          #pragma acc loop vector
          for(j=0; j<n; j++){
            float c = 0.0f;
            for(k=0; k<n; k++)
                c += A[i*n+k] * B[k*n+j];
            C[i*n+j] = c;
          }
       }
    }
}
```

Figure 6.5: A multi-GPU implementation of MM using hybrid model

6.2.3 2D-Heat Equation

We notice that in the previous two cases, the kernel on one GPU is completely independent of the kernel on the other GPU. Now we will explore a case where there is communication between different GPUs. One such interesting application is 2D-heat equation. The formula to represent 2D-heat equation is explained in [59] and



(a) matrix multiplication (b) 2D-heat equation

Figure 6.6: Performance comparison using hybrid model

is given as follows:

$$\frac{\partial T}{\partial t} = \alpha(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2})$$

where T is temperature, t is time, α is the thermal diffusivity, and x and y are points in a grid. To solve this problem, one possible finite difference approximation is:

$$\frac{\Delta T}{\Delta t} = \alpha[\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}]$$

where ΔT is the temperature change over time Δt and i, j are indexes in a grid. In this application, there is a grid that has boundary points and inner points. Boundary points have an initial temperature and the temperature of the inner points are also updated. Each inner point updates its temperature by using the previous temperature of its neighboring points and itself. The operation that updates temperature for all inner points in a grid needs to last long enough. This implies that many iterations are needed before arriving at the final stable temperatures. In our program, the number of iterations is 20,000, and we increase the grid size gradually from 512*512 to 4096*4096. Our prior experience working on single GPU implementation of 2D-heat

equation is discussed in [74]. Figure 6.7 shows the code snippet for the single GPU implementation. Inside the kernel that updates the temperature, we distribute the outer loop into all gangs and the inner loop into all vector threads inside each gang. Since the final output will be stored in *temp1* after pointer swapping, we just need to transfer this data out to host.

Let us discuss the case where the application uses two GPUs. Figure 6.8 shows the program in detail. In this implementation, *ni* and *nj* are X and Y dimension of the grid (does not include boundary), respectively. As seen in Figure 6.9, we partitioned the grid into two parts along Y dimension and run each part on one GPU. Before the computation, the initial temperature is stored in *temp1_h*, and after updating the temperature, the new temperature is stored in *temp2_h*. Then we swap the pointer so that in the next iteration the input of the kernel points to the current new temperature. Since updating each data point needs its neighboring points from the previous iteration, two GPUs need to exchange the halo data in every iteration. The halo data is referred to the data that needs to be exchanged by different GPUs. So far by simply using high-level directives or runtime libraries, data cannot be exchanged directly between different GPUs and the only workaround is to first transfer the data from one device to the host and then from the host to another device. In 2D-heat equation, different devices need to exchange the halo data, therefore the halo-data updating would go through the CPU. Because different GPUs use different parts of the data in the grid, we do not have to allocate separate memory for these partial data, instead we just need to use private pointer to point to the different positions of the shared variable *temp1_h* and *temp2_h*. Let *tid* represents the id of a thread,

then that thread points to the position $tid * rows * (ni + 2)$ of the grid (because it needs to include the halo region) and it needs to transfer $(rows + 2) * (ni + 2)$ data to the device where rows equals to $nj/NUM\_THREADS$. The kernel that updates the temperature in the multi-GPU implementation is exactly the same as the one in single GPU implementation.

```
void step_kernel{...}
{
  #pragma acc parallel present(temp_in[0:ni*nj], temp_out[0:ni*nj]) \
                       num_gangs(32) vector_length(32)
  {
    // loop over all points in domain (except boundary)
    #pragma acc loop gang
    for (j=1; j < nj-1; j++) {
        #pragma acc loop vector
        for (i=1; i < ni-1; i++) {
            // find indices into linear memory
            // for central point and neighbours
            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);

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

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

#pragma acc data copy(temp1[0:ni*nj]) copyin(temp2[0:ni*nj])
{
    for (istep=0; istep < nstep; istep++) {
        step_kernel(ni, nj, tfac, temp1, temp2);
        // swap the temp pointers
        temp = temp1; temp1 = temp2; temp2 = temp;
    }
}
```

Figure 6.7: Single GPU implementation of 2D-Heat Equation

Figure 6.6(b) shows the performance comparison of the different implementations,

```
omp_set_num_threads(NUM_THREADS);
rows = nj/NUM_THREADS;
LDA = ni + 2;
// main iteration loop
#pragma omp parallel private(istep)
{
  float *temp1, *temp2, *temp_tmp;
  int tid = omp_get_thread_num();
  acc_set_device_num(tid+1, acc_device_not_host);

  temp1 = temp1_h + tid*rows*LDA;
  temp2 = temp2_h + tid*rows*LDA;

  #pragma acc data copyin(temp1[0:(rows+2)*LDA]) \
                   copyin(temp2[0:(rows+2)*LDA])
  {
    for(istep=0; istep < nstep; istep++){
      step_kernel(ni+2, rows+2, tfac, temp1, temp2);

      /* all devices (except the last one) update the lower halo to the host */
      if(tid != NUM_THREADS-1){
        #pragma acc update host(temp2[rows*LDA:LDA])
      }
      /* all devices (except the first one) update the upper halo to the host */
      if(tid != 0){
        #pragma acc update host(temp2[LDA:LDA])
      }
      /* all host threads wait here to make sure halo data from all devices
         have been updated to the host */
      #pragma omp barrier

      /* update the upper halo to all devices (except the first one) */
      if(tid != 0){
        #pragma acc update device(temp2[0:LDA])
      }
      /* update the lower halo to all devices (except the last one) */
      if(tid != NUM_THREADS-1){
        #pragma acc update device(temp2[(rows+1)*LDA:LDA])
      }
      temp_tmp = temp1;
      temp1 = temp2;
      temp2 = temp_tmp;
    }
    /*update the final result to host*/
    #pragma acc update host(temp1[LDA:row*LDA])
  }
}
```

Figure 6.8: Multi-GPU implementation with hybrid model - 2D-Heat Equation

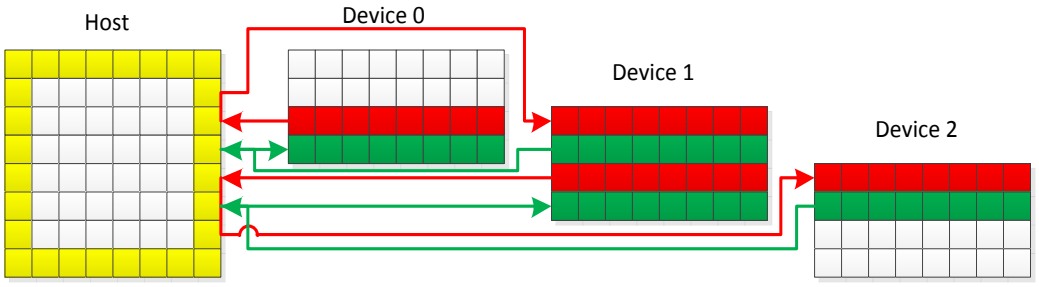


Figure 6.9: Multi-GPU implementation strategy for 2D-Heat Equation using the hybrid model. Consider there are 3 GPUs (Device 0, 1, and 2). The grid in the left has 6 rows (excluding boundaries, i.e. the top and the bottom rows). By splitting the 6 rows into 3 parts, each GPU is expected to compute only 2 rows. However, the computation for a data point requires the value of the neighboring points (top, bottom, left, and right data points), hence simply considering 2 rows of the grid for 1 GPU is not enough. For GPU Device 0, the last row added already has the left, top, and right data points, but lacks data points from the bottom, hence the bottom row needs to be added, leading to 3 rows in total. For GPU Device 1, the first and the second rows do not have data points from the top and the bottom, respectively, hence requiring an addition of the top and bottom rows. This leads to 4 rows in total. For GPU Device 2, the first row does not have data points from the top, requires the addition of the top row. This leads to 3 rows in total. Another point to note is that, values in the rows added need to be updated from other GPUs as indicated by the arrows.

i.e. single and multi-GPU implementations. While comparing the performances of multi-GPU with single GPU, we noticed that there is a trivial performance difference when the problem size is small. However, there is a significant increase in performance using multi-GPU for larger grid sizes. With the grid size as 4096*4096, the speedup of using two GPUs is around 2x times faster than the single GPU implementation. This is because as the grid size increases, the computation also increases significantly, while the halo-data exchange is still small enough. Thus the ratio of the computation/communication becomes larger. Using multi-GPU can be quite advantageous to decompose the computation.

6.3 Multi-GPU with OpenACC Extension

We see that programming using multi-GPU using OpenMP & OpenACC hybrid model shows significant performance benefits in Section 6.2. However there are some disadvantages too in this approach. First, the users need to learn two different programming languages which may impact the productivity. Second, in this approach the device-to-device communication happens via the host bringing more unnecessary data movement. Third, providing support for such hybrid model is a challenge for compilers. Compiler A provides support for OpenMP and Compiler B provides support for OpenACC, as a result it is not straight forward for different compilers to interact with each other. For instance, a Cray compiler does not allow OpenACC directives to appear inside OpenMP directives [2], therefore the examples in Figure 6.5 and Figure 6.8 are not compilable by the Cray compiler. CAPS compiler although

provides support for OpenACC, still uses an OpenMP implementation from another host compiler; this is also a challenge to follow and maintain. Ideally, one programming model should provide support for multi-GPU. Unfortunately the existing OpenACC standard does not yet provide support for multiple accelerator programming. To solve these problems, we propose some extensions to the OpenACC standard in order to support multiple accelerator devices.

6.3.1 Proposed Directive Extensions

The goal is to help the compiler or runtime realize which device the host will communicate with, so that the host can issue the data movement and kernel launch request to the specific device. The new extensions are described as follows:

(1) `#pragma acc parallel/kernels deviceid` (*scalar-integer-expression*)

This is to place the corresponding compute region into a specific device.

(2) `#pragma acc data deviceid` (*scalar-integer-expression*)

This is the data directive extension for the structured data region.

(3) `#pragma acc enter/exit data deviceid` (*scalar-integer-expression*)

This is the extension for unstructured data region.

(4) `#pragma acc wait device` (*scalar-integer-expression*)

This is used to synchronize all activities in a particular device since by default the execution in each device is asynchronous when multiple devices are used.

(5) `#pragma acc update peer to` (*list*) `to_device` (*scalar-integer-expression*) `from` (*list*) `from_device`(*scalar-integer-expression*)

(6) void `acc_update_peer(void* dst, int to_device, const void* src,`

`int from_device, size_t size)`

The purpose of (5) and (6) is to enable device-to-device communication. This is particularly important when using multiple devices, since in some accelerators device can communicate directly with another device without going through the host. If the devices cannot communicate directly, the runtime library can choose to first transfer the data to a temporary buffer in the host, then transfer it from the host to another device. For example, in CUDA implementation, two devices can communicate directly only when they are connected to the same root I/O Hub. If this requirement is not satisfied, then the data transfer will go through the host. (Note: We believe these extensions will address direct device-to-device communication challenge, however it also requires necessary support from the hardware. Our evaluation platform did not fulfill the hardware needs, hence we have not evaluated the benefit of these extensions quantitatively yet. Will do so as part of the future work.)

6.3.2 Task-based Implementation Strategy

We implement the extensions discussed in Section 6.3.1 in our OpenUH compiler. Our implementation is based on the hybrid model of pthreads + CUDA. CUDA 4.0 and later versions simplify multi-GPU programming by using only one thread to manipulate multiple GPUs. However, in our OpenACC multi-GPU extension implementation, we use multiple pthreads to operate multiple GPUs and each thread is associated with one GPU. This is because the memory allocation and free operations are blocking operations. If a programmer uses data `copy/copyin/copyout` inside a loop, the compiler will generate the corresponding data memory allocation

and transfer runtime APIs. Since the memory allocation is blocking operation, the execution in multiple GPUs cannot be parallel. CUDA code avoids this by allocating memory for all data first and then performs data transfer. In OpenACC, however, this is unavoidable because all runtime routines are generated by the compiler and the position of these routines cannot be randomly placed. Our solution is to create a set of threads and each thread manages the context of one GPU which is shown in Figure 6.10. This is a task-based implementation. Assume we have n GPUs attached to a CPU host, initially the host creates n threads and each thread is associated with one GPU. Each thread creates a empty First In First Out (FIFO) task queue which waits to be populated by the host main thread. Depending on the directive type and `deviceid` clause in the original OpenACC directive annotated program, the compiler generates the task enqueue request for the main thread. The task here means any command issued by the host and executed either on the host or on the device. For example, memory allocation, memory free, data transfer, kernel launch, and device to device communication, all of these are different task types.

The following are the definitions of the task structure and the thread controlling a GPU (refer it to GPU thread). The task is executed only by GPU thread. A task could be synchronous or asynchronous to the main thread. In the current implementation, most tasks are asynchronous except device memory allocation because the device address is passed from a temporary argument structure, so the GPU thread must wait for this to finish. Each GPU thread manages a GPU context, a task queue and some data structures in order to communicate with the master thread. Essentially this is still the master/worker model and the GPU threads are workers.

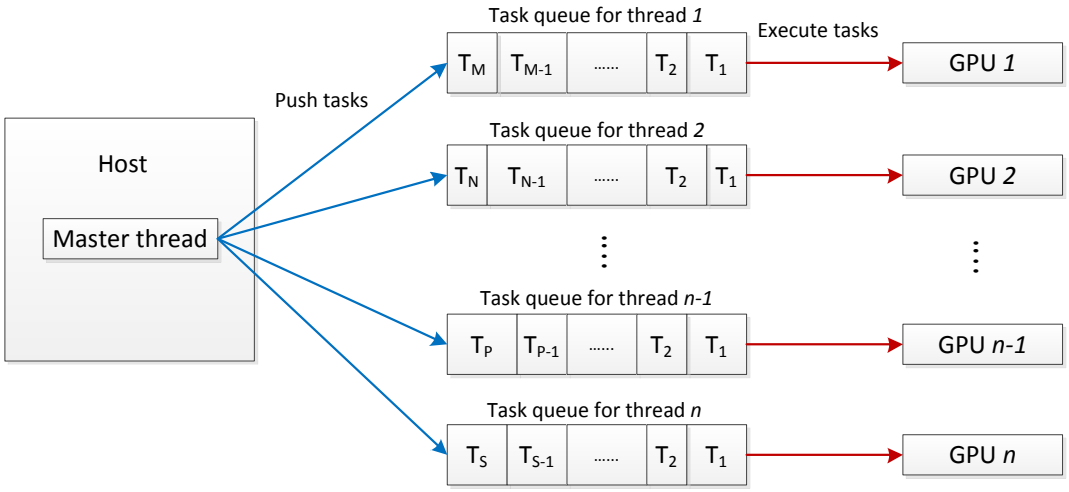


Figure 6.10: Task-based multi-GPU implementation in OpenACC. $T_i(i = 1, 2, ...)$ means a specific task

Algorithm 3 and algorithm 4 show a detailed implementation for the worker thread and master thread, respectively.

To enable device to device communication, we must enable such peer-to-peer access explicitly and this requires that all worker threads have created the GPU contexts. So each worker first creates the context for the associated GPU, then it waits until all workers have created the GPU contexts. The worker that is the last one to create the context will broadcast to all worker threads so that they can start to enable the peer-to-peer access. The worker then enters an infinite loop that waits for the incoming tasks. When the task is ready in the FIFO task queue, it will fetch the task from the queue head and execute that task. When there is no task available, the worker just goes to sleep to save CPU resource. Whenever a master pushes a task into the FIFO queue of a worker, it will signal that worker that the queue is

not empty and the task is ready. If the master notifies that the worker be destroyed, the worker will complete all pending tasks and then exit.

```
typedef struct _task_s
{
    int type; //task type (e.g. memory allocation and kernel launch, etc.)
    void* (*routine)(void*); // the task routine
    _work_args *args; // point to the task argument
    int work_done; // indicate whether the task is done
    int async; // whether the task is asynchronous
    struct _task_s *next; // next task in the task queue
} _task;

typedef struct
{
    int destroyed; // whether this thread is destroyed
    int queue_size; // the task queue size
    pthread_t thread; // the thread identity
    context_t *context; // the GPU context associated with this thread
    int context_id; // the GPU context id
    _task *queue_head; // head of the FIFO task queue
    _task *queue_tail; // tail of the FIFO task queue
    pthread_mutex_t queue_lock;
    pthread_cond_t queue_ready; // the task queue is not empty and ready
    pthread_cond_t work_done;
    pthread_cond_t queue_empty;
} _gpu_thread;
```

6.3.3 Evaluation with Benchmark Examples

In this section, we will discuss how to port some of the benchmarks discussed in Section 6.2 using the OpenACC extensions instead of using the hybrid model. The programs using the proposed directives were compiled by OpenUH compiler with "-fopenacc -nvcc,-arch=sm_35,-fmad=false" flag. We also compared the performance

Algorithm 3: The worker algorithm for multi-GPU programming in OpenACC

```
Function worker_routine
    Create the context for the associated GPU ;
    pthread_mutex_lock(...) ;
    context_created++;
    while context_created != num_devices do
        /* wait until all threads have created their contexts    */
        pthread_cond_wait(...);
    end
    pthread_mutex_unlock(...);
    if context_created == num_devices then
        pthread_cond_broadcast(...);
    end
    Enable peer access among all devices;
    while (1) do
        pthread_mutex_lock(cur_thread→queue_lock);
        while cur_thread → queue_size == 0 do
            pthread_cond_wait(&cur_thread→queue_ready, &cur_thread→
            queue_lock);
            if cur_thread → destroyed then
                pthread_mutex_unlock(&cur_thread → queue_lock);
                /* the context is blocked until the device has
                   completed all preceding requested tasks        */
                Synchronize the GPU context;
                pthread_exit(NULL);
            end
        end
        /* fetch the task from the queue head                     */
        cur_task = cur_thread→queue_head;
        cur_thread→queue_size−−;
        if cur_thread→queue_size == 0 then
            cur_thread→queue_head = NULL;
            cur_thread→queue_tail = NULL;
        else
            cur_thread→queue_head = cur_task→next;
        end
        pthread_mutex_unlock(&cur_thread→queue_lock);
        /* execute the task                                       */
        cur_task→routine((void*)cur_task→args);
    end
```

Algorithm 4: The master algorithm for multi-GPU programming in OpenACC

```
Function enqueue_task_xxxx
    Allocate memory and populate the task argument;
    Allocate memory and populate the task;
    pthread_mutex_lock(&cur_thread→queue_lock);
    /* push the task into the FIFO queue                              */
    if cur_thread→queue_size == 0 then
        cur_thread→queue_head = cur_task;
        cur_thread→queue_tail = cur_task;
        /* signal the worker that the queue is not empty and the
           task is ready                                              */
        pthread_cond_signal(&cur_thread→queue_ready);
    else
        cur_thread→queue_tail→next = cur_task;
        cur_thread→queue_tail = cur_task;
    end
    cur_thread→queue_size++;
    pthread_mutex_unlock(&cur_thread→queue_lock);
    /* if the task is synchronous                                     */
    if cur_task→async == 0 then
        pthread_mutex_lock(&cur_thread→queue_lock);
        /* wait until this task is done                               */
        while cur_task→work_done == 0 do
            pthread_cond_wait(&cur_thread→ work_done, &cur_thread→
            queue_lock);
        end
        pthread_mutex_unlock(&cur_thread→ queue_lock);
    end
```

with that of the CUDA version. All CUDA codes were compiled using "-O3 -arch=sm_35 -fmad=false" flag.

```
for(d=0; d<num_devices; d++)
{
    blocks = n/num_devices;
    start = id*blocks;
    end = (id+1)*blocks;

    #pragma acc data copyin(A[start*n:blocks*n])\
                     copyout(C[start*n:blocks*n])\
                     copyin(B[0:n*n]) deviceid(d)
    {
      #pragma acc parallel deviceid(d)\
            num_gangs(32) vector_length(32)
       {
       #pragma acc loop gang
       for(i=start; i<end; i++){
          #pragma acc loop vector
          for(j=0; j<n; j++){
            float c = 0.0f;
            for(k=0; k<n; k++)
                c += A[i*n+k] * B[k*n+j];
            C[i*n+j] = c;
          }
       }
    }
}

for(d=0; d<num_devices; d++){
    #pragma acc wait device(d)
}
```

Figure 6.11: A multi-GPU implementation of MM using OpenACC extension

Figure 6.11 shows a code snippet of multi-GPU implementation of matrix multiplication using the proposed OpenACC extensions. Using the proposed approach, the user still needs to partition the problem explicitly into different devices. If there is any dependence between devices, it is difficult for the compiler to do such analysis and manage the data communication. For the totally independent loop, we may further automate the problem partition in compiler as part of the future work. Figure 6.12 shows the performance comparison using different models. We can see that the performance of manual CUDA version and OpenACC extension version are

much better than that of the hybrid model. CAPS compiler seems did not performe well using the hybrid-model implementation. The performance of the proposed OpenACC extension version is the best and it is very close to the optimized CUDA code. There are several reasons for this. First, the loop translation mechanisms from OpenACC to CUDA in different compilers are different. Loop translation means the translation from OpenACC nested loop to CUDA parallel kernel. In the translation step, the OpenACC implementation in OpenUH compiler uses redundant execution model which has no synchronization between different OpenACC parallelism like gang and vector. However, PGI compiler uses another execution model which loads some scalar variables into shared memory in gang parallelism and then the vector threads fetched them from shared memory. The detailed comparison between these two loop translation mechanisms was explained in [76]. OpenUH compiler does not need to do those unnecessary shared-memory stores and loads operations and therefore reduces overhead. Second, we found that CAPS compiler used similar loop-translation mechanism as OpenUH. However, its performance is still worse than the OpenUH compiler. The possible reason is that it has non-efficient runtime-library implementation. Since CAPS itself does not provide OpenMP support, it needs complex runtime management to interact with the OpenMP runtime in other CPU compilers. This result demonstrates the effectiveness of our approach: not only simplifies the multi-GPU implementation but also maintains high performance.

We also port the 2D-Heat Equation to the GPUs using the proposed OpenACC directive extension. In the code level, the device-to-device communications are not required to go through the host, instead the `update peer` directive can be used to

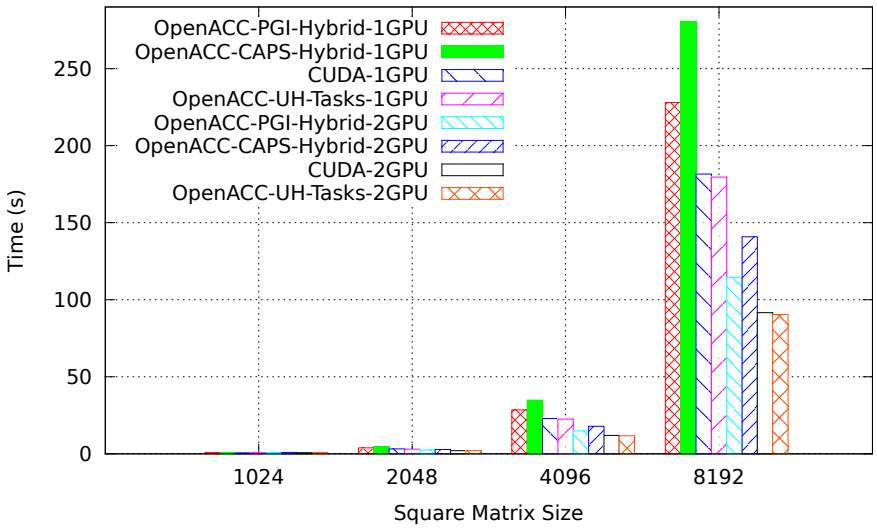


Figure 6.12: Performance comparison for MM using multiple models. PGI and CAPS compilers compile the hybrid model implementation

reduce the code complexity and therefore improve the implementation productivity. Figure 6.13 shows the detailed multi-GPU implementation code using the OpenACC extension and Figure 6.14 explains this implementation graphically. Compared to the Figure 6.9 that uses the hybrid model, it is obvious to see that the data transfer between devices is simpler. Figure 6.15 show the performance comparison using different models. The performance of the hybrid model using CAPS compiler is not shown because it is around 5x slower than PGI's performance. When the grid size is 4096*4096, the execution time of OpenACC version is around 60 seconds faster than the hybrid model and it is close to that of the optimized CUDA code. We notice that there is almost no performance loss with our proposed directive extension.

```
for(d=0; d<num_devices; d++){
    #pragma acc enter data copyin(temp1_h[d*rows*LDA:(rows+2)*LDA]) device(d)
    #pragma acc enter data copyin(temp2_h[d*rows*LDA:(rows+2)*LDA]) device(d)
}

for(istep=0; istep<nstep; istep++){
    for(d=0; d<num_devices; d++)
        step_kernel_(ni+2, rows+2, tfac, temp1_h+d*rows*LDA, temp2_h+d*rows*LDA)
    }

    /* wait to finish the kernel computation */
    for(d=0; d<num_devices; d++){
        #pragma acc wait device(d)
    }
    /* exchange halo data */
    for(d=0; d<num_devices; d++){
        if(d > 0){
          #pragma acc update peer to(temp2_h[d*rows*LDA:LDA]) to_device(d)
                              from(temp2_h[d*rows*LDA:LDA]) from_device(d-1)
        }

        if(d < num_devices - 1){
       #pragma acc update peer to(temp2_h[(d+1)*rows*LDA+LDA:LDA]) to_device(d)
                           from(temp2_h[(d+1)*rows*LDA+LDA:LDA]) from_device(d+1)
        }
    }

    /* swap pointer of in and out data */
    temp_tmp = temp1_h;
    temp1_h = temp2_h;
    temp2_h = temp_tmp;
}

for(d=0; d<num_devices; d++){
    #pragma acc exit data copyout(temp1_h[(d*rows+1)*LDA:rows*LDA]) deviceid(d)
}

for(d=0; d<num_devices; d++){
    #pragma acc wait device(d)
}
```

Figure 6.13: Multi-GPU implementation with OpenACC extension - 2D-Heat Equation

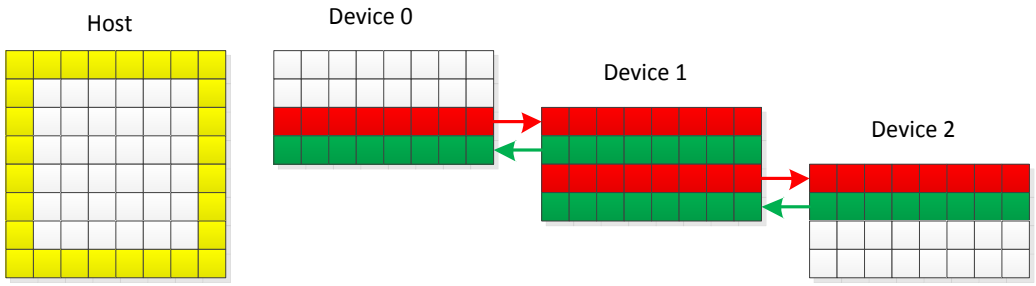


Figure 6.14: Multi-GPU implementation of 2D-Heat Equation using OpenACC extension

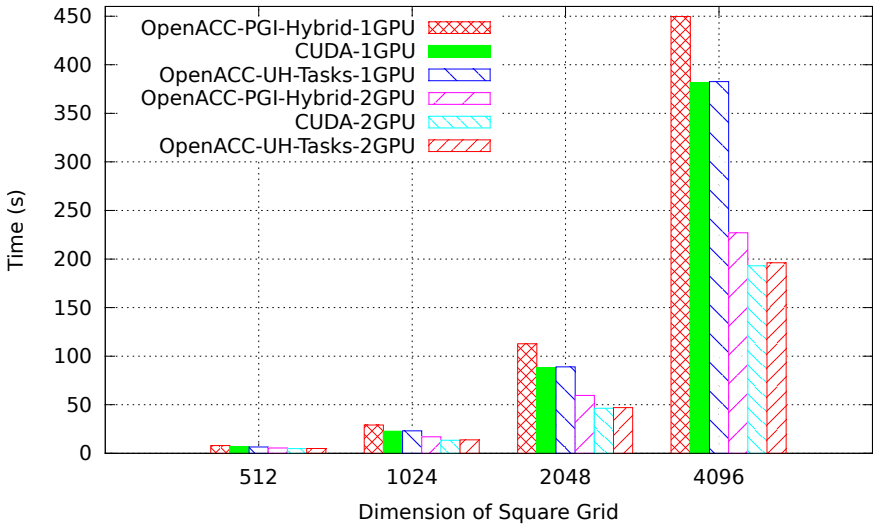


Figure 6.15: Performance comparison for 2D-Heat Equation using multiple models. PGI compiler compiles the hybrid model implementation

6.4 Summary

This chapter explored the programming strategies of multi-GPU within a single node using the hybrid model, OpenMP & OpenACC. We demonstrated the effectiveness of our approach by exploring three applications of different characteristics. In the first application where there were different kernels, each kernel was dispatched to one GPU. The second application had a large workload that was decomposed into multiple small sub-workloads, after which each sub-workload was scheduled on one GPU. Unlike the previous two applications that consist of totally independent workloads on different GPUs, the third application required some communication between different GPUs. We evaluated the hybrid model with these three applications on multi-GPU and noticed reasonable performance improvement. Based on the experience gathered in this process, we have proposed some extensions to OpenACC in order to support multi-GPU and implemented it using task-based strategy. By using the proposed directive extension, we can simplify the multi-GPU programming while obtaining better performance compared to the hybrid model. Most importantly, the performance was close to that of the optimized manual CUDA code.

Chapter 7

Locality-Aware Auto-tuning for Loop Scheduling

In the porting of NPB benchmark suite to GPUs using OpenACC model, one optimization, we manually applied was loop-scheduling tuning. We manually tried different loop schedules and chose the one that delivered the best performance. This was tedious and very time-consuming because we had to run the application after we tried each loop schedule and recorded its performance. In large applications that have hundreds of kernels, it was impossible to do this. A better way to solve this issue is to apply an auto-tuning technique to let the compiler automatically choose an optimal loop schedule. Our goal is to achieve a reasonable performance improvement when using the loop schedule chosen by our framework [75] and compare that to the default loop schedule chosen by the compiler.

7.1 Related Work

We used auto-tuning technique to find the optimal loop schedule. Several approaches can be used for auto-tuning. Here, we grouped them into three categories: exhaustive search, heuristics, and machine learning.

Exhaustive Search This approach searches all combinations of parameters to find the best solution. At present there is no related work that solely uses exhaustive search to find the best loop schedule, but there are related works that use this approach. Cui et al. [28] used auto-tuning technique to test the SGEMM and DGEMM with different parameters to find the best performance on Fermi GPU architecture. Their auto-tuner had two components, a code generator and an execution engine. The code generator generated different matrix multiply kernels based on different combinations of parameters. The execution engine then executed them and identified the best one. Montgomery et al. [54] used efficient search approach such as direct search to identify the optimal loop schedule. Their approach executed the kernels with different loop schedules. Grauer-Gray et al. [33] improved the performance of applications using high-level language auto-tuning. The optimizations that they applied included loop permutation, loop unrolling, loop tiling, and specifying which loops to parallelize. They generated different transformed codes using a python script which contained different combinations of unroll, tiling, permutation, and parallelization transformations for a particular kernel.

Heuristics This approach uses an analytical model or algorithm to restrict the search space. Lee et al. [47] presented a framework to automatically and efficiently

map a nested loop to GPU. They targeted a high-level language such as Domain Specific Language (DSL). The parameters that formed the search space included the dimension of the nested loop, the block size, and the degree of parallelism which was essentially the grid size. They applied hard constraints and soft constraints to restrict the search space. In the soft constraints, they selected a set of commonly used GPU optimizations such as memory coalescing and reduced thread branching and gave a weight to each of them. A loop schedule was assigned a score based on the weight of each soft constraint. For all thread schedules that satisfy the hard constraints, they did an exhaustive search for all soft-constraint combinations, and found the schedule with the best score. However, they did not explain what soft constraints were considered and what score was assigned to each constraint. Another key factor that was not considered by their model is the data locality.

Machine Learning There are two approaches that used machine learning. The first approach constructed a knowledge base with optimized parameters for training benchmarks. This was used to predict the parameter of the new benchmark using similar training benchmarks. For instance, Siddiqui et al. [61] presented how to choose the optimal loop schedule and the gang-vector number within each loop schedule. Their approach included two steps: firstly identify the loop schedule that delivers the best performance with the compiler default settings; secondly tune the gang-vector number using a brute force search. The learner recorded the best parameter for different problem sizes. When a new problem size was determined, the Euclidian distance between the new problem size and the problem sizes in the training set was calculated. The problem size with the least distance was chosen. The optimal

and near-optimal parameter settings for the new problem size. This approach was time consuming and not accurate. First, it was only used for different problem sizes of the same application. Secondly, the loop schedule that gave the best performance with the compiler default gang-vector number was not guaranteed to have better performance than the loop schedule with other gang-vector number. Thirdly, brute force search was used for the gang-vector number which is very time-consuming. The second approach in machine learning is to build a machine learning model (e.g. tree) based on selected features for all training benchmarks. This model is used to predict the new benchmark. However, there is no related work on loop-scheduling research.

7.2 The Motivating Example

In Chapter 4, we saw how loop-scheduling tuning was applied manually. Manual optimization is tedious and error-prone. The goal of this chapter is to apply the loop-scheduling optimization automatically in the compiler. So far commercial compilers have optimized this automatically, but their approaches are not effective. We tried PGI 15.1 compiler for OpenACC programs with the default loop schedules chosen by the compiler and the loop schedules chosen by manual optimization. The result is seen in Figure 7.1. The time of the manually optimized schedules were normalized to 1. It was observed that for Vecadd, Matvec, and Jacobi, the compiler's default schedule was around 10% slower than the optimized schedule. For Laplacian and Wave13pt, the compiler's default schedule was 20% - 30% slower. For Matmul, the default schedule using `parallel` directive was greater than 200% slower than

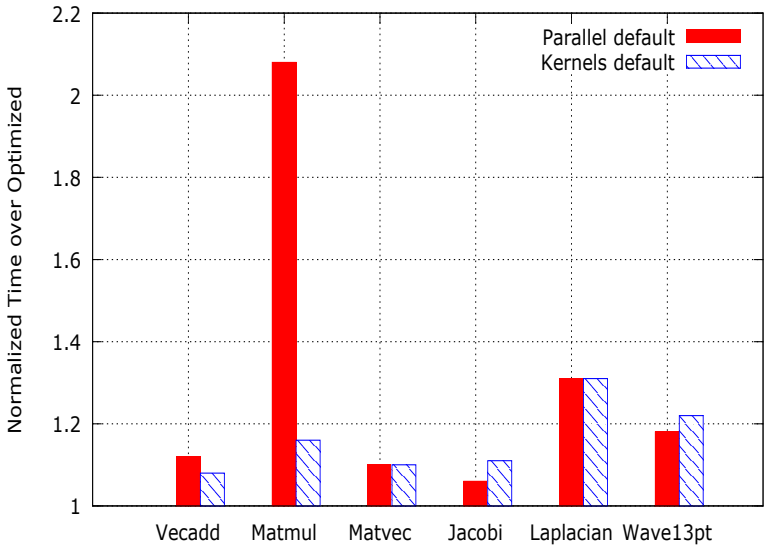


Figure 7.1: Loop scheduling comparison

Table 7.1: Loop scheduling in different cases

Code	Parallel default	Kernels default	Manual Optimized
Vecadd	gang/seq/vector	seq/gang/gang vector	gang/gang/vector
Matmul	gang/vector/seq	gang/gang vector/seq	gang vector/gang vector/seq
Matvec	gang vector	gang vector	gang vector
Jacobi	gang/vector	gang/gang vector	gang/vector
Laplace	gang/seq/vector	seq/gang/gang vector	gang/worker/gang vector
Wave	gang/seq/vector	seq/gang/gang vector	gang/worker/vector

the optimized schedule. Table 7.1 shows the comparison between different loop-schedules applied by the compiler and manually. Our goal was to choose an optimal loop-schedule by the compiler with an analytical model, so that there is no need for the user to try different loop schedule combinations.

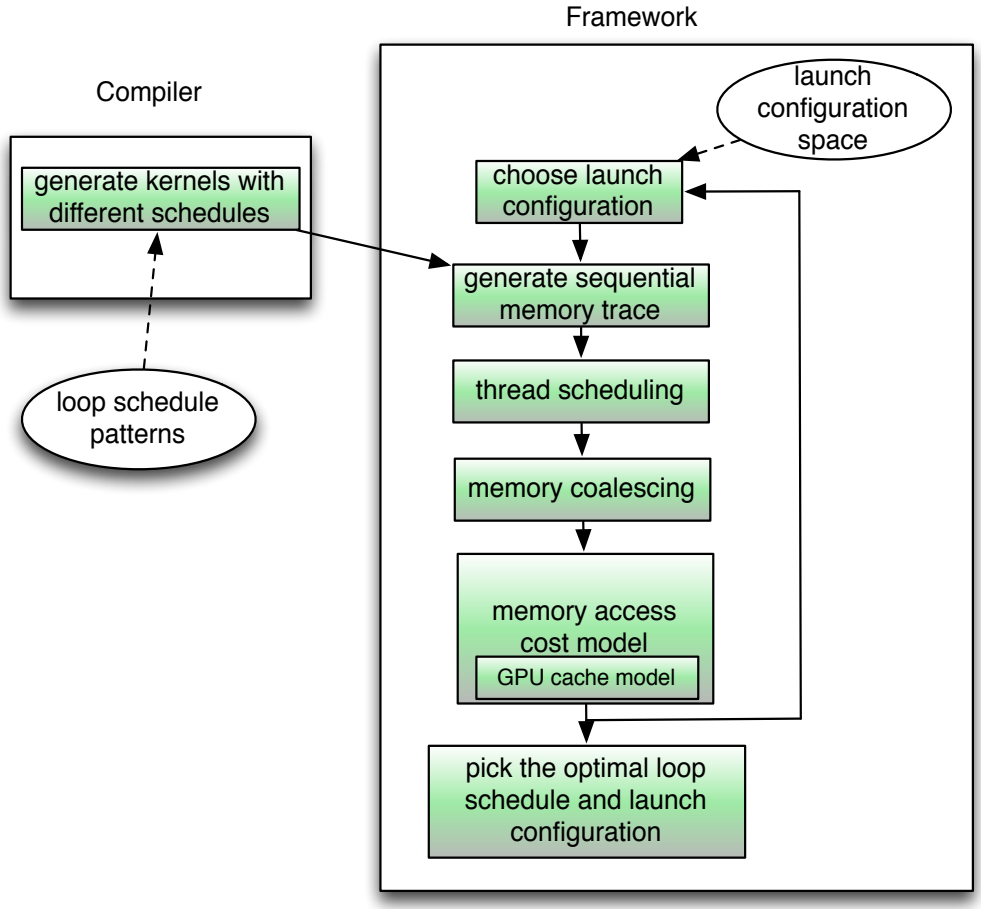


Figure 7.2: The framework of auto-tuning for loop scheduling

7.3 Loop Scheduling Auto-tuning

7.3.1 The Auto-tuning Framework

In this section, we describe our auto-tuning framework, the proposed analytical model that enabled the identification of the appropriate loop schedule, and the launch configuration used for each of the loop schedules. Figure 7.2 gives an overview of the auto-tuning framework. The compiler generated multiple kernel files with different

loop schedules. The loop schedule was chosen from a set of loop schedule patterns which covers both double- and triple-nested loops. The framework chose a launch configuration from the launch configuration search space which depended on the iteration space of each loop. Then the framework generated a memory trace based on the loop schedule and the launch configuration. This memory trace was used in the GPU cache model. Until this step, the extracted memory trace defined the sequential behavior of the program, since the loop iterations to GPU threads were sequentially assigned. To simulate the GPU's parallel execution model, the memory references in the trace was reordered to reflect the GPU parallel execution. The order of threads influences thread scheduling. In the GPU, a warp (the smallest execution unit) defines a set of consecutive threads. If consecutive threads access consecutive memory addresses, the memory accesses are coalesced. They are merged into fewer memory transactions. Our model simulates the memory coalescing behavior of GPU architecture. For instance, if the memory addresses, referenced by all threads in a warp, are in one cache line, then memory access requests will be merged into one memory request.

After memory requests were coalesced, the memory trace was fed into the memory access cost model where a memory access cost was computed, with the cache model. This process was repeated until the framework iterates over all loop schedules and launch configuration space. Finally, the framework picked the optimal-loop schedule and the corresponding launch configuration that had minimal memory access. The compiler then recompiled the same program using the selected loop schedule. The major components in this framework will be discussed in the following sub-sections:

7.3.2 Loop Schedule Patterns

We only consider the double- and triple-nested loops. Note that the loop nest level means the parallelizable loop nest. For instance, in the body of the parallelizable loop nest, there could be another loop nest that is sequentially executed. In the current GPU programming models such as OpenACC, the maximum level of the parallelizable loop nest is three. If a nested loop has more levels to parallelize, it can be collapsed into two or three loops. In the current implementation, the double- and triple-nested loops were tested. We considered three types of loop schedules for both types of loop nests. The notations used in the loop schedules are as follows:

- bx, by, and bz: denote X, Y, and Z dimension of the grid, respectively
- tx, ty, and tz: denote X, Y, and Z dimension of the thread block, respectively
- num_bx, num_by, and num_bz: denote the size of X, Y, and Z dimension of the grid, respectively
- num_tx, num_ty, and num_tz: denote the size of X, Y, and Z dimension of the thread block, respectively

The three loop schedules for the double-nested loop (x-loop for the inner loop and y-loop for the outer loop) are:

- schedule 2_1: x-loop is mapped to the X-dimension of a thread block, and y-loop is mapped to the X-dimension of the grid.
- schedule 2_2: x-loop is mapped to the X-dimension of both the thread block and the grid, and y-loop is mapped to the Y-dimension of the grid.

- schedule 2_3: x-loop is mapped to X-dimension of both the thread block and the grid, and y-loop is mapped to the Y-dimension of both the thread block and the grid.

These loop schedule directives are implicitly added by the compiler. The graphical explanations for these loop schedules are seen in Figure 7.3, 7.4, and 7.5. The detailed mapping functions from the loop iterations to GPU threads are seen in Listing 7.3, 7.4, and 7.5. The purpose of schedule 2_2 is to overcome the limit of GPU threads within one block. In both schedule 2_1 and 2_2, the threads computing the outer loop are in different thread blocks, likely to be scheduled to different GPU SMs (Streaming Multiprocessors). This may not exploit the data locality efficiently. How can the data locality be improved? We consider the loop schedule 2_3 which allows threads computing the outer loop iterations to remain in the same block improving the data locality. For a triple-nested loop, a code example is seen in Listing 7.2 and other similar loop schedules are designed. Because of the space limit, we only illustrate the graphical representation for one loop schedule in Figure 7.6, in which x-loop, y-loop, and z-loop refer to the inner-most loop, the middle loop, and the outer-most loop, respectively. The loop schedule in Figure 7.6 means that the x-loop is mapped to X-dimension of a thread block, y-loop is mapped to Y-dimension of the same thread block, and z-loop is mapped to X-dimension of the grid.

```
#pragma acc loop
for(j = j_start; j < j_end; j++){
    #pragma acc loop
    for(i = i_start; i < i_end; i++){
        ......
    }
}
```

Listing 7.1: Double nested loop example

```
#pragma acc loop
for(k = k_start; k < k_end; k++){
  #pragma acc loop
  for(j = j_start; j < j_end; j++){
    #pragma acc loop
    for(i = i_start; i < i_end; i++){
        ......
    }
  }
}
```

Listing 7.2: Triple nested loop example

```
#pragma acc loop bx(num_bx)
for(j = j_start; j < j_end; j++){
    #pragma acc loop tx(num_tx)
    for(i = i_start; i < i_end; i++){
        ......
    }
}
mapping function to CUDA:
j = j_start + blockIdx.x + t * gridDim.x , (t = 0, 1, ..., (j_end - j_start)/gridDim.x - 1)
i = i_start + threadIdx.x + t * blockDim.x , (t = 0, 1, ..., (i_end - i_start)/blockDim.x - 1)
```

Listing 7.3: Loop schedule 2_1

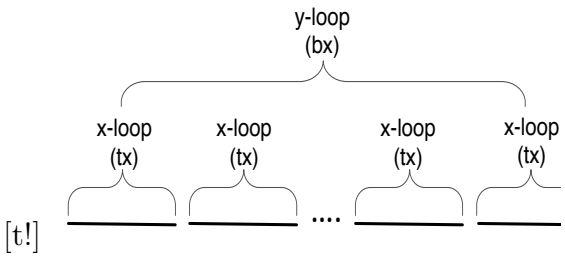
```
#pragma acc loop by(num_by)
for(j = j_start; j < j_end; j++){
    #pragma acc loop bx(num_bx) tx(num_tx)
    for(i = i_start; i < i_end; i++){
        ......
    }
}
mapping function to CUDA:
j = j_start + blockIdx.y + t * gridDim.y
```

$(t = 0, 1, ..., \frac{j_{end} - j_{start}}{gridDim.y} - 1)$

$i = i_{start} + threadIdx.x + blockIdx.x * blockDim.x + t * blockDim.x * gridDim.x$

$(t = 0, 1, ..., \frac{i_{end} - i_{start}}{blockDim.x * gridDim.x} - 1)$

Listing 7.4: Loop schedule 2_2



[t!]

Figure 7.3: Loop schedule 2_1

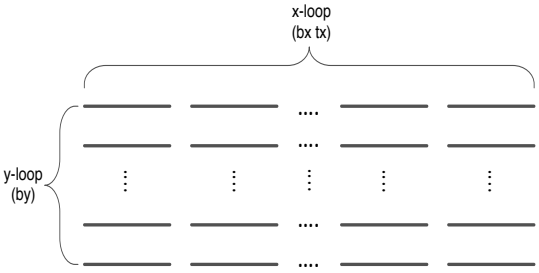
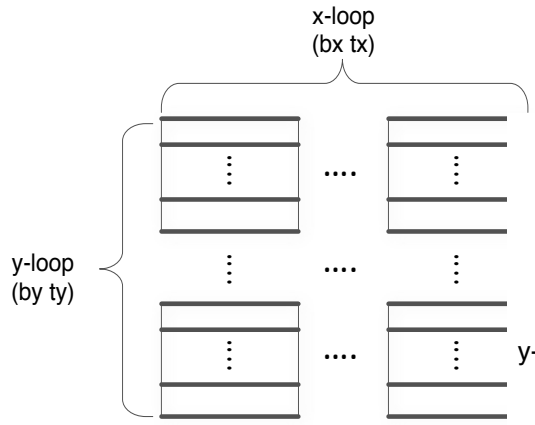


Figure 7.4: Loop schedule 2_2



[t!]

Figure 7.5: Loop schedule 2_3

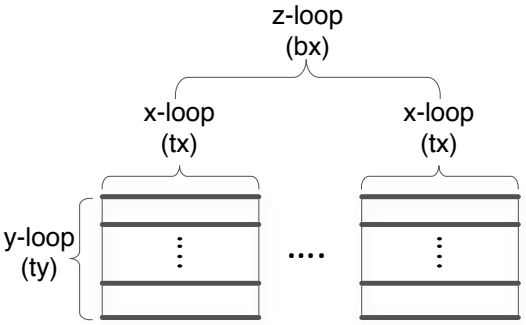


Figure 7.6: Loop schedule 3_1

```
#pragma acc loop by(num_by) ty(num_ty)
for(j = j_start; j < j_end; j++){
    #pragma acc loop bx(num_bx) tx(num_tx)
    for(i = i_start; i < i_end; i++){
        ......
    }
}
```

```
mapping function to CUDA:
```

$j = j_{start} + threadIdx.y + blockIdx.y * blockDim.y + t * blockDim.y * gridDim.y$

$(t = 0, 1, ..., \frac{j_{end} - j_{start}}{blockDim.y * gridDim.y} - 1)$

$i = i_{start} + threadIdx.x + blockIdx.x * blockDim.x + t * blockDim.x * gridDim.x$

$(t = 0, 1, ..., \frac{i_{end} - i_{start}}{blockDim.x * gridDim.x} - 1)$

Listing 7.5: Loop schedule 2_3

7.3.3 Thread Scheduling

The memory trace is defined for how the memory is accessed, which in turn is defined for how the thread blocks are scheduled into different Streaming Multiprocessors (SMs) and how the threads are scheduled within each SM. When the GPU launches a grid of threads for a kernel, the grid is divided into 'waves' of thread blocks. For example, let us assume there are 15 SMs. Each SM has two thread blocks hence 30 thread blocks in total. Thread block 0 and thread block 15 are assigned to SM 0. Thread block 1 and thread block 16 are assigned to SM 1. If there was a scenario with 60 thread blocks and each SM allows at most 2 blocks (30 blocks for 15 SMs), we will need to assign these blocks into two waves; 30 thread blocks to the first wave and the other 30 thread blocks to the second wave. We use a round-robin scheduling mechanism to schedule the thread blocks to all SMs in all waves.

The equation to calculate the number of waves is given in Equation 7.1. The number of waves is obtained by dividing the total number of thread blocks by the active thread blocks per SM times the number of SMs. The active blocks per SM is given in Equation 7.2. For instance, in Kepler GPU, the $max\_threads\_per\_SM$ is

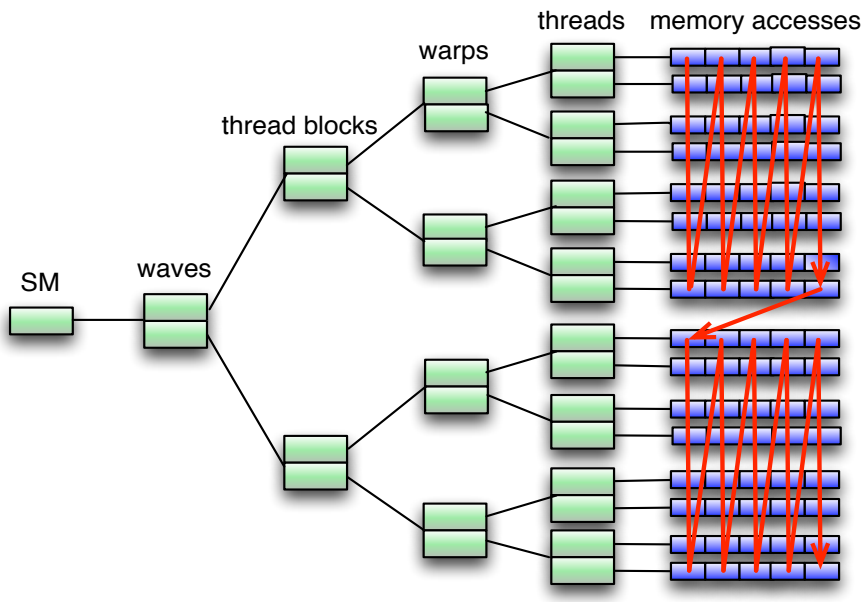


Figure 7.7: Thread scheduling used in the auto-tuning framework

2048 and $max\_thread\_blocks\_per\_SM$ is 16 and upon knowing the number of thread blocks in the kernel, which is specified by the launch configuration, we can determine the number of waves.

$$waves = \frac{thread\_blocks}{active\_blocks\_per\_SM \times \#SMs} \tag{7.1}$$

$$active\_blocks\_per\_SM = \\ min(max\_threads\_per\_SM/block\_size, max\_thread\_blocks\_per\_SM) \tag{7.2}$$

Figure 7.7 shows the thread scheduling mechanism. It highlights two waves that are scheduled within one SM. Each wave has two thread blocks; each thread block has two warps; each warp has two threads; each thread has five memory accesses. We

access the memory references in a round-robin manner. This memory access pattern produces a memory trace.

7.3.4 Memory Access Cost Model

After memory coalescing, the memory trace is fed into the memory access cost model which computes the memory access cost for a specific loop schedule and launch configuration. The metric used in this model is presented as

$$Cost_{mem} = \sum_{i}^{\#levels} (N_i \times L_i) \tag{7.3}$$

where N_i means the number of transactions happened in level i of the memory hierarchy, and L_i means the latency of memory level i.

The rationale behind this metric is the memory hierarchy in GPU architecture which is seen in Figure 7.8. When the kernel accesses a global memory address, it loads that address from L1 cache. If the data is already in L1 cache, then the access is a hit. If the data is not in L1 cache, then the access is a miss and it needs to load the data from L2 cache. If the data is not in L2 cache, then it needs to further load the data from DRAM. So the formula after expanding the Equation 7.3 is seen in Equation 7.4 which is the sum of the memory access cost from L1, L2, and DRAM. The formula for calculating each individual cost is given in Equation 7.5, Equation 7.6, and Equation 7.7. The '4' in Equation 7.6 and Equation 7.7 explains the number of global memory load transaction that is increased by 1 every 128 bytes in L1 cache, but 4 every 32 bytes in L2 cache and DRAM. Since the memory access latency orders from high to low - DRAM, L2 cache, and L1 cache, the goal is to

access high-order memories as less as possible. In other words, we would like to have few global loads and low L1 and L2 cache miss rates as possible. When there is intra-thread data reuse or inter-thread data reuse, different loop schedules have different cache miss rates, and finally the performance of the kernels using those loop schedules would be different.

$$Cost_{mem} = Mem_{L1} + Mem_{L2} + Mem_{DRAM} \tag{7.4}$$

$$Mem_{L1} = global\_loads * (1 - L1\_miss\_rate) * L1\_latency \tag{7.5}$$

$$Mem_{L2} = global\_loads * L1\_miss\_rate * 4 * L2\_latency \tag{7.6}$$

$$Mem_{DRAM} = global\_loads * L1\_miss\_rate * L2\_miss\_rate * 4 * DRAM\_latency \tag{7.7}$$

The key factors of the model are to estimate the global memory loads, L1, and L2 cache miss rates. The reuse distance model [13] is used to estimate L1 and L2 cache miss rates. It is a classic model to predict the cache misses in CPU applications. The primary reasons for cache misses are cold/compulsory, conflict, and capacity misses, famously termed as the 3C model. The **cold miss** occurs when there is no data in the cache, no matter how big the cache is. The **conflict miss** usually occurs in direct-mapped caches and set-associative caches. Two cache lines may map to the same cache slot even though there may be empty slots. The **capacity miss** happens when there are no more available slots in the cache. The reuse distance

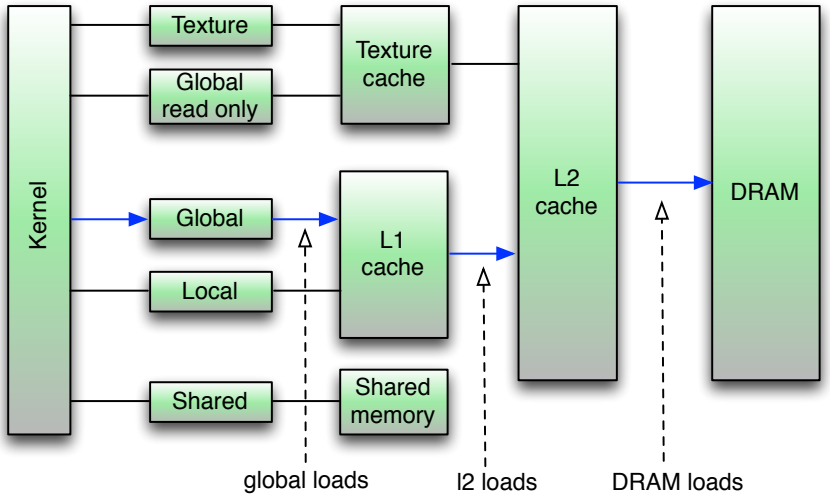


Figure 7.8: GPU memory hierarchy

model assumes that a LRU replacement fully-associative cache is used. So it can only predict cold miss and capacity miss.

To the best of our knowledge, there is no existing work that discusses L2 cache modeling in GPU. We found a couple of other related work discussing GPU L1 cache modeling. Tang et al. [63] applied the reuse distance theory to model the GPU L1 cache. However, there were a few weaknesses and limitations in their approach: (1) they assumed only one thread block is active in one SM which is not true in the real hardware; (2) they modeled the cold miss and conflict miss but did not model the capacity miss, however, some research have shown that only a minority of the misses are conflict misses in both CPU [20] and GPU [56]; (3) they validated their model against a GPU simulator which is not a real hardware per se. Nugteren et al. [56] also used the reuse distance to model GPU L1 cache. However, in their implementation, all thread blocks were scheduled into only one SM which is not the case in a real

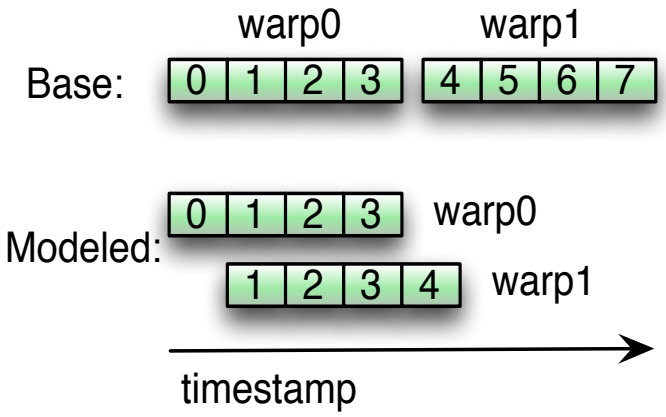


Figure 7.9: L1 cache modeling

hardware. Our thread scheduling mechanism overcomes the drawbacks of the above two papers discussed.

The reuse distance theory can measure both spatial locality and temporal locality if the distance is measured with cache line granularity. The spatial locality defines that the nearby memory addresses are likely to be referenced again in the near future. The temporal locality defines that the same data is likely to be referenced again in the near future.

The spatial locality is reflected by the memory coalescing level in the GPU kernel. If a GPU kernel has coalesced memory accesses, then it has better spatial locality than the kernel that has uncoalesced memory accesses. This is because the coalesced memory accesses allow the nearby data elements to be accessed at the same time the current data is accessed.

The temporal locality is reflected by the loop schedule. Different loop schedules pose different temporal locality since the execution order of the threads are different.

Table 7.2: Reuse distance example. Assume cache line has 16 bytes and the cache size is 32 bytes. The reuse distance is based on cache line granularity

address	0	8	16	96	8	16	17	104
cache line	0	0	1	6	0	1	1	6
reuse distance	∞	0	∞	∞	2	2	0	2
cache hit/miss	miss	hit	miss	miss	miss	miss	hit	miss

The reuse distance theory can effectively capture both the spatial locality and temporal locality. Table 7.2 shows a reuse distance example. In this example, assume the cache line has 16 bytes. If the data is firstly accessed or when a cold miss happens, the reuse distance is recorded as ∞. The reuse distance is a metric that defines the *distinct* memory accesses between the current memory access and the last access. If the reuse distance is larger or equal to the total number of cache lines, then a data reference is missed in the cache. The cache hit rate can be obtained by diving the hits by the total number of hits and misses.

Although the classic reuse distance model can predict the cache miss rate in the CPU, it cannot be simply applied as-is on the GPU since the architectures are significantly different. The most important difference is that in GPU, the threads in a warp execute in lock-step manner and therefore memory coalescing is important in the memory accesses of a warp. If the memory addresses referenced by all the threads in a warp are in a cache line, then the memory accesses are merged into one memory access. Another difference is the parallel memory processing in GPU. Therefore in our implementation, the L1 cache modeling includes the parallel memory processing. We also compare it with the base implementation. The difference of the "Base" and "Modeled" is seen in Figure 7.9. In the "Base" version, the memory coalescing is

applied to the memory trace. Then the memory requests from different warps are processed in order. If the memory requests in a warp are not coalesced, then they are also processed in order within a warp. In the "Modeled" version, we also apply memory coalescing, but we further add a timestamp. The timestamp is added to the following warps and also added to the threads in the same warp if their memory requests are not coalesced.

In the implementation of the reuse distance model, a key factor is the input which is a memory trace. In our analytical model, the memory traces are different for different loop schedules. This is because different loop schedules assign the loop iterations into GPU threads differently, thus the memory traces are different, and eventually the cache misses are different.

For the L2 cache modeling, we must first apply L1 cache modeling for all SMs and record the cache misses in their individual list. Then the memory trace is processed in round-robin manner which is similar to the description in Figure 7.7.

7.4 Performance Evaluation

The experimental platform is Intel Xeon processor E5520 with frequency 2.27 GHz and 32 GB main memory and an Nvidia Quadro K6000 GPU card which uses K40 architecture. L1 and L2 cache sizes are 16 KB and 1.5 MB, respectively. The cache line size for both L1 and L2 is 128 bytes. The proposed framework was implemented within the OpenUH compiler. The actual L1 and L2 cache hit rates were obtained from l1_cache_global_hit_rate and l2_l1_read_hit_rate metrics in CUDA profiler called

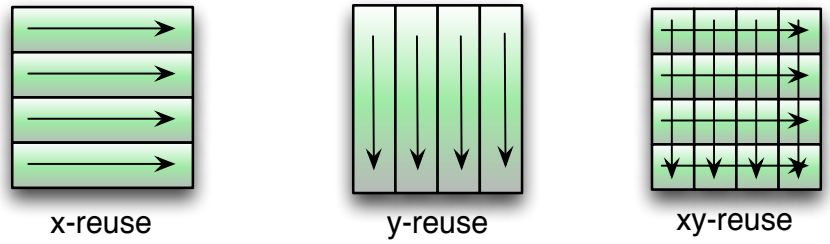


Figure 7.10: Data reuse patterns

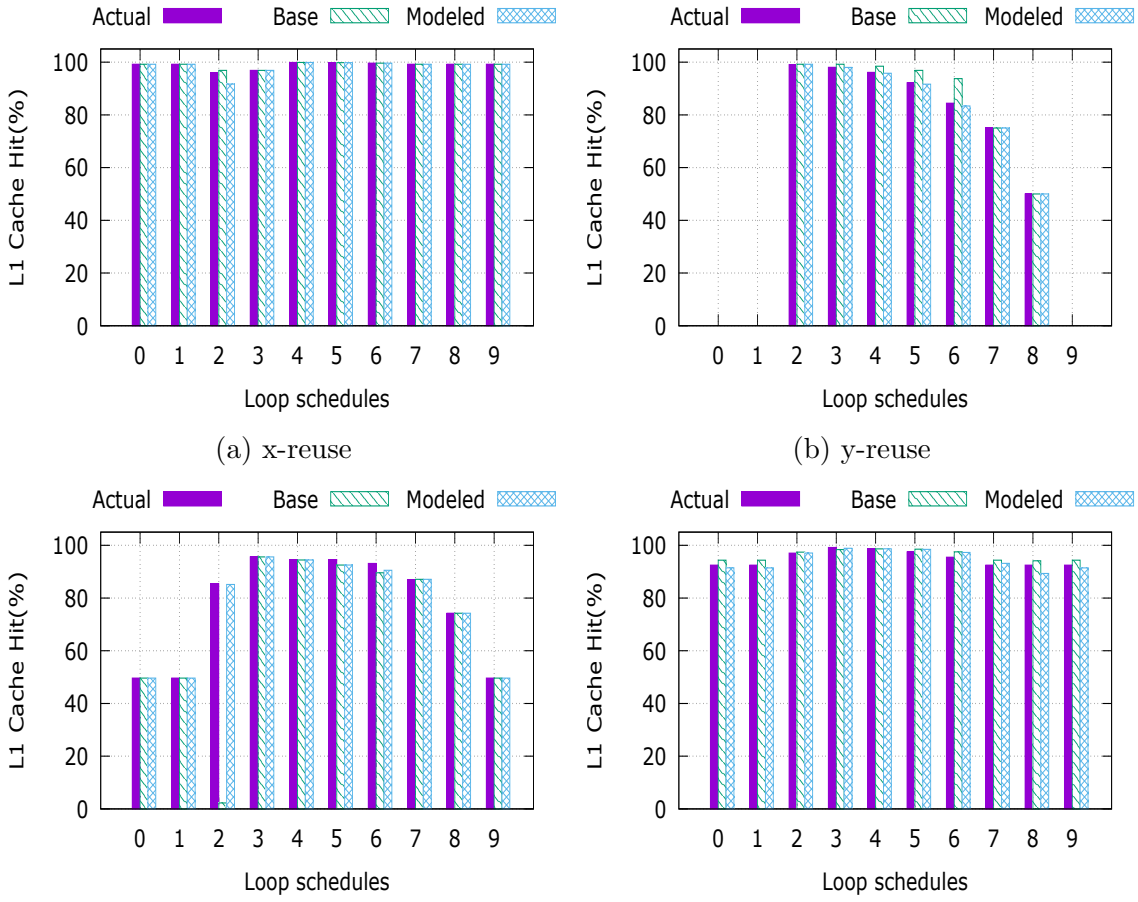
nvprof and the actual global memory loads were obtained from gld_transactions metric. Our auto-tuning framework was evaluated with diverse benchmarks and a machine learning algorithm called Support Vector Machine (SVM) [68].

7.4.1 Benchmarks

To evaluate our auto-tuning framework, we considered several benchmarks: two synthetic benchmarks (x-reuse and y-reuse), four from kernelGen OpenACC Performance Test Suite [12] (Matrix Multiplication, Jacobi, Laplacian and Divergence), one from CUDA SDK (Matrix Transpose), and one from EPCC OpenACC benchmarks [11] (Himeno). We tested different data reuse patterns using the two synthetic benchmarks. Figure 7.10 shows these two benchmarks along with another pattern i.e. xy-reuse, a classic Matrix Multiplication case. The "x" here refers to the inner loop and "y" refers to the outer loop in a double-nested loop. In the x-reuse benchmark, the inner loop reuses the common data; while in the y-reuse benchmark, the outer loop reuses the common data. The third case is the xy-reuse where both the inner and the outer loop reuse some common data.

Figure 7.11 shows the results for L1 cache hit modeling for some of the benchmarks discussed above. Figure (a) and (b) are results for the two synthetic benchmarks and Figure (c) and (d) are results for a couple of benchmarks from kernelGen suite. Results for other benchmarks were quite similar, so we have not included them in the dissertation. The results indicate that modeled result is more accurate than the "Base" version since it takes into account the parallel memory processing. Figure 7.11 (a) shows that the cache hit rates are high for all loop schedules. This is because for all iterations in x loop, the data they share are in one row and in the same contiguous memory section. Figure 7.11 (b) shows that the shared data are in the same column and therefore they are not contiguous in memory. This leads to relatively lower cache hit. Figure 7.11 (c), result of Matrix Multiplication, shows that there is data reuse in both x and y loops and therefore the shape of cache hit results seems like a combination of x-reuse and y-reuse. Figure 7.11 (d), result of Jacobi shows that, the overall hit rate is slightly lesser than x-reuse. This is because the data that the threads share are stencil-like. For instance, considering a 4-point stencil, for different points, the data that the threads access are not in contiguous memory locations. However, for a specific point, the data that the threads share are still in contiguous memory location. As a result, the cache hit rates are still relatively high. If the cache hit is high, the indication is that the threads take lesser time to fetch the data from high-latency memory.

The GPU L2 cache modeling result is seen in Figure 7.12. We show the results for partial benchmarks including Laplacian, Divergence, and Himeno. The results indicate that some loop schedules have low L2 cache hit while others have high



(a) x-reuse

(b) y-reuse

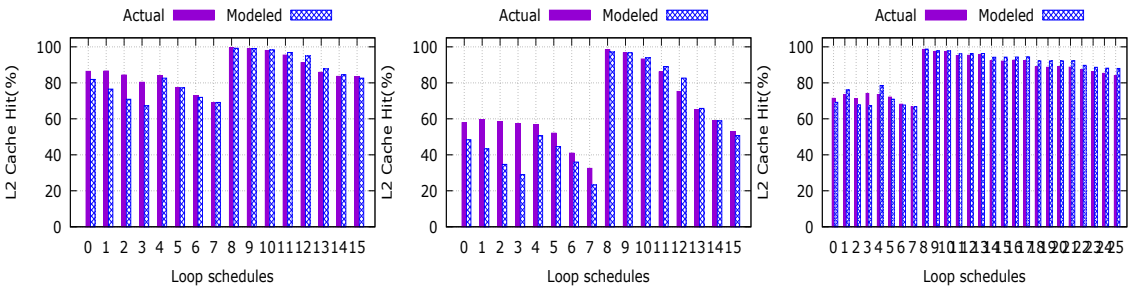
(c) Matrix Multiplication (xy-reuse)

(d) Jacobi

Figure 7.11: GPU L1 cache modeling

L2 cache hit. This illustrates the importance of choosing the right loop schedules. The error percentage of the modeled L2 hit against the actual hit is only 4.37%, 13.72%, and 2.76% for Laplacian, Divergence , and Himeno, respectively. The low error percentages indicate that our model can accurately capture the L2 locality for different loop schedules.

Figure 7.13 shows the global memory loads of kernels in the four benchmarks discussed in Figure 7.11. The plots show that the modeled loads (before kernel

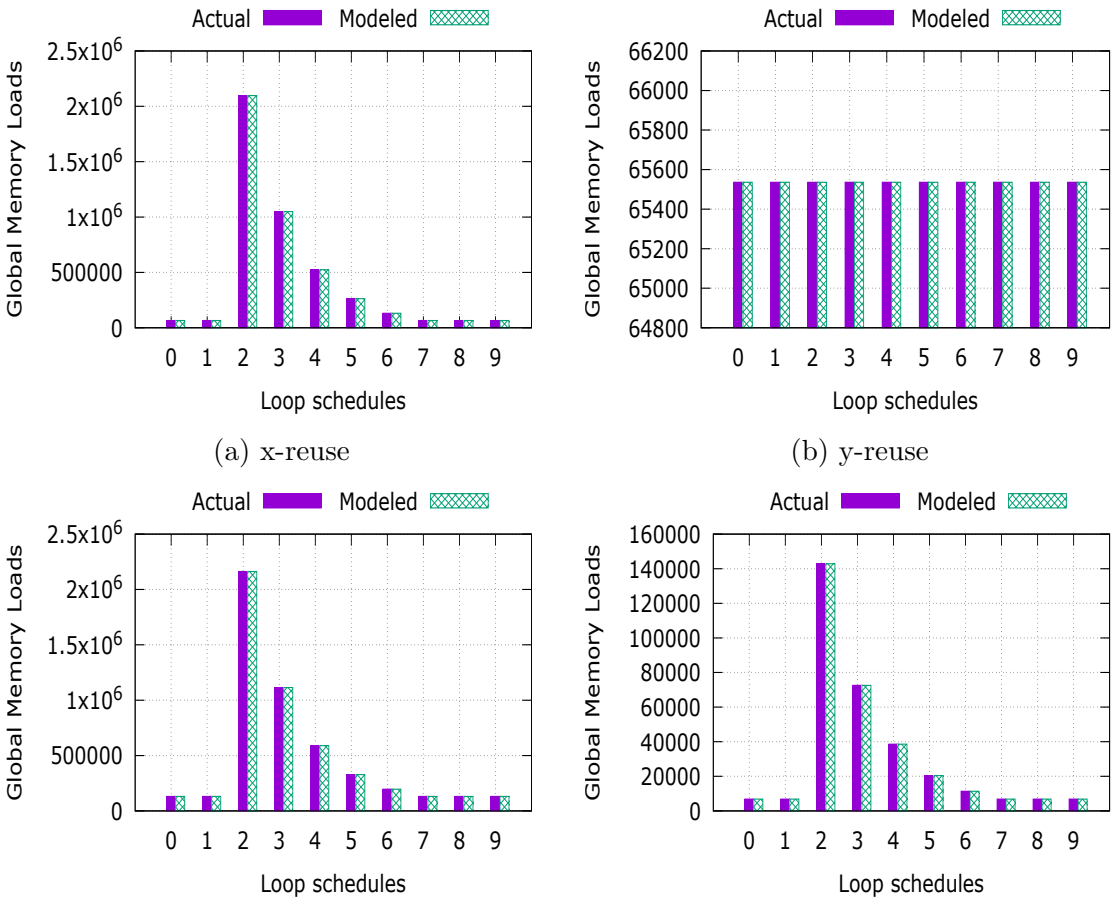


(a) Laplacian (b) Divergence (c) Himeno

Figure 7.12: GPU L2 cache modeling

launch) were exactly the same as the actual loads (profiled results) thus indicating that our proposed model was accurately predicting the memory loads. Figure 7.13 (b) indicates that for y-reuse synthetic benchmark, no matter what the loop schedule is, the memory access appears to be fully coalesced leading to the same number of global memory loads all the time. In the other three plots, the tallest bars indicate that the loop schedules had fully uncoalesced memory accesses, while the shortest bars indicate that the loop schedules had fully coalesced memory accesses, and the bars between the tallest and the shortest bars indicate that the partial memory coalescing occurred. Higher the global memory loads, higher the time taken by the threads to process the memory requests.

Figure 7.14 shows several plots that demonstrate the close correlation of the kernel performance and the memory access cost modeling. We use the coefficient of determination R^2 to measure the strength of the relationship between the kernel performance and the memory access cost in our model. R^2 is a popular indicator on how well a variable can be used to predict the value of another variable. The values of R^2 range from 0 (poor indicator) to 1 (excellent predictor). The R^2 value



(a) x-reuse

(b) y-reuse

(c) Matrix Multiplication (xy-reuse)

(d) Jacobi

Figure 7.13: Global memory loads

for all benchmarks are listed in Table 7.3 and the average value was 0.93 indicating the strong correlation between the kernel performance and the memory access cost modeling. Based on the memory access cost modeling, an optimal or a sub-optimal loop schedule was chosen by the framework. For all benchmarks tested, the speedup of the loop schedule chosen by the model against the default loop schedule chosen by the compiler are listed in Table 7.3. Since the memory access patterns in different benchmarks were different, the achieved speedup were also different. This proves the

Table 7.3: Evaluation results

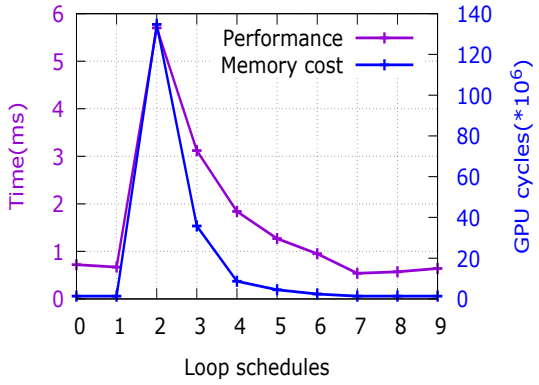
Benchmark	Source	Nested Loop Type	R^2	Speedup
x-reuse	synthetic	double	0.927	1.0
y-reuse	synthetic	double	0.683	2.74
Matrix Multiplication	Performance Test Suite	double	0.913	1.03
Jacobi	Performance Test Suite	double	0.998	1.1
Laplacian	Performance Test Suite	triple	0.999	1.05
Divergence	Performance Test Suite	triple	0.999	0.96
Matrix Transpose	CUDA SDK	double	0.943	1.37
Himeno	EPCC	triple	0.994	1.09

effectiveness of the proposed framework.

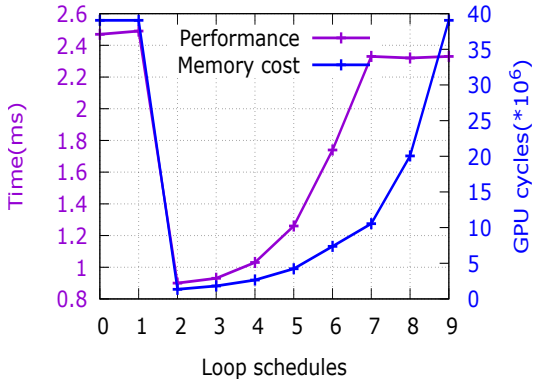
7.4.2 Supprt Vector Machine (SVM)

Support Vector Machine (SVM) [68] is a classical machine learning algorithm to perform classification and regression analysis. In this dissertation, we developed the OpenACC version of SVM [77]. Sequential Minimal Optimization (SMO) [58] is a popular algorithm used to solve the SVM QP problem by iteratively solving a series of smaller QP subproblems with only two unknown variables that are solvable analytically. Cao et al. [23] proposed the parallel SMO algorithm called PSMO to parallelize SVM by distributing the dataset into multiple computing nodes. Herrero-Lopez et al. [40] improved this algorithm and developed P2SMO algorithm for multi-class classification. Our OpenACC implementation was based on the CUDA version of P2SMO implementation. The CUDA SVM[1] was compiled by nvcc compiler with flag "-O3". The OpenACC SVM was compiled by OpenUH compiler. The dataset we used come from different sources. *adult* was from UCI [15] dataset, *letter* and

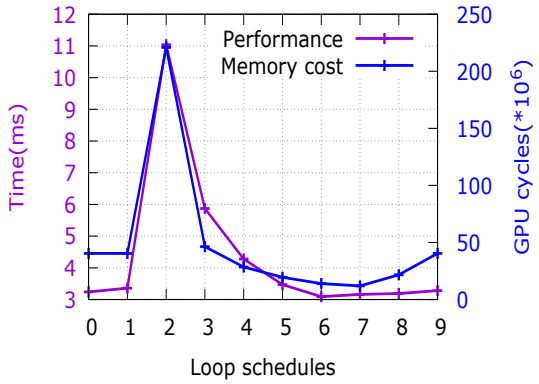
[1]https://code.google.com/p/multisvm/



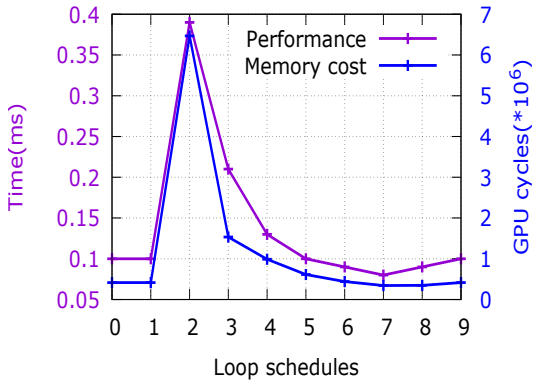
(a) x-reuse



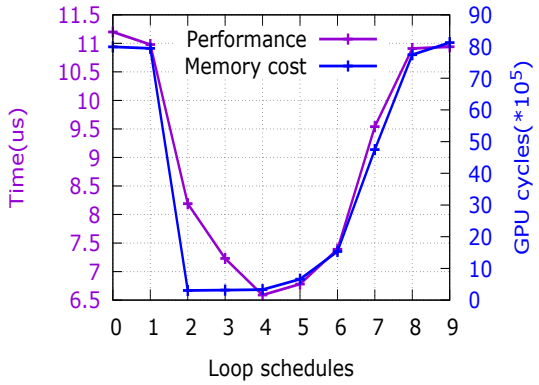
(b) y-reuse



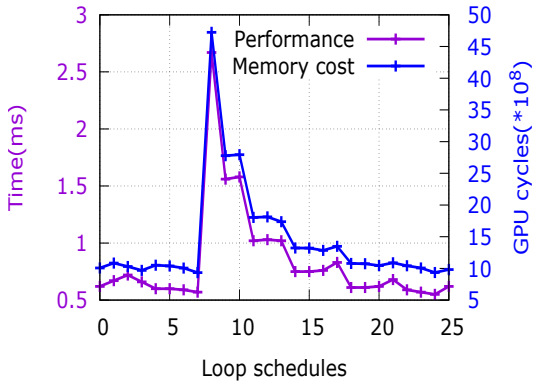
(c) Matrix Multiplication (xy-reuse)



(d) Jacobi



(e) Matrix Transpose



(f) Himeno

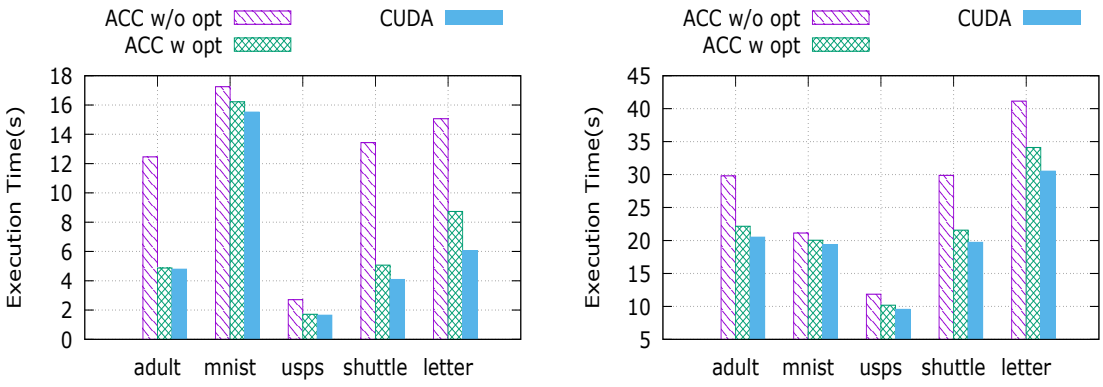
Figure 7.14: Plots demonstrating correlation of Performance vs Memory Access Cost Modeling

shuttle were from Statlog dataset [21]; both *mnist* [46] and *usps* [41] were hand-written datasets used for text recognition. The characteristics of each dataset are presented in Table 7.4 where C is the regularization parameter and γ is the stopping parameter of the SMO algorithm.

Table 7.4: Characteristics of the experiment dataset

Dataset	Training Samples	Features	Classes	C	γ
adult	32561	123	2	100	0.001
mnist	30000	780	10	10	0.125
usps	7291	256	10	100	0.001
letter	15000	16	26	100	0.001
shuttle	43500	9	7	100	0.001

Figure 7.15 (a) shows the kernel performance of ACC-SVM against CUDA-SVM. It is seen that there is significant performance improvement after applying the loop scheduling optimization enabled by our auto-tuning framework. The average kernel performance speedup for all datasets was 1.92x. The kernel performance gap between the two versions was 15.58%.



(a) Kernel Performance (b) Application Performance

Figure 7.15: Performance of ACC-SVM against CUDA-SVM

Figure 7.15 (b) shows the total execution time of the application of ACC-SVM against CUDA-SVM; this included the kernels time plus the data movement and the host code time. The average speedup after the loop scheduling optimization for the whole application was 1.23x. The application performance gap between ACC-SVM with optimization and CUDA-SVM was 7.87%.

7.5 Summary

This chapter discussed the importance of auto-tuning loop scheduling for GPU computing. We proposed an analytical model-based auto-tuning framework to identify the optimal or sub-optimal loop schedule that is better than the default loop schedule chosen by the compiler. The model used in the framework was locality-aware as it could predict the cache locality for each loop schedule. The model also predicted the total number of global memory loads and based on these information it computed a memory access cost for each loop schedule. The framework iterated over all loop schedule patterns and launch configuration space and picked the loop schedule with the least memory access cost. We analyzed the proposed framework with multiple benchmarks. The results indicated that the memory access cost modeling had strong correlation with the kernel performance and the loop schedule picked by the framework achieved 1.29x speedup over the default loop schedule chosen by the compiler. We also evaluated the framework with SVM application, and the average kernel performance speedup for all datasets was 1.92x , and the whole application speedup was 1.23x. For the future work, we will integrate more factors into the model to improve

the prediction of the loop schedule.

Chapter 8

Conclusions and Future Work

The GPU architecture has gained great success in almost last decade, but the GPU programming is still challenging. In recent years, several high-level directive-based programming models have emerged and they simplify the GPU programming while maintaining the high performance. To solve the portability issue, OpenACC has been selected as the standard among these models. Compared to low-level programming models, the directive-based models bring lots of research opportunities to shrink the performance gap between the applications using high and low level models. What parallelization techniques and optimizations are required have not been sufficiently studied in prior works. This dissertation tries to address these issues by applying different parallelization techniques and optimizations, both manually by the user and automatically in the compiler and runtime. The automatic optimizations focus on the runtime library design and implementation, reduction algorithm and multi-GPU extension. These approaches are implemented in an open source compiler.

The evaluation of these approaches indicate that they are highly competitive to the commercial compiler implementation and optimized implementation using low-level model. We also focus on the locality-aware auto-tuning for loop scheduling which tries to find an optimal mapping from a loop nest to GPU threads hierarchy. The experiment has shown that the proposed framework is effective in choosing better loop schedules than the default ones chosen by the compiler.

Both the profiling tool and debugging tool help the application developer to port the applications to GPUs easily. The profiling tool is used to profile and trace data collection. The debugging tool is used to identify the errors introduced by application parallelization or the compiler. For both profiling and debugging, the user can use the regular CPU tool for the host code and the GPU vendor provided tool for the device code. To use a single tool for both purposes, there are some alternative approaches. For the profiling, the user can use the library that supports the profiling interface defined in OpenACC 2.5. For the debugging, the compiler may generate both CPU version and device kernel for the same compute region and compare their results to check whether there is any mismatch, and insert APIs to track the host-device memory coherence at runtime [49]. Another possible solution is to make the compiler offer a bit-wise reproducibility mode with respect to CPU execution [14].

In the future work, based on our application porting experiences, lots of research still can be done to further simplify GPU programming and improve the performance using directive-based model. In the computation part, it is still the user's responsibility to specify which computation regions are offloaded to the device. It is possible that the compiler can do some analysis for all loop nests and then calculate

the computational intensity (i.e. the ratio of floating point operations per memory access) for each of the loop nests, and finally decide whether it is profitable to offload those loop nests. In the data part, the programmer's burden can be further reduced if the compiler can analyze the definition and use of all data and give some hints to the user on how the data directives should be added. Another challenging data problem is the user-defined data type including the classes and structures in C/C++ and derived types in Fortran. This is because the user-defined data that involves pointer indirection requires deep copy instead of the shallow copy in the current standard. The deep copy directives are being proposed and discussed by the OpenACC committee. In addition, since one of the goals of OpenACC is the portability among different types of accelerators, the current directives set does not fully utilize all hardware features of GPU. Therefore the compiler may perform some GPU specific optimizations when applying the loop transformation underneath. For instance, to make better use of the user manageable cache/shared memory of GPU, the application developer can use `tile` directive to partition a loops iteration space into smaller blocks and use `cache` directive to load the tile block data into shared memory. However the current compiler technique is still not mature yet to fully exploit the shared memory and texture cache in GPU deep memory hierarchy. Once these challenges are solved, the proposed loop scheduling auto-tuning framework can be improved to evaluate the new transformed kernels.

Bibliography

[1] HMPP Directives Reference Manual (HMPP Workbench 3.1).

[2] Cray C and C++ Reference Manual, 2003.

[3] NPB-CUDA. http://www.tu-chemnitz.de/informatik/PI/forschung/download/npb-gpu/, 2013.

[4] NPB-UPC. http://threads.hpcl.gwu.edu/sites/npb-upc, 2013.

[5] OpenCL Reduction. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/, November 2013.

[6] The GNU OpenMP Implementation. http://gcc.gnu.org/onlinedocs/libgomp.pdf, November 2013.

[7] 11 Tricks for Maximizing Performance with OpenACC Directives in Fortran. http://www.pgroup.com/resources/openacc_tips_fortran.htm, 2014.

[8] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2014.

[9] NVIDIA Kepler GK110 Architecture Whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2014.

[10] OpenACC. http://www.openacc-standard.org, 2014.

[11] EPCC OpenACC Benchmarks. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite, 2015.

[12] KernelGen Performance Test Suite. `https://hpcforge.org/plugins/mediawiki/wiki/kernelgen/index.php/Performance_Test_Suite`, December 2015.

[13] George Almási, Clin Caşcaval, and David A Padua. Calculating Stack Distances Efficiently. In *ACM SIGPLAN Notices*, volume 38, pages 37–43. ACM, 2002.

[14] Andrea Arteaga, Oliver Fuhrer, and Torsten Hoefler. Designing Bit-Reproducible Portable High-Performance Applications. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1235–1244. IEEE, 2014.

[15] Arthur Asuncion and David Newman. UCI Machine Learning Repository, 2007.

[16] 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.

[17] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.

[18] David Bailey et al. The NAS Parallel Benchmarks. *NASA Ames Research Center*, 1994.

[19] Matthew Baker, Swaroop Pophale, Jean-Charles Vasnier, Haoqiang Jin, and Oscar Hernandez. Hybrid Programming using OpenSHMEM and OpenACC. In *First OpenSHMEM Workshop: Experiences, Implementations and Tools*, pages 74–89. Springer, 2014.

[20] Kristof Beyls and Erik DHollander. Reuse Distance as a Metric for Cache Behavior. 14:350–360, 2001.

[21] P Brazdil and J Gama. Statlog Datasets. *Inst. for Social Research at York Univ., http://www.liacc.up.pt/ML/statlog/datasets.html*, 1999.

[22] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive Programming of GPU Clusters with OmpSs. In *Parallel and Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568. IEEE, 2012.

[23] Li Juan Cao, S Sathiya Keerthi, Chong-Jin Ong, Jian Qiu Zhang, and Henry P Lee. Parallel Sequential Minimal Optimization for the Training of Support Vector Machines. *IEEE Transactions on Neural Networks*, 17(4):1039–1049, 2006.

[24] P. Charoenrattanaruk and C.F. Eick. Design and Evaluation of a High Performance Computing Framework for the CLEVER Clustering Algorithm, 2011.

[25] Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar. Dynamic Task Parallelism with a GPU Work-Stealing Runtime System. In *Languages and Compilers for Parallel Computing*, pages 203–217. Springer, 2013.

[26] J.H. Chen, A. Choudhary, B. De Supinski, M. DeVries, ER Hawkes, S. Klasky, WK Liao, KL Ma, J. Mellor-Crummey, N. Podhorszki, et al. Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D. *Computational Science and Discovery*, 2(1):015001, 2009.

[27] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, 2012.

[28] Xiang Cui, Yifeng Chen, Changyou Zhang, and Hong Mei. Auto-tuning Dense Matrix Multiplication for GPGPU with Cache. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 237–242. IEEE, 2010.

[29] C.F. Eick, R. Parmar, W. Ding, T.F. Stepinski, and J. Nicot. Finding Regional Co-location Patterns for Sets of Continuous Variables in Spatial Datasets. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, GIS '08, pages 30:1–30:10, New York, NY, USA, 2008. ACM.

[30] Leback et al. The PGI Fortran and C99 OpenACC Compilers. *Cray User Group*, 2012.

[31] L.A. Feldkamp, L.C. Davis, and J.W. Kress. Practical Cone-beam Algorithm. *JOSA A*, 1(6):612–619, 1984.

[32] Michael Frumkin, Haoqiang Jin, and Jerry Yan. Implementation of NAS Parallel Benchmarks in High Performance Fortran. *NAS Techinical Report NAS-98-009*, 1998.

[33] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.

[34] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[35] T. Han and T.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] Mark Harris. Optimizing Parallel Reduction in CUDA. *NVIDIA Developer Technology*, 6, 2007.

[37] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel Prefix Sum (Scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[38] A Hart, R Ansaloni, and A Gray. Porting and Scaling OpenACC Applications on Massively-parallel, GPU-accelerated Supercomputers. *The European Physical Journal Special Topics*, 210(1):5–16, 2012.

[39] O. Hernandez, W. Ding, B. Chapman, C. Kartsaklis, R. Sankaran, and R. Graham. Experiences with High-level Programming Directives for Porting Applications to GPUs. *Facing the Multicore-Challenge II*, pages 96–107, 2012.

[40] Sergio Herrero-Lopez, John R Williams, and Abel Sanchez. Parallel Multiclass Classification using SVMs on GPUs. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 2–11. ACM, 2010.

[41] Jonathan J Hull. A Database for Handwritten Text Recognition Research. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(5):550–554, 1994.

[42] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical report, NAS-99-011, NASA Ames Research Center, 1999.

[43] Guido Juckeland, William C. Brantley, Sunita Chandrasekaran, Barbara M. Chapman, Shuai Che, Mathew E. Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John A. Stratton, Alexey Titov, Ke Wang, G. Matthijs van Waveren, Brian Whitney, Sandra

Wienke, Rengan Xu, and Kalyan Kumaran. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*, pages 46–67, 2014.

[44] A. Kak and M. Slaney. *Principles of Computerized Tomographic Imaging*. Society for Industrial and Applied Mathematics, 2001.

[45] Toshiya Komoda, Shinobu Miwa, Hiroshi Nakamura, and Naoya Maruyama. Integrating Multi-GPU Execution in an OpenACC Compiler. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 260–269. IEEE, 2013.

[46] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[47] HyoukJoong Lee, Kevin J Brown, Arvind K Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-Aware Mapping of Nested Parallel Patterns on GPUs. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 63–74. IEEE, 2014.

[48] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *SC 2010*, pages 1–11. IEEE Computer Society, 2010.

[49] Seyong Lee, Dong Li, and Jeffrey S Vetter. Interactive Program Debugging and Optimization for Directive-based, Efficient GPU Computing. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 481–490. IEEE, 2014.

[50] Seyong Lee and Jeffrey S Vetter. Early Evaluation of Directive-based GPU Programming Models for Productive Exascale Computing. In *SC 12*, pages 1–11. IEEE Computer Society Press, 2012.

[51] John M. Levesque, Ramanan Sankaran, and Ray Grout. Hybridizing S3D into an Exascale Application Using OpenACC: An Approach for Moving to Multi-petaflops and Beyond. In *Proceedings of*, SC '12, pages 15:1–15:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[52] Chunhua Liao, Oscar Hernandez, Barbara M. Chapman, Wenguang Chen, and Weimin Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.

[53] J. MacQueen et al. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, page 14. California, USA, 1967.

[54] Calvin Montgomery, Jeffrey L Overbey, and Xuechao Li. Autotuning OpenACC Work Distribution via Direct Search. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, page 38. ACM, 2015.

[55] Ramachandra Nanjegowda, Oscar Hernandez, Barbara Chapman, and Haoqiang H Jin. Scalability Evaluation of Barrier Algorithms for OpenMP. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 42–52. Springer, 2009.

[56] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 37–48. IEEE, 2014.

[57] Simon J Pennycook, Simon D Hammond, Stephen A Jarvis, and Gihan R Mudalige. Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS LU Benchmark. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):23–29, 2011.

[58] John C Platt. 12 Fast Training of Support Vector Machines using Sequential Minimal Optimization. *Advances in kernel methods*, pages 185–208, 1999.

[59] G. Pullan. Cambridge CUDA Course. `http://www.many-core.group.cam.ac.uk/archive/CUDAcourse09/`, 2009.

[60] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE Intl Symp. on IISWC*, pages 137–148. IEEE, 2011.

[61] Shahzeb Siddiqui and Saber Feki. Historic Learning Approach for Auto-tuning OpenACC Accelerated Scientific Applications. pages 224–235, 2014.

[62] Kyle Spafford, Jeremy Meredith, Jeffrey Vetter, Jacqueline Chen, Ray Grout, and Ramanan Sankaran. Accelerating S3D: A GPGPU Case Study. In *Euro-Par 2009–Parallel Processing Workshops*, pages 122–131. Springer, 2010.

[63] Tao Tang, Xuejun Yang, and Yisong Lin. Cache Miss Analysis for GPU Programs Based on Stack Distance Profile. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 623–634. IEEE, 2011.

[64] Xiaonan Tian, Rengan Xu, Yonghong Yan, Sunita Chandrasekaran, Deepak Eachempati, and Barbara Chapman. Compiler Transformation of Nested Loops for General Purpose GPUs. *Concurrency and Computation: Practice and Experience*, 28(2):537–556, 2016.

[65] Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara Chapman. Compiling A High-Level Directive-based Programming Model for GPGPUs. In *Intl. workshop on LCPC 2013*, pages 105–120. Springer International Publishing, 2014.

[66] S.Z. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu. CUDA-lite: Reducing GPU Programming Complexity. *Languages and Compilers for Parallel Computing*, pages 1–15, 2008.

[67] D. Unat, X. Cai, and S.B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[68] Vladimir Vapnik. *The Nature of Statistical Learning Theory*. Springer Science and Business Media, 2013.

[69] Cheng Wang, Rengan Xu, Sunita Chandrasekaran, Barbara Chapman, and Oscar Hernandez. A Validation Testsuite for OpenACC 1.0. In *Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), 2014 IEEE 28th International*, pages 1407–1416. IEEE, 2014.

[70] N. Whitehead and A. Fit-Florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. 2011.

[71] M. Wolfe. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.

[72] Xingfu Wu and Valerie Taylor. Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-Scale Multicore Clusters. *The Computer Journal*, 55(2):154–167, 2012.

[73] Rengan Xu, Sunita Chandrasekaran, and Barbara Chapman. Exploring Programming Multi-GPUs Using OpenMP and OpenACC-Based Hybrid Model.

In *Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1169–1176. IEEE, 2013.

[74] Rengan Xu, Sunita Chandrasekaran, Barbara Chapman, and Christoph F Eick. Directive-based Programming Models for Scientific Applications-A Comparison. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, pages 1–9. IEEE, 2012.

[75] Rengan Xu, Sunita Chandrasekeran, Xiaonan Tian, and Barbara Chapman. An Analytical Model-based Auto-tuning Framework for Locality-aware Loop Scheduling. In *Proceedings of 2016 International Supercomputing Conference*, 2016.

[76] Rengan Xu, Maxime Hugues, Henri Calandra, Sunita Chandrasekaran, and Barbara Chapman. Accelerating Kirchhoff Migration on GPU using Directives. In *Proceedings of the First Workshop on Accelerator Programming using Directives*, pages 37–46. IEEE, 2014.

[77] Rengan Xu, Dounia Khaldi, Abid Malik, and Barbara Chapman. ACC-SVM: Accelerating SVM on GPUs using OpenACC. In *Proceedings of the First Workshop of Mission-Critical Big Data Analytics (MCBDA 2016)*, 2016.

[78] Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, and Barbara Chapman. Multi-GPU Support on Single Node Using Directive-Based Programming Model. *Scientific Programming*, 2015:15, 2015.

[79] Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, Yonghong Yan, and Barbara M. Chapman. NAS Parallel Benchmarks for GPGPUs Using a Directive-Based Programming Model. In *Languages and Compilers for Parallel Computing - 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers*, pages 67–81, 2014.

[80] Rengan Xu, Xiaonan Tian, Yonghong Yan, Sunita Chandrasekaran, and Barbara Chapman. Reduction Operations in Parallel Loops for GPGPUs. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, pages 10:10–10:20, New York, NY, USA, 2007. ACM.