A COMPILER OPTIMIZATION FRAMEWORK FOR DIRECTIVE-BASED GPU COMPUTING

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

Xiaonan Tian

August 2016

A COMPILER OPTIMIZATION FRAMEWORK FOR DIRECTIVE-BASED GPU COMPUTING

Xiaonan Tian

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Edgar Gabriel
Dept. of Computer Science

Dr. Jaspal Subhlok
Dept. of Computer Science

Dr. Weidong Shi
Dept. of Computer Science

Dr. Gregory Rodgers
Advanced Micro Devices Research, Inc.

Dean, College of Natural Sciences and Mathematics

Acknowledgements

Though this dissertation is under my name, a number of great people have contributed to its production. I would like to express here my sincere appreciation for all those who made this PhD work possible.

My deepest gratitude is to my committee chair and advisor, Dr. Barbara Chapman, offered and supported me from the beginning of my Ph.D program. I have been amazingly fortunate to have an advisor who gave me trust and freedom to explore on my own. She has been the most important people for my graduate research. Her patience and support helped me overcome many crisis siutations and finish this dissertation. It was an honor for me to be a member of Dr.Chapman's HPCTools group and work with a number of brilliant teammates.

I also would like to thank my other committee members, Dr. Edgar Gabriel, Dr. Larry Shi, Dr. Jaspal Subhlok, and Dr. Gregory Rodgers, who all took the time to review my work and offer their valued feedback. Specifically, I am grateful to Dr. Oscar R. Hernandez from the Oak Ridge National Laboratory for giving me generous help during my 2014 summer internship at ORNL and Dr. Gregory Rodgers from AMD research. He is my committee member and was my mentor during my 2015 summer internship at AMD research. Additionally, I very much appreciate their guidance for my Ph.D research.

As a student in the HPCTools research group, I owe my gratitude to several senior group members. Dr.Yonghong Yan and Dr.Sunita Chandrasekeran guided my research at the early years of my Ph.D research. Dr.Deepak Eachempati and

Dr.Dounia Khaldi supervised my work in the past three years. They provided appreciated suggestions for the organization of the dissertaion. They have my gratitude.

I am also thankful to Rengan Xu who is my close teammate working on the OpenACC compiler. His expertise knowledge of application porting helps me impove compiler optimization for GPUs. I greatly value his friendship and deeply appreciate his help.

Most important, none of this would have been possible without the love and endless support from my parents and my lovely spouse Dayin He. They encouraged me throughout this endeavor. This dissertation is dedicated to my family.

Finally, I appreciate NVIDIA and Total who funded the OpenACC compiler research in this dissertation.

A COMPILER OPTIMIZATION FRAMEWORK FOR DIRECTIVE-BASED GPU COMPUTING

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

Xiaonan Tian

August 2016

Abstract

In the past decade, accelerators, commonly Graphics Processing Units (GPUs), have played a key role in achieving Petascale performance and driving efforts to reach Exascale. However, significant advances in programming models for accelerator-based systems are required in order to close the gap between achievable and theoretical peak performance. These advances should not come at the cost of programmability. Directive-based programming models for accelerators, such as OpenACC and OpenMP, help non-expert programmers to parallelize applications productively and enable incremental parallelization of existing codes, but typically will result in lower performance than CUDA or OpenCL. The goal of this dissertation is to shrink this performance gap by supporting fine-grained parallelism and locality-awareness within a chip.

We propose a comprehensive loop scheduling transformation to help users more effectively exploit the multi-dimensional thread topology within the accelerator. An innovative redundant execution mode is developed in order to reduce unnecessary synchronization overhead. Our data locality optimizations utilize the different types of memory in the accelerator. Where the compiler analysis is insufficient, an explicit directive-based approach is proposed to guide the compiler to perform specific optimizations.

We have chosen to implement and evaluate our work using the OpenACC programming model, as it is the most mature and well-supported of the directive-based accelerator programming models, having multiple commercial implementations. However, the proposed methods can also be applied to OpenMP. For the hardware platform, we choose GPUs from Advanced Micro Devices (AMD) and

NVIDIA, as well as Accelerated Processing Units (APUs) from AMD.

We evaluate our proposed compiler framework and optimization algorithms with SPEC and NAS OpenACC benchmarks; the result suggests that these approaches will be effective for improving overall performance of code executing on GPUs. With the data locality optimizations, we observed up to 3.22 speedup running NAS and 2.41 speedup while running SPEC benchmarks. For the overall performance, the proposed compiler framework generates code with competitive performance to the state-of-art of commercial compiler from NVIDIA PGI.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, there has been a shift from systems relying on multi-core processors to systems utilizing many-core processors, often in heterogeneous architecture configurations. This shift has been most prominently realized in the increasing use of Graphics Processing Units (GPUs) as general-purpose computational accelerators, such as those provided by NVIDIA and AMD. These accelerators provide *massively parallel* computing capabilities to users while preserving the flexibility provided by CPUs for different workloads. However, effectively exploiting the full potential of accelerators requires dealing with the programming challenges faced when mapping computational algorithms to hybrid and heterogeneous architectures.

Low-level heterogeneous programming models, such as CUDA [3] and OpenCL [9],

offer programming interfaces with execution models that closely match general-purpose GPU (GPGPU) architectures. Effectively utilizing these interfaces to create highly optimized applications requires programmers to thoroughly understand the underlying architecture. In addition, they must be able to significantly change and adapt program structures and algorithms. This affects productivity, portability and performance.

An alternative approach would be to use high-level, directive-based programming models, e.g., OpenACC [8] and OpenMP [15], to achieve the same goal. These models allow the user to insert both directives and runtime calls into existing Fortran or C/C++ source code, enabling a portion of their code to execute on the accelerator. Using directives, programmers may give hints to compilers to perform certain transformations and optimizations on the annotated code regions. The user can insert directives incrementally to parallelize and optimize a program, enabling a productive migration path for legacy applications.

OpenMP, a parallel programming interface comprising a set of compiler directives, library routines and environment variables, has established itself as the de facto standard for writing parallel programs in C/C++ and Fortran on shared memory multi-core CPU systems. OpenMP added an initial extension to its feature set for making use of accelerators in version 4.0, and further extended its accelerator support in version 4.5 (released in November of 2015). The *target* directive identifies the offload region running on the accelerator, wherein a massive number of threads may be organized into teams as prescribed by the programmer. OpenMP also provides *simd* constructs for directing compiler vectorization within each thread. As

of this writing, there is no compiler that fully supports the OpenMP 4.5 specification, though it is expected that compliant-implementations will be available soon. Given its comprehensive support for parallelizing codes for CPU targets as well as its evolving support for heterogeneous devices, OpenMP has the potential to fully exploit parallelism on systems containing multi-core host processors and additional accelerators, making it a strong candidiate for heterogeneous computing.

OpenACC, a directive-based parallel programming interface in many ways inspired by OpenMP, was the first standardized specification released to facilitate programming of accelerators. Unlike OpenMP which offers a primarily prescriptive interface for expressing parallelism, OpenACC provides both prescriptive and descriptive mechanisms for this purpose. In particular, much of OpenACC is intended for *describing* additional program information to compilers, so that it may more effectively generate code for execution on accelerators. OpenACC's descriptive interface allows flexibility for the compiler to interpret how to map high-level language abstractions to the hardware layer. Several vendors have delivered OpenACC compilers since its first specification published in 2011, and in general its support for accelerator-based computing has evolved in response to user feedback more rapidly compared to OpenMP.

The OpenMP and OpenACC accelerator models have not evolved independently from each other. For example, OpenMP adopted unstructured data directives and asynchronous execution of offload regions from OpenACC 2.0 into its 4.5 accelerator model specification. Conversely, OpenACC enacted semantic changes to some of its data clauses in version 2.5, resulting in consistency with the corresponding behavior

in the OpenMP 4.0 specification. In general, user experience with OpenACC has informed the direction taken by OpenMP in its accelerator specification [5].

The performance gap [32, 17] between programs accelerated with OpenACC/OpenMP using its relatively high-level abstractions compared to lower-level CUDA/OpenCL versions indicates that more optimization research is required. In this dissertation, we present a compiler framework for interpreting the OpenACC programming model and propose a comprehensive loop scheduling implementation covering coarse- and fine-grained parallelism. Based on this compiler framework, a set of compiler optimizations are proposed to reduce synchronization and improve data locality. Prescriptive directive-based solutions for explicitly guiding compiler optimization are also proposed in order to direct loop scheduling transformations and further enhance data locality.

1.2 Contributions of this Dissertation

The contributions are summarized as follows:

1. We constructed a prototype open-source OpenACC compiler based on a branch of the industrial strength Open64 compiler. Our OpenACC compiler accepts applications written in the C and Fortran base languages and targets NVIDIA GPUs and AMD GPUs/APUs. This implementation serves as a compiler infrastructure for researchers to explore advanced compiler techniques for OpenACC, to extend OpenACC to other programming languages, or to build performance

tools used with OpenACC programs.

2. We designed a rich set of loop-scheduling strategies within the compiler to efficiently distribute kernels or parallel loops to the threading architectures of GPU accelerators. Our findings provide guidance for users to adopt suitable loop schedules depending on the application requirements. Classical dependence analysis was used to guide scheduling of nested loops and exploit an unconventional redundant execution mode for the purpose of reducing synchronization between each level of parallelism.

3. We present both implicit and explicit compile-time data locality optimization techniques to more efficiently utilize deep memory hierarchies in GPUs. This optimization can be separated into several different points: 1) compile-time read-only array/pointer detection for each offload region in order to utilize the read-only data cache; 2) extension of the Scalar Replacement (SR) algorithm to fully exploit the rich set of register file resources; and 3) new clauses to assist the compiler to reduce register file usage.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 provides an overview of GPU architectures and their respective low-level and high-level programming models. Chapter 3 discusses the OpenACC programming model. Chapter 4 reviews the related work for existing directive-based compiler implementations. Chapter 5 highlights the OpenACC compiler framework in OpenUH. Chapter 6 explains two

sets of loop scheduling transformations for GPUs and our redundant execution mode for multiple levels of parallelism. Chapter 7 describes data locality optimizations on the GPU. Chapter 8 presents an evaluation of our compiler implementation and optimization algorithms. Concluding remarks are given in Chapter 9 along with potential avenues for future work.

Chapter 2

GPU Architectures and Their Programming Models

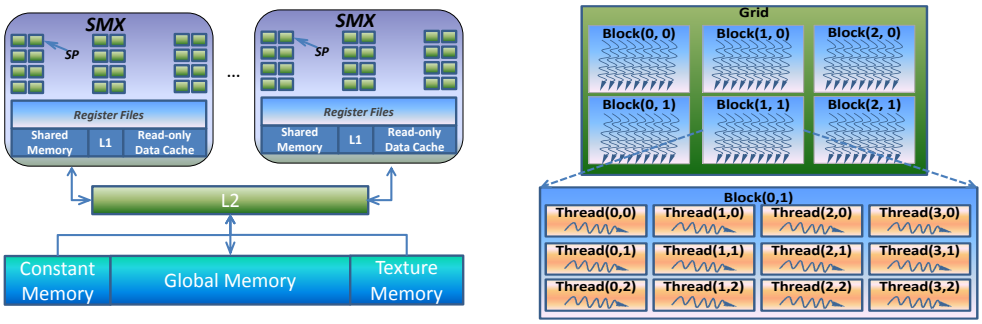
GPUs, when functioning as accelerators, are capable of executing a massive number of threads in parallel. To support this capability, GPUs have significantly different architectures compared to CPUs. So, in Section 2.1 we first give a brief introduction on the GPU architecture and its programming models for expressing general-purpose parallelism, i.e., not specifically graphics-oriented. OpenCL is a portable computing framework that defines a programming language extension to C and assorted APIs for programming GPUs from both NVIDIA and AMD. NVIDIA's CUDA computing platform provides a variety of tools for programming NVIDIA GPUs apart from OpenCL, including the CUDA C extension to C/C++. In our work, we utilize CUDA C and the CUDA runtime interface for supporting low-level programming of NVIDIA GPUs, and OpenCL for corresponding low-level programming of AMD

GPUs. Therefore, both the CUDA and OpenCL programming models are introduced in Section 2.2.

The trend towards energy efficiency has led to unified architectures in which CPUs and GPUs are integrated onto the same die. In such an architecture, both CPUs and GPUs can share the same virtual memory space and eliminate the need for memory transfers between CPUs and GPUs. AMD's Accelerated Processing Unit (APU) is an example of such an architecture, and it supports zero-copy through pointer passing between the CPU and GPU. This feature can greatly reduce the data movement between the host and accelerator. Section 2.1.3 highlights the AMD APU system based on **H**eterogeneous **S**ystem **A**rchitecture (**HSA**).

2.1 GPU Architectures

The processor architecture of GPUs and CPUs are fundamentally different. Modern GPUs are throughput-oriented devices made up of hundreds of processing cores. They maintain a high throughput and hide memory latency by supporting multi-thread switching among thousands of threads. Each GPU supports the concurrent execution of hundreds to thousands of threads, following the single-instruction multiple-threads (SIMT) model. Typically, the GPU has a two-level hierarchical architecture. It is made of vector processors at the top level, and each vector processor contains a large number of scalar processors at the lower level. Since NVIDIA and AMD use their own terms to describe their architectures, we discuss each of them separately.



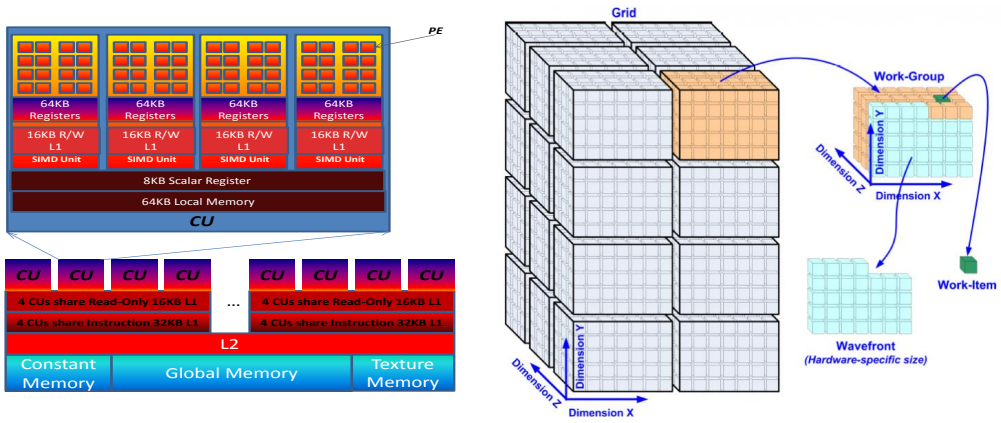
(a) NVIDIA GPU Architecture (b) CUDA Threads Model

Figure 2.1: NVIDIA GPU Architecture and Threads Model

2.1.1 NVIDIA GPU Architectures

In Figure 2.1(a), the NVIDIA GPU consists of multiple Next Generation Streaming Multiprocessors (SMXs), and each SMX consists of many scalar processors (SPs, also referred to as cores). Each thread is executed by an SP. The smallest scheduling and execution unit is called a warp which has 32 threads. Warps of threads are grouped together into a thread block, and blocks are grouped into a grid. Figure 2.1(b) shows how the blocks can be organized into a one-, two- or three-dimensional grid of thread blocks in the CUDA programming model. Each thread has its own unique thread ID which can be identified by $threadIdx.x$, $threadIdx.y$ and $threadIdx.z$ in each block. Thread blocks cannot synchronize with each other, but the threads within a block can do so. Each SMX can have at most 64 warps or 16 blocks allocated at a time. Due to resource limitations (registers per thread and shared memory per block), the number of warps and blocks may be less.

Each SMX has a number of different types on-chip memory resources. The on-chip software managed cache, termed *shared memory*, is shared by SPs within the



(a) AMD GPU Architecture (b) OpenCL Threads Model

Figure 2.2: AMD GCN GPU architecture and OpenCL Threads Model

same SMX. A fixed number of registers are logically partitioned among the threads running on the same SMX. The L1 cache and read-only data cache are also shared by SPs within each SMX. All the SMXs share the L2 cache, global memory and two kinds of special purpose memory: constant memory and texture memory. Accesses to constant memory may be jointly issued by threads within the same warp, and accesses to texture memory are optimized for 2D or 3D spatial locality. Both constant memory and texture memory are read-only.

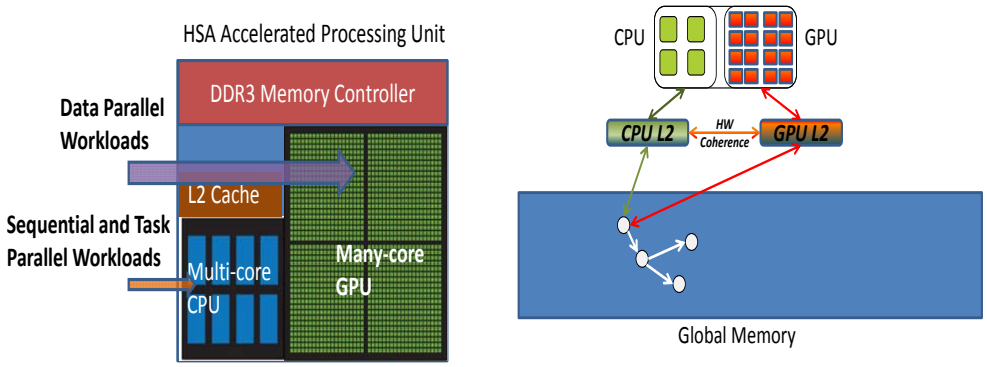
2.1.2 AMD GPU Architecture

In Figure 2.2(a), the AMD **G**raphics-**C**ore-**N**ext (**GCN**) GPU consists of multiple Compute Units (CU). Each CU is partitioned into four separate SIMD units. Each SIMD consists of 16 processing elements (PEs). In the AMD execution model, a thread is called a work-item, where an individual work-item is executed on a single PE. A wavefront, consisting of 64 work-items, is analogous to NVIDIA's concept of

a warp. Each SIMD unit has the capacity of 1 to 10 wavefronts. Once launched, wavefronts do not migrate across SIMD units. One or more wavefronts of work-items are grouped into a work-group, and one or more work-groups execute on a single CU. In Figure 2.2(b), each work-item has a unique ID and can be organized into a multi-dimensional work-group. The work-groups are further grouped into a multi-dimensional grid. Thread synchronization is only supported within a work-group.

2.1.3 AMD APU Architecture

Traditionally, CPUs and GPUs have been designed as separate processing architectures and do not work together efficiently. Each has a separate memory space, requiring an application to explicitly copy data from CPU to GPU and then back again. A program running on the CPU queues work for the GPU using system calls through a device driver stack that is managed by a distinct scheduler. This introduces significant dispatch latency, making the process worthwhile only when the amount of parallel computation is substantial enough to offset this overhead. The AMD APU is an architecture consisting of a CPU and GPU integrated onto a single chip. The Heterogeneous System Architecture (HSA) is a cross-vendor set of specifications that allow for the integration of CPU and GPU on the same bus with a shared memory space and share task queues. The goal of HSA is to reduce communication latency between CPUs and GPUs, relieving the burden of moving data between devices' disjoint memories. The HSA-based APU can efficiently support a wide assortment of data-parallel and task-parallel programming models, as shown in



(a) AMD HSA Architecture (b) AMD APU Unified Memory Access

Figure 2.3: AMD HSA based Unified Memory System

Figure 2.3(a). In this dissertation, APU refers specifically to an HSA-based APU that provides a truly unified memory architecture.

Figure 2.3(b) shows how the integrated GPU has access to the entire CPU memory space, and the CPU can pass a virtual memory pointer to GPU kernels without moving data. The main benefit is that it reduces the cost of data movement when CPU and GPU access the same data set.

2.2 GPU Low-Level Programming Models

As the architecture of the GPU is markedly different from most commodity CPUs, GPUs can achieve high performance computing capability by exploiting data parallelism. This requires a programming model that is substantially different from traditional CPU parallel programming models. In this section, we briefly describe the CUDA [3] and OpenCL [9] programming interfaces, the two most widely used

low-level programming models on GPUs.

Achieving inter-thread data locality is particularly important when programming GPUs, as compared to CPUs, to achieve peak performance. In traditional CPU multi-threading programming, data locality is often exploited within a thread because spatial data locality within a thread can increase cache utilization. Locality across threads, however, may increase false-sharing due to cache coherence on a typical shared memory, multi-core system. Therefore, each thread usually takes a block of data rather than one element. However, a GPU issues memory accesses with respect to a group of threads (e.g., a warp or wavefront) rather than a single thread as on a CPU. Memory operations are issued per warp/wavefront, just as with other instructions. Threads in a warp/wavefront provide memory addresses, and the hardware determines into which cache lines those addresses fall and requests the needed lines. Memory is accessed at a 32-byte granularity. Consider a scenario in which threads in a warp request data from consecutive memory addresses. The data can be ready in 5 or less transactions since all the data transferred is efficiently used. This is referred to as a *coalesced* memory access. In another other scenario, a scattered access pattern may generate 32 memory transactions to make the data ready for all the threads within a warp. As a result, the data locality among threads within the same warp plays a critical role in optimizing global memory access performance.

```
__global__ void va_kernel(float* A, float* B, float* C, int n){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i<n)
      *(C+i) = *(A+i) + *(B+i);
}

void vectoradd(float* A, float* B, float* C, int n){
  float *d_A, *d_B, *d_C;
  int isize = n*sizeof(float);
  cudaMalloc((void **)&d_A, isize);
  cudaMalloc((void **)&d_B, isize);
  cudaMalloc((void **)&d_C, isize);
  cudaMemcpy(d_A, h_A, isize, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, isize, cudaMemcpyHostToDevice);
  int threadsPerBlock = 256;
  int blocksPerGrid = (n+255)/256;
  va_kernel<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, n);
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```

Figure 2.4: Vector Addition in CUDA

2.2.1 CUDA Programming Model

The CUDA programming language was introduced by NVIDIA to program its massively parallel GPU architectures and the CUDA kernel function follows to a Single Program Multiple Data (SPMD) programming model. The GPU is treated as a co-processor that executes the compute-intensive kernel functions. CUDA provides an abstraction for describing a hierarchy of threads, different types of memories and synchronization. The hierarchy of threads abstraction tightly matches the GPU hardware architecture. In CUDA, threads have a two-level hierarchy. A grid is a set of thread blocks that execute a kernel function. Each grid consists of blocks of threads. Each block is composed of hundreds of threads. Threads within one block can share data using shared memory and can be synchronized at a barrier. All threads within a block are executed concurrently on a single SMX. The programmer specifies the number of threads per block and the number of blocks per grid during the kernel launch.

Figure 2.4 shows a vector addition written using CUDA C and the CUDA library API. The keyword "__global__" identifies the GPU kernel function which runs on the GPU. Other functions are still executed on the CPU. In this kernel function, each thread operates on its own portion of the data and performs one addition. On the CPU side, CUDA runtime functions are called to allocate device memory and transfer the data onto the device (shown between lines 10 and 14). The GPU kernel launch is called with a specified thread hierarchy (line 17). After the computation finishes, the data is copied back from the GPU device memory to the CPU host memory.

2.2.2 OpenCL Programming Model

OpenCL is an open-standard language for programming GPUs and is supported by all major vendors of GPUs. Like CUDA, OpenCL also provides mechanisms to describe thread topology, memory hierarchy of the GPU and synchronization. NDRange (or Grid) describes the space of work-items, which can be one-, two- or three-dimensional, as shown in Figure 2.2(b). Unlike CUDA, OpenCL programmers can just specify the total number of work-items and let the OpenCL implementation determine how to compose them into work-groups. OpenCL provides the following disjoint address spaces: global, local, constant and private. Private Memory can only be accessed by each work-item. Variables inside a kernel function not declared with an address space qualifier are in the private address space. Local Memory is a low-latency, high-bandwidth on-chip software managed cache that is shared by the work-items within the same work-group. Constant Memory and Texture Memory

```
const char *kernelSource =
  "__kernel void va_kernel(__global float* A, __global float* B,\n"
  " __global float* C, int n)\n"
  "{\n"
    "int i = get_global_id(0);\n"
    "if(i<n)\n"
    " *(C+i) = *(A+i) + *(B+i);}\n"

void vectoradd(float* A, float* B, float* C, int n){
    cl_mem d_a, cl_mem d_b, cl_mem d_c;
    cl_platform_id cpPlatform;
    cl_device_id device_id;
    cl_context context;
    cl_command_queue queue;
    cl_program program;
    cl_kernel kernel;
    int isize = n*sizeof(float);
    //Initialize the device
    ...
    program = clCreateProgramWithSource(context, 1,
                    (const char **) & kernelSource, NULL, &err);
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    kernel = clCreateKernel(program, "va_kernel", &err);

    d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, isize, ...);
    d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, isize, ...);
    d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, isize, ...);
    clEnqueueWriteBuffer(queue, A, CL_TRUE, 0, isize, A, ...);
    clEnqueueWriteBuffer(queue, B, CL_TRUE, 0, isize, B, ...);

    clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
    clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);

    clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &n, ...);
    clFinish(queue);
    clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0,
                                isize, C, ... );

    // release OpenCL resources
    ...
}
```

Figure 2.5: Vector Addition in OpenCL

are read-only memory. Texture Memory is allocated using the OpenCL Image APIs. Global Memory, Constant Memory and Texture Memory can be accessed by all the work-items in the grid and reside in the off-chip DRAM. OpenCL provides intrinsic functions to synchronize work-items inside a work-group. However, there is no way to communicate among work-groups.

Figure 2.5 shows a vector addition in OpenCL. All the functions prefixed with "cl" are defined by the OpenCL runtime library. Because the names in the OpenCL

```
1 void vectoradd(float* A, float* B, float* C, int n){
2   SNK_INIT_LPARM(lparm1, 512)
3   va_kernel(A, B, C, n, lparm1);
4 }
```

Figure 2.6: Vector Addition in APU

runtime APIs are too lengthy, some of the statements shown in the figure are shortened using ellipses. The "__kernel" keyword identifies the OpenCL kernel function and "__global" indicates the global data. As can be observed from the OpenCL code, programmers are required to initialize devices, manage the data movement, and launch the kernel by calling the OpenCL runtime. As when programming using CUDA, the OpenCL programmer must explicitly manage many low-level details to map data to the suitable memory space and to manage the synchronization between host and device. OpenCL exposes quite a lot of low-level details of GPU architectures and consequentially OpenCL programming can entail a steep learning curve.

For the APU architecture, another approach is used to make programming on the CPU side less burdensome. CLOC (CL Offline Compiler) takes the OpenCL kernel functions and generates a Platform Interface Function (PIF) for each kernel. The CPU can directly call the kernel function like any other function, with an additional parameter to setup the number of threads. Since data management and initialization is not exposed to the user, programming on the APU using CLOC becomes much easier. The same vector addition example can be simplified, as shown in Figure 2.6. Note that the same kernel function is used as in Figure 2.5. In this example, CLOC helps hide the device initialization and the kernel launch details, and it is therefore

much easier than what the OpenCL API offers.

2.2.3 Summary of CUDA and OpenCL

In this section, we briefly described the CUDA and OpenCL programming models. CUDA and OpenCL are seamlessly designed to describe GPU architectures. By explicitly controlling the resources on the GPU, programmers can achieve greater performance by manually optimizing the applications written using the low-level languages and APIs provide by CUDA or OpenCL.

Table 2.1: NVIDIA and AMD Equivalent Terminology

NVIDIA Term	AMD Term
Shared Memory	Local Memory
SMX	Compute Unit
Warp	Wavefront
Thread Block	Work Group
Thread	Work Item
threadIdx.x/y/z	get_local_id(0)/(1)/(2)
blockDim.x/y/z	get_local_size(0)/(1)/(2)
blockIdx.x/y/z	get_group_id(0)/(1)/(2)
gridDim.x/y/z	get_num_groups(0)/(1)/(2)

To aid in our comparisons when discussing programming on GPUs from NVIDIA and AMD, Table 2.1 presents the respective terminology we will use. For example, the compute unit (CU) in AMD is equivalent to SMX in NVIDIA. The warp in NVIDIA is named wavefront in AMD. The warp consists of 32 threads for NVIDIA, while a wavefront consists of 64 threads on AMD platforms. Local memory in AMD has the similar function as shared memory in NVIDIA GPUs. In this dissertation, we use "block" to refer to thread-block or work-group and use "thread" to refer to

CUDA thread or work-item, for NVIDIA and AMD, respectively.

Both CUDA and OpenCL are considered low-level programming models for GPU-style accelerators. It exposes many of the architectural features of the GPU, such as a multi-dimensional grid and thread block configuration, the scratchpad memory, disjoint memory spaces between CPU and GPU, and so on. A number of challenges can be observed in using low level programming models. First, programmers are required to thoroughly understand the underyling architecture of the accelerator. Second, these models require rewriting of significant portions of existing legacy applications that one may wish to accelerate. This is time consuming and error-prone. Third, the portability of the application becomes another challenge by writing code with CUDA/OpenCL. By portability, we mean that the source code and performance are portable across different types of GPUs. CUDA is a single-vendor solution. OpenCL can execute across a plethora of different platforms and architectures, but it is far from trivial to achieve optimum performance by sharing one piece of low level code. Meanwhile, high-level directive-based parallel programming interfaces, like OpenACC and OpenMP, address the above challenges by providing a more viable approach for addressing productivity and portability. The challenge then becomes developing compiler implementations that can do this while also delivering competitive performance.

2.3 Summary

In this chapter, we reviewed GPU architectures and their low-level programming models that are relevant to our research. GPUs are throughput-oriented devices that use hundreds of cores to execute a massive number of threads and utilize thread switching to hide memory latency. CUDA and OpenCL are two widely used, seamless programming models for mapping parallelism onto the massively parallel GPU architecture. The strength of these two programming models allow expert programmers to hand-tune their codes to exploit the full capabilities of the GPU. However, the programmer must have substantial knowledge of the hardware details of the GPU in order to achieve good performance. In the next chapter, we discuss OpenACC, an emerging directive-based programming model that aims to provide interfaces for exploiting the power of GPUs in a more productive and portable manner.

Chapter 3

OpenACC Programming Model

OpenACC is a high-level directive-based Application Programming Interface (API) that can be used to port HPC applications to different types of accelerators such as NVIDIA GPUs, AMD GPUs and other accelerators. It allows programmers to provide simple hints, known as "directives", to the compiler, identifying which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code itself. By exposing parallelism to the compiler, directives help the compiler perform the detailed transformation of mapping the computation onto the accelerator.

The OpenACC specification is driven by the feedback from many application users and has quickly evolved. The specification was first proposed during Supercomputing 2011 by Cray, PGI, CAPS and NVIDIA. Version 2.0 was officially released in June 2013. The new features in 2.0 include new controls over data movement (such as unstructured data regions), support for explicit function calls and atomic operation

in the compute region. Version 2.5 of the specification was released in October 2015, notably with an added profiling interface.

OpenMP's accelerator programming model is another directive-based approach. Since OpenMP 4.0 was released in late 2013, support for accelerator programming has become part of the specification. When we started working on the OpenACC compiler in 2012, OpenMP was not ready to support GPUs. Although the work described in this dissertation focused on OpenACC transformation and optimization, the same strategy can also be applied in an implementation of the OpenMP accelerator model targeting GPUs.

In this chapter, we first briefly describe the major features of the OpenACC specification. It is categorized into four subjects: memory model, execution model computing directives and data directives. Then, the future direction of OpenACC is discussed. The OpenACC specification uses generic terms to describe the heterogeneous programming environment. In general, the host is a CPU and the device is the accelerator (e.g., GPU).

3.1 Overview of OpenACC Programming Model

3.1.1 Memory Model

Typically, the memory of host and accelerator (device) are physically separated in the heterogeneous system as shown in Figure 3.1. In this case, the host may not be able to access the memory on the accelerator directly because it is not mapped onto

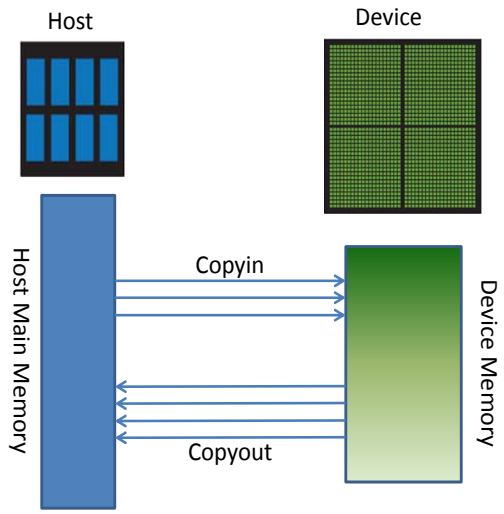


Figure 3.1: OpenACC Separate Memory Model

the host thread's virtually memory space. Explicit data movement between the host and accelerator usually is necessary. The host thread can call the accelerator driver APIs to perform the data movement between the two separated memories. Similarly, it is not valid to assume the accelerator can access the host memory directly.

The concept of separate host and accelerator memories is clear in low-level accelerator programming models like CUDA or OpenCL. Data movement between host and device memories are explicitly managed via low-level APIs by the user. In the OpenACC model, data movement between the memories is based on the directives from programmers and low-level APIs are hidden by the compiler and runtime. The OpenACC runtime maintains a present table which contains the map between host data and a corresponding copy on the device.

OpenACC uses a set of directives and clauses to manage the separate memories. In a unified memory system where the host and device share the same virtual

memory, such data directives and clauses can be ignored by the compiler. For example, compiler implementations can bypass all the data directives and clauses for the AMD HSA-based APUs in which the GPU shares physical and virtual memory with the CPU. In such scenarios, the compiler implementation does not have to create data copies for device. In more general cases, the host and device has physically or virtually separate memories. So in this case, the compiler translates high-level data clauses into the device memory management APIs and create data copies in device memory. However, data directives are necessary in order to maintain the portability of applications on these different architectures.

3.1.2 Execution Model

The execution model assumes that the main program runs on the host while the compute-intensive regions of the program are offloaded to the attached accelerator. The accelerator and the host may have separate memories, and the data movement between them may be controlled explicitly. OpenACC provides a rich set of data transfer directives, clauses and runtime calls as part of its standard. To minimize the performance degradation due to data transfer latency, OpenACC also allows asynchronous data transfer and asynchronous computation to be initiated from the host, thus enabling overlapped data movement and computation.

The execution model defined by OpenACC is host-directed execution. Compute intensive regions that are identified using directives by the programmer are offloaded to the accelerator. The device executes the offload region in parallel. Each offload

region contains nested loops which are executed in work-sharing mode.

Generally, each offload region becomes a compute kernel running on the device. Within the offload region, the host thread controls the execution flow by allocating the memory on the device, initiating the data transfer, transferring the kernel function code to the accelerator, passing arguments to the kernel function, placing the device kernel on the launching queue, waiting for completion, transferring the results back to the host and deallocating the memory on the device. If the device computation is executed asynchronously, then the host part can continue its execution until a synchronization point.

Most current accelerators support two or three levels of parallelism. Accelerators consist by multiple execution units, each of which can execute multiple threads. Each thread adopts SIMD operations. OpenACC provides these three levels of parallelism via gang, worker and vector parallelism. Gang level is coarse-grain parallelism which is fully parallel execution across execution units. At this level, synchronization support is very limited. Worker level is fine-grain parallelism which is usually implemented as multiple threads execution within each execution unit. Finally, vector level is an additional SIMD parallelism within each thread. In a typical parallelism execution scenario, a number of gangs are created on the accelerator when the kernel is launched, each gang contains one or more workers and vector parallelism is for SIMD or vector operations within each worker.

3.1.3 Compute Offload Directives

OpenACC offers two kind of accelerator compute constructs which are `parallel` and `kernels` directives. The code regions constructed by `parallel` and `kernels` directives are called offload region in this dissertation. The `parallel` directive provides prescriptive language feature that allows the user to explicitly control compiler translation. The entire `parallel` region becomes a single computational kernel running on the accelerator. In contrast, the `kernels` directive provides a descriptive capability that allows the compiler to take more control over transformation. The compiler may split the code in the `kernels` region into a sequence of accelerator kernels. Typically, each loop nest becomes a distinct kernel. When the program encounters a `kernels` construct, it launches the sequence of kernels in order on the device.

The `loop` directive in the `parallel`/`kernel` region can be used to identify the potential parallel loops and performs working-sharing parallelism. OpenACC exposes these three levels of parallelism via gang, worker and vector parallelism that can be used to partition the loop parallelism. The three levels of parallelism are described by clauses `gang`, `worker` and `vector` in the `loop` construct. How parallel execution of the different loop iterations is mapped onto the different levels of parallelism can be controlled using the gang, worker and vector clauses. These three clauses are called `loop scheduling` clauses in this dissertation.

The offload regions are launched by host with one or more gangs, and each gang includes one or more workers. Vector parallelism is for SIMD or vector operations

within a worker. Typically, for NVIDIA and AMD GPUs, `gang` maps a parallel loop iterations onto the grid-level. But worker and vector mapping may have different interpretations depending on the compiler implementation. For further performance tuning, it is possible to control the number of gangs, workers and the vector size with the `num_gangs`, `num_workers` and `vector_length` clauses within a parallel construct.

The OpenACC's model of an accelerator architecture is a collection of compute units and each of them can execute SIMD operations. This maps fairly well onto most current accelerator hardware. For NVIDIA GPUs, for example, the compute unit maps roughly onto the streaming multiprocessors, or the thread blocks, and the vector dimension maps roughly onto the threads within a thread block. There is no support for any synchronization between gangs, since current accelerators typically do not support synchronization across compute units. A program should try to map parallelism that shares data to workers within the same gang, since those workers will be executed within the same compute unit, and will share resources (such as data caches) that would make access to the shared data more efficient.

3.1.4 Data Directives

OpenACC has a structured `data` construct that allows a program to tell the compiler when certain data needs to be available on the device. One important clause for this construct is the `copy` clause, which says that if a copy of the data is allocated in device memory, that data must be initialized with the existing values from host memory, and when the device is done processing the data the final values must be copied back

to host memory. Execution of a `data` construct creates a data region, which is the dynamic range of the construct. The data specified in the `data` construct's clauses will be available on the device over the entire data region, including any procedures called within that region.

OpenACC also includes dynamic or unstructured data life-times, with the `enter data` and `exit data` directives. The `enter data` directive acts very much like the entry to a structured data construct, and the `exit data` directive acts very much like the exit from a structured data construct. The unstructured data region is especially useful within C++ constructors and destructors.

For some targets, the data directives can be completely ignored. For a multi-core and single physical memory targets, there is only ever one copy of the data and the same address is used by the host and accelerator cores, such as for AMD APUs. For an attached accelerator within its own memory, data must be allocated and copied to and from device memory since the accelerator cores cannot access host memory. For zero-copy memory (either type) or managed memory, the data needs to be allocated in the proper way. OpenACC does not control the memory allocation; the data could be static, global, local to a procedure (on the stack), or dynamically allocated.

3.1.5 Vector Addtion Example with OpenACC directives

Consider the OpenACC code example given in Figure 3.2 to demonstrate a directive-based approach to programming an acclerator. In Figure 3.2(a), the structured `data` directive is used to create the arrays a, b and c. At the beginning of the data region,

the compiler helps users allocate device memory to map these arrays. By the end of data region, the compiler helps users do two things: (1) the array c will be copied out from device memory to host memory and (2) arrays a, b and c will be deallocated from device memory. The `parallel` directive is used to tell the compiler that the entire enclosed region becomes a compute kernel running on the accelerator. The parallel region can also prescribe how the configuration of threads with the clauses `num_gangs`, `num_workers` and `vector_length`. The `loop` construct tells the compiler the loop can be parallelized and the iterations are distributed across gangs and vector lanes. There is only one compute kernel generated by compiler.

In Figure 3.2(b), unstructured data directives and a `kernels` compute construct are used. This code defines arrays to be allocated in the device memory for the remaining duration of the program, or until an `exit data` directive that deallocates the data is encountered. They also tell whether data should be copied from the host to the device memory at the `enter data` directive, and copied from the device to host memory at the `exit data` directive. The dynamic range of the program between the `enter data` directive and the matching `exit data` directive is the data lifetime for that data. The loop nests in the kernels construct are converted by the compiler into parallel kernels that run efficiently on the accelerator. Each loop nest in the `kernels` construct is compiled and launched separately. In CUDA/OpenCL terms, each loop nest becomes a separate kernel. In particular, this means that the code for the first loop nest will complete before the second loop nest begins execution.

```
#pragma acc data create(a,b,c) \
  copyout(c)\
{
 #pragma acc parallel \
  num_gangs((n+127)/128) \
  num_workers(1) \
  vector_length(128)
 {
  #pragma acc loop gang vector
  for( i=0; i<n; i++) {
   a[i] = sinf(i) * sinf(i);
   b[i] = cosf(i) * cosf(i);
  }

  #pragma acc loop gang vector
  for( i=0; i<n; i++) {
  c[i] = a[i] + b[i];
  }
 }
}

```

(a) Structured Data Region and Parallel Offload Region

```
#pragma acc enter data \
  create(a,b,c)

 #pragma acc kernels \
    present(a, b, c)
 {
  #pragma acc loop
  for( i=0; i<n; i++) {
   a[i] = sinf(i) * sinf(i);
   b[i] = cosf(i) * cosf(i);
  }

  #pragma acc loop
  for( i=0; i<n; i++) {
   c[i] = a[i] + b[i];
  }
 }

#pragma acc exit data \
  copyout(c) delete(a,b)

```

(b) Unstructured Data Directives and Kernels Offload Region

Figure 3.2: OpenACC Parallel and Kernels version of Vector Addition: assume that a, b and c are the scalar arrays in C

3.2 Present Status and Future Directions

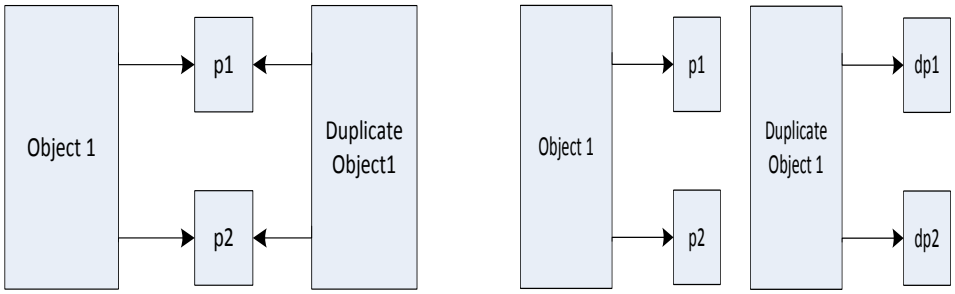
The OpenACC programming model has focused on accelerator computing and has released three version of specifications since 2011. OpenACC 1.0 basically defined the structured data region, and offload computational regions – the `parallel` and `kernels` directives. Three levels of parallelism (gang, worker, and vector) can be defined by the `loop` directive. OpenACC 2.0 introduced directives to handle unstructured data regions, offloading procedures (called from within compute regions), nested parallelism and atomic operations inside offload regions. OpenACC 2.5 added a number of changes. The most significant of these is a profiling interface. There are several other directives introduced (e.g. `init` and `shutdown` directives for device control). The behavior of some data clauses were also also changed – `copy`, `copyin`,

and `copyout` or now aliases for `pcopy`, `pcopyin`, and `pcopyout`, respectively. Essentially, this means that there is an implied presence check before these data transfer opertions. The new features of the specification are based on feedback from application users. In this section, we will review the major features that the OpenACC committee is working on for version 3.0.

3.2.1 Deep Copy

Deep copy is the significant feature that the OpenACC committee is pushing to have ready for the 3.0 release. The current OpenACC specification performs "shallow copy". Shallow copy of an aggregate data object copies all of the member field values. This works well if the fields are values, but may yield unexpected results when fields are pointers that point to host memory. The pointer is copied, but the data that it points to may not be accessible on the device. Consequentially, both original and duplicated objects point to the same memory location as shown in Figure 3.3(a). For a shallow copy, the offloaded kernel running on the accelerator tries to access the duplicated object and the pointer field inside this object points to the host memory space. In this case, a runtime error will most likely occur. Note that for unified memory architectures like AMD APU, deep copy support may not be necessary for correctness, though it may still in some cases be beneficial for performance.

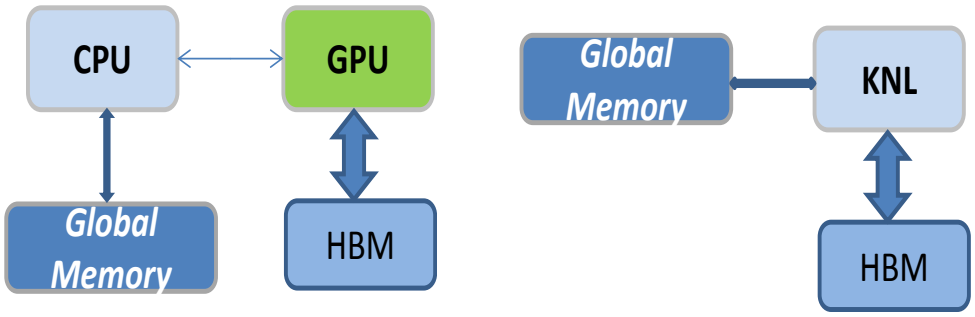
The concept of deep copy is to copy all fields in the aggregate data object which may include dynamically allocated memory pointed to by the fields as shown in Figure 3.3(b). The operation also makes copies of dynamically allocated memory



(a) Shallow Copy (b) Deep Copy

Figure 3.3: Difference between Shallow Copy and Deep Copy

that are pointed by the fields. The aggregate data type can be nested. The deep copy is to handle nested dynamic data structures. For example, there could be a large array that is used on a device and the entire array is moved to the device. But suppose the the array is actually an array of structures, each element of that array is a structure that has a field that is another allocatable array, and each one of these allocatable arrays could have differing sizes. It is extremely challenging for the compiler to decide how to transfer the data in this case. In Fortran, all of pointer and pointee information is stored in the dope vector structure which is initialized at the pointer initialization. This information includes the pointee's memory location and length. For Fortran applications, the compiler has the information needed to perform a full deep copy. But for C/C++ codes, the language does not provide a means to capture the memory size of pointees at compile-time. Therefore, it is necessary to define directives that give the compiler hints on how to generate the transfers and avoid runtime errors. Moreover, if the data is only used partially, it is not necessary to fully move the entire nested dynamic structure, which will increase the data traffic



(a) Heterogeneous Node Architecture (b) Intel Knights-Landing Architecture
Figure 3.4: Modern HPC Node Memory Hierarchy

between host and device. As a consequence, a directive-based approach can help users shape the aggregate data structures and move the data efficiently.

3.2.2 Memory Hierarchy Management

OpenACC is getting ready for the exposed memory hierarchies associated with upcoming systems. We consider a modern HPC heterogeneous node in Figure 3.4. Figure 3.4(a) depicts the memory hierarchy model for next generation supercomputer at Oak Ridge National Lab, known as Summit [13]. Each Summit node consists of multiple IBM POWER9 CPUs and NVIDIA Volta GPUs all connected together with NVIDIAs high-speed NVLink and a vast amount of memory. Each node will have coherent memory, high bandwidth memory (HBM) and DDR4, addressable by all CPUs and GPUs. Apparently, HBM is much closer to the GPU and has a higher bandwidth to feed the massively parallel units in GPUs. The frequently used data is likely to be placed in the HBM in order to achieve peak performance. In contrast,

another system is shown in Figure 3.4(b), a self-hosted accelerator node with the Intel "Knight Landing" architecture [6]. HBM is packaged on board memory with five times the bandwidth compared to DDR4 global memory. In both systems, the DDR4 and HBM are true unified address spaces, though they have separate physical memories. Users have to explicitly manage the memory resources via the lower-level interfaces, which exposes a large amount of hardware detail and can therefore hinder application portability. In particular, the GPU has its own deep memory hierarchy on board which increase the complexity of the entire memory system inside each compute node. In summary, it is necessary to propose high-level abstractions to describe and manage the data movement across the memory hierarchy. It respects performance but is also as natural and portable as possible across all the different various systems.

3.2.3 Other Features

Another feature under discussion is improving support for multiple devices. While OpenACC already supports multiple devices, it is not as convenient as users may expect. We have had success with utilizing multiple MPI ranks and multiple OpenMP threads, and having each process, or each thread attach to a different devices [66, 67]. But there may be better ways to make it work with the new directives and routines under discussion which allow a single thread to make use of multiple devices.

However, the challenge there is more about managing the data than the computation. It is relatively easy to spread the computation across resources; the challenge

is to make sure the data is in the right place in order to obtain the desired performance. So the upcoming releases will more likely focus on descriptive interfaces for improving data locality.

3.3 Summary

In this chapter, we presented an overview of the OpenACC programming model which is used in our research. OpenACC provides a rich set of directives and APIs that allow users to annotate data regions, explicitly control data transfers and expose loop parallelism. These features can enable non-expert programmers to portably and productively achieve performance for their applications. OpenACC directives hide many details of the underlying implementation, freeing a programmer's attention for other tasks. Beyond these core features defined in OpenACC specification, the language continues to evolve, with features for describing complex memory hierarchies, optimizing data locality, and improved multiple devices support all under consideration. In the next chapter, we discuss the existing implementations of OpenACC and other directive-based approaches.

Chapter 4

Related Work

Languages such as OpenCL and CUDA offer a standard interface for general-purpose programming of GPUs. However, with these languages programmers must explicitly manage numerous low-level details involving communication and synchronization. This burden makes programming GPUs difficult and error-prone, rendering these devices inaccessible to most programmers. Therefore, there are both commercial and academic efforts to support high-level programming models that can also effectively exploit GPUs full potential.

There are some approaches [51, 52] that use high level languages for GPU programming. Microsoft C++ AMP [1] (Accelerated Massive Parallelism) is sophisticated C++ specified solution using C++ templates and lambda functions for data and compute. Lime [26], a Java-compatible language, was proposed for heterogeneous systems. The backend of the Lime compiler generates OpenCL code for accelerator devices. Other high level languages, like Python, have similar solutions for

GPU computing. However, these new languages requires programmers to rewrite their applications, which is not a practical approach for accelerating large amounts of existing legacy code. Since our work focuses on the directive-based approach, we concentrate the related work on the directive-based approaches.

As the existing infrastructure of OpenUH was the basis for our experimental platform, we also include an introduction of the OpenUH compiler framework in Section 4.3. OpenUH is a modern compiler infrastructure which the HPCTools group has used to support programming models research for over ten years. The author has added and maintained support for the OpenACC programming model targeting NVIDIA and AMD GPUs since version 3.1.0 of the compiler. The source code can be downloaded from [12].

4.1 OpenACC Implementations

The accULL [54] is a prototype OpenACC 1.0 implementation based on the Python library. accULL was the first released implementation of OpenACC. However, their syntax does not follow the OpenACC standard in some key respects. For example, the data region representation only includes length, and it does not support imperfect nested loop transformations. We could therefore say that accULL is an implementation of a simplified OpenACC dialect.

The Omni OpenACC compiler [58] behaves as a source-to-source translator and generates CUDA C which is then compiled by NVIDIA's CUDA compiler. Their loop scheduling implementation follows the OpenACC specification. A limitation

in their solution is that the application cannot make use of multi-dimensional grid topologies for NVIDIA GPUs. The generated CUDA C exhibits significant overhead due to inefficient loop scheduling transformations.

Rose-OpenACC [62] implements an OpenACC dialect based on the ROSE compiler framework and is primarily used within the research community. The implementation uses OpenCL as its target language. Only the `gang` and `worker` clauses are interpreted and mapped to work-group and work-item in OpenCL, respectively. Each work-item (or "thread" in CUDA) takes a chunk of consecutive iterations, where the size of each chunk is the tiling size determined by an OpenCL runtime. However, while this strategy may work efficiently for traditional CPU-based accelerators such as Xeon Phi, it is definitely not a GPU-friendly solution since it does not take into account coalescing of memory accesses. Moreover, the OpenCL kernel function includes frequent OpenCL device function calls which contribute significant overhead. If thread divergence occurs within the device function, as observed for Omni OpenACC, the performance will degrade considerably when executing on the GPU.

OpenARC [45, 46, 44] is based on the Cetus source-to-source framework. OpenARC is a framework for compiling, debugging, and profiling OpenACC applications. They proposed self-defined directives to allow users to address data locality issues. For example, arrays can be mapped to different CUDA memory spaces, including the shared memory and texture memory. Exposing the lower-level device features makes the application non-compatible on other non-GPUs platforms.

IPMACC [42] is an open source framework for source-to-source translation of

OpenACC for C applications and supports execution over the CUDA and OpenCL runtime. New directives [41] are proposed to exploit the scratched memory in order to close the gap between the OpenACC and OpenCL/CUDA applications.

XcalableACC [50] (XACC) is a hybrid programming model that combines a directive-based PGAS model called XMP [49] and OpenACC for targeting accelerator-based clusters. The framework is based on the Omni compiler mentioned earlier. An innovative method employed by this compiler allows data to be directly transferred among accelerators. Stencil applications are used to evaluate this method.

GCC, as of version 5.1, provides an experimental implementation of OpenACC 2.0 [4]. However, the released GCC 5 only includes a limited preliminary implementation. For example, with the `parallel` construct, the execution model only allows for one gang, one worker and a number of vectors. All the vectors execute the redundant mode. The `kernels` construct is supported only in a naive way. Reductions are not supported inside kernels constructs. GCC 6 is expected to include an improved implementation for OpenACC.

The NVIDIA PGI compiler has pioneered directive-based accelerator computing. OpenACC was initially developed by PGI, Cray and NVIDIA, and it is largely based on the PGI Accelerator programming model [64]. The Cray and PathScale compilers also support OpenACC. Unlike most non-commercial efforts from academia and the open-source community, commercial compilers generally adopt a source-to-binary compilation path. However, since these commercial implementations are closed-source and their inner-workings are mostly inaccessible to researchers, they cannot be used to gain an understanding of OpenACC compiler technology or to explore

possible improvements to it.

4.2 Other Directive-based Compiler Implementations

hiCUDA [31] is one of the earliest programming models using a directive-based approach to ease the programmability for CUDA devices. In hiCUDA, users can explicitly control the hardware resources, such as shared memory, grid and thread-block topology. From the user's perspective, hiCUDA is still a low-level programming model and helps programmers translate an annotated region into CUDA. Therefore, in an optimized hiCUDA application the inserted directives may exceed the original code size due to their explicit CUDA mapping strategies. Unlike the OpenMP and OpenACC model, the directives are not structured, and the programmer is responsible for managing everything, including all the data movement, loop scheduling, and even on-chip resources.

The PGI Accelerator Programming Model [64] is a directive-based accelerator programming model for CPU+Accelerator systems and served as an early prototype model for OpenACC. 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 transferred back from the accelerator to the host at the exit

of compute region. Data directives allow the programmer to manually control data movement, i.e. where to transfer the data other than at the boundaries of compute regions. Loop directives allow 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 through a component called the *Planner*. PGI's compiler optimizes the data transfer by "data region" directive and its clauses in order to remove unnecessary data copies. Using the loop scheduling directive, the user can add the data in the highest level of the data cache with the "cache" clause and thereby reduce data access latencies.

OpenMPC [43] is a compiler framework for translating an OpenMP program to a CUDA program. The main contributions of this work include an interpretation of OpenMP semantics under the CUDA model and a set of transformations that optimize global memory accesses. However, there are several drawbacks in OpenMPC. First, there is only one level of parallelism and a lack of fine-grained parallelism. The OpenMP parallel loop region is mapped to threads in the grid. For nested loops, if the outer loop is marked with an OpenMP loop directive, the inner loop is executed sequentially by each thread even if the loops are parallelizable. The OpenMPC framework can only generate one dimensional thread-blocks and one dimensional grids. This limitation prevents the implementation from exploiting massively parallel thread topologies on the GPU. Second, the data locality optimization requires users to explicitly control with OpenMPC customized data clauses, and the compiler lacks implicit analysis and optimization to relieve developers of this programming

burden. Third, the kernels splitting in the OpenMPC generates many small kernels and potentially increases kernel launch overhead.

Hybrid Multicore Parallel Programming (HMPP) [14], proposed by CAPS Enterprise, also provides a set of directives to improve the performance by enhancing code generation. In HMPP, the two most important concepts are *codelet* and *callsite*. The codelet represents the function that is offloaded to the accelerator, and the callsite is the place at which the codelet is invoked. It is the programmers responsibility to annotate the code by identifying codelets and inform the compiler about the codelets and where to call them. Within the codelet, the user can put read-only data into constant memory, preload frequently used data into shared memory, or explicitly specify the grid size for NVIDIA architectures. If the loop is so complex that the compiler is not able to analyze it, the user can give some hints to the compiler that all iterations in the loop are independent. The HMPP compiler does not perform much implicit analysis and optimization to improve loop transformation and data locality. As a result, HMPP is an explicit and prescriptive directive-based solution that emphasizes the need for expert programmers to generate optimized code.

The OmpSs [28] programming model proposed by the Barcelona Supercomputing Center is another directive-based approach for aiding application porting to heterogeneous architectures. However, OmpSs does generate accelerator code for the user. Instead, it requires programmers to write the accelerator kernel functions manually. This solution exposes too many low-level hardware details and limits the portability of applications. Additionally, the performance of computational kernels running on the accelerators relies on the user's programming skill. Analysis and optimization

in the OmpSs compiler is also limited. Expert accelerator knowledge is required to fully exploit the compute capabilities of heterogeneous architectures. Applications written with OmpSs are practically "baked" to specific machines, execution models and architectures. For the compiler, it is not possible to safely remap or optimize such applications to new machines or architectures.

[48] presents a prototype of OpenMP using the ROSE compiler called Heterogeneous OpenMP (HOMP) and shares experiences with the ROSE OpenMP accelerator model. Three cases which includes Jacobi, AXPY and Matrix Multiplication were given to evaluate their initial implementation. Three choices were given to schedule the loop iterations among GPU threads using loop `distribute` construct: 1) use only the master threads of multiple thread blocks when `distribute` is used right before the loop; 2) use threads from a single thread block; 3) use a combination of multiple blocks and multiple threads per block when applicable. However, the explanation of loop scheduling in the paper is still vague. Data locality optimizations were not mentioned since HOMP is only an initial implementation. Since HOMP only concentrates on CUDA, their implementation is limited to users of NVIDIA GPUs.

The OpenMP 4.5 implementation on GCC and LLVM is another important implementaiton of directive-based accelerator support [10, 7, 11]. GCC 5 supports two offloading configurations which include OpenMP to Intel Xeon Phi coprocessors and NVIDIA GPU targets. The LLVM community is also working on their OpenMP 4.5 implementation. The project contributors include IBM, Intel, TI, AMD, DOE labs, among other members.

4.3 Existing OpenUH Infrastructure

The OpenUH compiler [47] is a branch of the open source Open64 compiler suite for C, C++, Fortran 95/2003. It is an industrial-strength optimizing compiler that integrates all components needed of a modern compiler. OpenUH serves as parallel programming infrastructure in the compiler research community and serves as the basis for a broad range of research endeavors, such as language research [29, 33, 27], static analysis of parallel programs [24, 34], performance analysis [53], task scheduling and dynamic optimization [18].

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

OpenUH did not provide any support on directive-based accelerator programming prior to our implementation of OpenACC, though OpenMP 2.5 directive support was already present. The support for OpenMP provided existing procedures for parsing

compiler directives and representing them within the compiler's intermediate representation (IR). To develop a baseline OpenACC implementation, we enhanced the front-ends to recognize OpenACC syntax, extended the IR to represent OpenACC operations within the compiler's abstract syntax tree, and implemented code generation for accelerators. We will discuss the design and implementation details in the next chapter.

4.4 Summary

In this chapter, we described the related compiler efforts regarding directive-based approaches for GPU computing. The common problem faced by existing open-source compiler implementations for directive-based programming of accelerators is a lack of focus on compiler optimization. For instance, OpenARC, accULL and OmniACC focus on compiler framework construction and IR extension to represent the language features. Rose OpenACC does entail some loop transformations, however, data locality analysis and optimization are not mentioned in their framework. None of them explicitly explained the details of loop scheduling transformation and data locality optimization for their respective implementations. Commercial compilers are well designed, but their source codes are mostly inaccessible to researchers and hence they cannot be used as further research platform. Some compiler implementations, such as OpenMPC and hiCUDA, are specific to CUDA devices and consequentially

have lost their generality. hiCUDA and HMPP requires low-level directives to describe the hardware feature and resources, so portability is sacrificed across different vendors' architectures. The existing open source research compilers are purely source-to-source translators that have little analysis and optimization capability in their underlying compiler framework. Further, none of them has Fortran support. Therefore, there remains a need for robust, open source and optimizing OpenACC compiler for academia research and teaching.

We also introduced our existing OpenUH compiler infrastructure which is the basis for the OpenACC compiler implementation presented in this dissertation. Our goal is to create general loop scheduling transformations for GPGPUs and APUs, and further to propose sophisticated compiler analysis and transformations to improve the data locality across different architectures. In the next chapter, we present our OpenACC design within the OpenUH.

Chapter 5

OpenACC Support in OpenUH Compiler

In this chapter, we present a high-level overview of compiler design and runtime implementation of OpenACC within the OpenUH compiler, a branch of the Open64 compiler framework. The chapter has been organized as follows. First, the enhanced components in OpenUH are described in Section 5.1. These components correspond to parts of the compiler infrastructure that haven been modified to support the OpenACC model. This is followed by a discussion of a proposed algorithm using liveness analysis to assist the generation of function kernels that target GPU platforms. This is presented in Section 5.2.

5.1 OpenACC Components

The creation of an OpenACC compiler requires innovative research solutions to meet the challenges of mapping high-level loop iterations to low-level threading architectures of the hardware. It also leverage runtime support for handling data movement and scheduling of computation on the accelerators.

The components of the OpenACC implementation framework is shown in Figure 5.1. The compiler is comprised of several modules, each module operating at one of the multiple Intermediate Representations (IR) of the Open64 framework (called WHIRL). From the figure, each compiler module performs a set of analyses using its corresponding IR. Before transitioning to a module at a lower-level, the IR is simplified to the WHIRL at a lower-level.

The compiler framework is used to faciliate source-to-source translation of a program with high-level OpenACC offload regions into high-level CUDA/OpenCL kernel functions. The CUDA version of the program is lowered using the NVIDIA CUDA compiler (NVCC) and target NVIDIA GPUs. The OpenCL version, on the other hand, can be targetted to either the AMD APUs or the AMD discrete GPUs. For the former, the OpenCL version is passed through CLOC [2], AMD's Offline CL Compiler, which in turn translates the OpenCL kernel functions into HSA object files. To target AMD's discrete GPU, the OpenCL kernel functions are directed to OpenCL driver APIs. The parts of the OpenACC program that are intended to target the host CPU are translated to x86 binaries by directing them to the code-lowering modules (backend, code generation, etc.) of the OpenUH compiler.

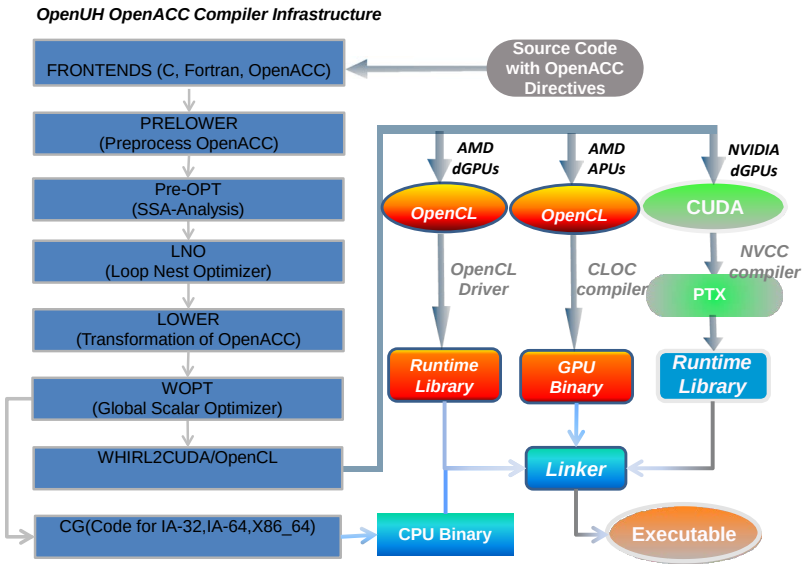


Figure 5.1: OpenUH compiler framework for OpenACC: for NVIDIA GPUs, NVCC is the NVIDIA CUDA Compiler; for AMD APUs, CLOC which is the AMD CL Offline Compiler compiles the OpenCL-like kernel functions to GPU object file; for AMD discrete GPUs, AMD OpenCL SDK is used to compile the OpenCL kernel functions dynamically during the runtime.

We have identified the following challenges that must be addressed to create an effective implementation of OpenACC directives. As part of the design, it was crucial to create an extensible parser and IR system that facilitate inclusion of new and modified features of the OpenACC standard. This would enable future development of features that can take advantage of the aggressive compiler optimizations. The extension of the IR was necessary to distinguish the code regions that target CPU from those that target the GPU. This was aided by the flexibility of the OpenUH and the WHIRL IR framework that allowed for the inclusion of the said extensions without too much difficulty. Second, an algorithm based on liveness analysis was designed to reduce unwanted device memory movement and assist the data locality optimization within the offload regions. The details for these are discussed in section 5.2. Third, we need to design and implement an effective strategy to distribute the loop nest across the GPU threads hierarchy. This is referred to as loop scheduling. Dependence analyses was necessary to reduce synchronization among loops. The performances of generated GPU kernel function largely depends on the loop mapping mechanisms. Solutions to these are discussed in more detail in Chapter 6.

The OpenACC support that has been implemented in OpenUH comprises four components:

- an extended front-end that accepts the OpenACC directive syntax in C and Fortrans
- a back-end analysis, optimization and translation that target OpenACC offload regions

- enhanced IR-to-source tools that support the translation of CUDA/OpenCL kernel functions
- a portable runtime system for support AMD/NVIDIA GPU architectures.

The following sections discuss each of topic in detail.

5.1.1 Front-end

OpenUH front-end inherits from the original Open64 compiler. Open64 uses GNU GCC 4.2.0 to parse C/C++ and Cray Fortran 95 to compile the Fortran applications. These two front-ends generate high level GNU and CRAY IR which are then translated to very high level WHIRL. We modified the GNU C and Cray Fortran 95 front-end to accept the OpenACC directives and generate respective WHIRL IR nodes that represent OpenACC related constructs. The programming languages being targeted was C and Fortran[1]. In order to take advantage of the analysis and optimizing capabilities in the OpenUH back-end, the entire OpenACC IR representation is preserved through the backend. The intrinsic functions that appear in the OpenACC offload region need to be identified and mapped to the corresponding CUDA/OpenCL intrinsic functions. For this, the front-end of OpenUH was extended to support the entire set of intrinsics defined in the CUDA/OpenCL specification.

[1]The OpenACC support for C++ front-end remains unimplemented

5.1.2 Back-end

The main research contribution of this work relates to the back-end of the compiler. We introduced a new phase called 'OpenACC lower' which translates the OpenACC IR into runtime function calls and CUDA/OpenCL IR modules. On addition of this work, the back-end of the compiler can be divided into the three parts - the 'Pre-Optimizer' (Pre-OPT), the 'Loop Nest Optimizer' (LNO) and the 'OpenACC IR lower'. Both Pre-OPT and LNO perform a set of analyses, the results of which are reflected in IR that targets the offload region. The 'OpenACC IR lower' phase takes advantage of these analyses results and perform additional set of optimizations and transformations, as needed. The proposed new algorithm that extends the liveness analysis is built into this Pre-OPT phase (Section 5.2).

The LNO phase can potentially do many analyses and optimizations including dependence analysis, auto-parallelization, cost-model-based loop scheduling optimization, and scalar replacement. In OpenACC *kernels* directive region, compiler relies on the dependence analysis results to determine whether a given loop can be parallelized or not. The scalar-replacement is based on the dependence analysis to extract the data reuse information across the loop iterations in the offload region. The detail of the improved scalar replacement algorithm and implementation is discussed in Chapter 7.

OpenACC IR lower phase is the most important module in the entire OpenACC compiler. It does the complex transformation work. First, offload regions are extracted as outline CUDA/OpenCL functions. Symbol table extensions are made to

identify CUDA/OpenCL languages' features, such as Shared Memory in CUDA and Local Memory in OpenCL. CPU and GPU IR are separated at this time. Compiler continues compiling the CPU IR and generates binary object for CPU. GPU IR be forwarded to another phase called WHIRL2CUDA/WHIRL2OpenCL to generate GPU kernel functions. Second, the original offload region IR is replaced by IR which represent runtime function calls. The runtime functions are used to move the data between CPU and GPU, and launch the GPU kernels. Third, loop scheduling transformation is applied to OpenACC loops inside the offload region. So the original sequential loops identified as OpenACC loop directives are executed in parallel on the GPU. Fourth, scalar replacement transformation is finalized based on the information generated at LNO phase. Fifth, all the array references are transformed into pointer and offset dereference operations.

5.1.3 CUDA/OpenCL Generation

Our implementation compiles the CPU code-region to x86 binary and uses the source-to-source translation approach for generating the binary that targets the accelerator. After the OpenACC IR lower transformation, the IR module for GPU code is translated into OpenCL or CUDA kernel function which depends on the compilation target. As the primary goal of OpenUH is for compiler research, it also supports purely source-to-source compilation path for OpenACC applications. The source-to-source approach can clearly show how the OpenACC compiler translation performs for both CPU and GPU. This makes OpenUH a portable compiler that targets multiple target platforms.

Compared to binary code generation, the source-to-source approach for the GPU provides greater flexibility to users. It allows the programmer to leverage advanced optimization features in provided by the NVIDIA CUDA compiler and AMD OpenCL compiler. It also provides the user with an ability to manually optimize the generated CUDA code region before it is forwarded to the NVIDIA compiler.

The WHIRL2CUDA and WHIRL2OpenCL components are based on the original WHIRL2C which translates the WHIRL IR into the C languages. If the host side source-to-source translation is not invoked, the WHIRL2CUDA and WHIRL2OpenCL checks the IR module flag and see if they are accelerator kernel functions. Then such accelerator module IR will be translated into CUDA/OpenCL kernel functions.

Consider the OpenACC code targeting NVIDIA GPU in Figure 5.2 (a) as an example. Figure 5.2 (b) and (c) show the translated CUDA kernel and the equivalent host CPU pseudo code. We have created a WHIRL2CUDA/OpenCL tool that can produce NVIDIA CUDA/AMD OpenCL kernels after the transformation of offloading code regions. Compared to binary code generation, the source-to-source approach provides much more flexibility to users. It allows users to leverage the advanced optimization features in the back-end compilation step performed by the CUDA/OpenCL compiler *nvcc*. It also gives users some options to manually optimize the generated CUDA/OpenCL code for further performance improvement.

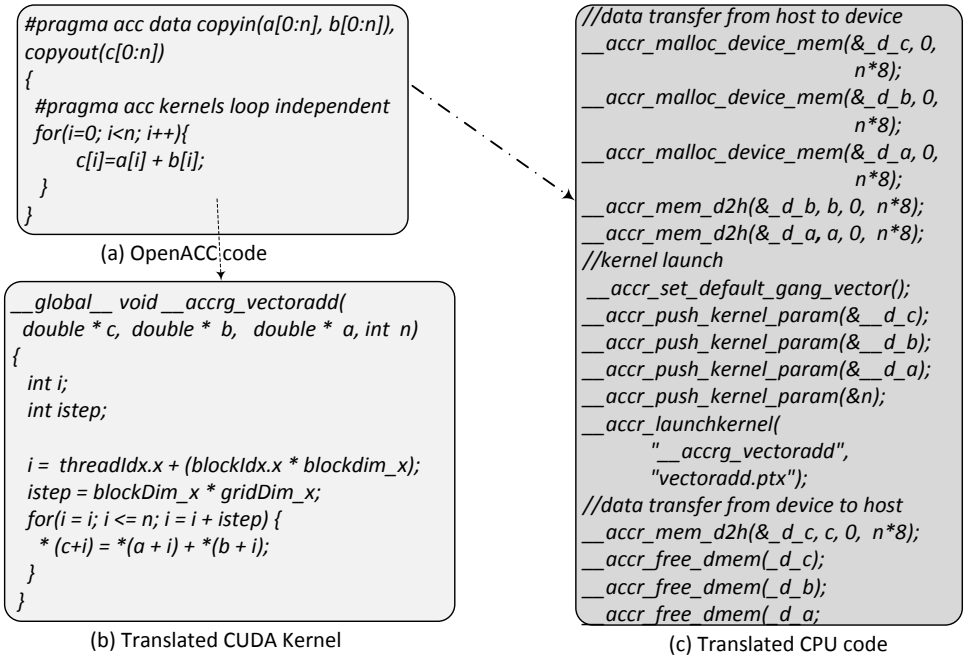


Figure 5.2: OpenACC vector addition example: OpenUH compiler outlines the offload region as a GPU kernel function in (b); the CPU code is basically replace with OpenACC runtime function calls after the OpenACC IR transformation.

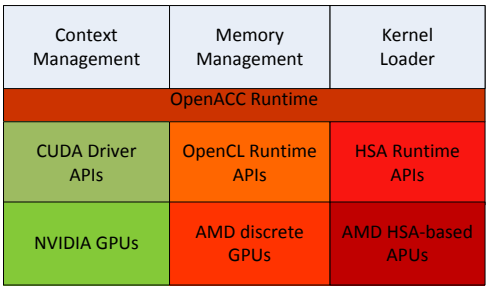
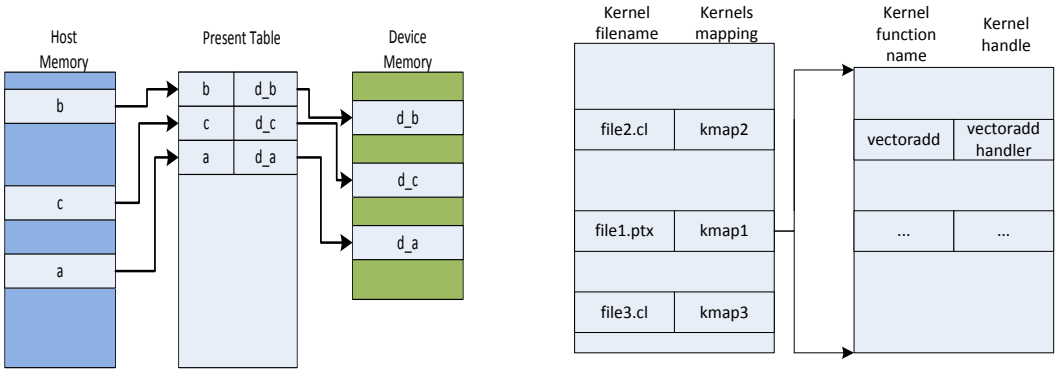


Figure 5.3: OpenACC runtime library frameowork

5.1.4 Runtime Support

A portable runtime is created to support multiple devices, data movement management, and GPU kernel launching. The runtime library in Figure 5.3 consists of three modules: context module, memory manager, and kernel loader. The context module is in charge of initializing, creating and managing the virtual execution environment. The runtime implementation is based on top of vendor's Software Development Kit (SDK) (CUDA Driver, AMD OpenCL SDK and AMD HSA SDK) for different vendor's GPU architecture support.

Data manager module handles the data allocation and traffic between host and device. The major overhead in this module is to do data mapping lookup operations as application developers can only use the host address. Every kernel launch and data update requires device data addresses. Inefficient data mapping mechanism will increase the application runtime overhead. In our implementation, we choose the hash table to map the host data and device data. Such table is called Data Present Table (DPT). Each hash table lookup usually cost O(1). If the data is already in the DPT, runtime uses the host data address as hash key to find the



(a) Hashmap based Data Present Table (b) Hashmap based Kernel Present Table

Figure 5.4: Optimized Runtime Data and Kernel Management to Reduce Runtime Overhead

respective data entry on device memory. If such entry does not exist, runtime calls the device *malloc* function to create a copy for the host data and put it in the DPT. In the large application, the data is frequently queried at the runtime and hashtable can minimize the lookup cost. Figure 5.4(a) shows an example of the DPT. The DPT uses the host data as hash-key. The value in the DPT entry is the device data address. For unified memory architecture, like AMD APUs, this module ignores all the data allocation and movement between host and device.

The kernel model controls the kernel functions' online compilation, GPU thread configuration and kernel functions launch. CUDA kernel functions for NVIDIA GPUs and OpenCL kernel functions for AMD APUs are compiled into virtual ISA code offline according to the Figure 5.1. Runtime library still needs to finalize the virtual ISA code into the real hardware ISA code. OpenCL kernel functions for AMD discrete GPUs are compiled into binary during the runtime. In this module, the

online compilation may cause huge overhead if it cannot be handled properly. We adopt two level of hashmap to map the kernel functions and real executable kernel handler which is claimed by low-level vendor's API. Figure 5.4(b) demonstrates how to use the hashmap to store the two levels of kernel mapping method. It first uses kernel source code file names as hashkey to map another hashmap which takes kernel function names as hashkey and maps to the executable kernel handler. This method makes sure that every kernel file is loaded and compiled at most once when the first kernel in this file is invoked. The threads configuration and kernel launch are implemented using the vendor's APIs.

5.2 Liveness Analysis for Offload Regions

5.2.1 Classic Liveness Analysis

Live variable analysis is a widely used iterative data flow analysis to compute live variables for each basic block [40]. A Basic Block (BB) is a straight-line code sequence with no branches 'in' except to the entry and no branches 'out' except at the exit. The variables are live at the exit from each BB because they may be potentially used before their next write. In other words, a variable is live if it holds a value that may be needed in the future. So to compute liveness at a given point, we need to look into the future. Liveness flows backwards through the Control Flow Graph (CFG), because the behavior at future nodes determines liveness at a given node. The liveness analysis analyzes a CFG to determine which places variables are live or

not.

Liveness analysis is used to help in register allocation at the compiler's code generation phase. Before the register allocation, IR contains an unbounded number of variables. But the actual machine has a limited number of registers. Variables with disjoint live ranges can map to same register. In other words, a variable is not live at the exit of BB, then the register assigned to this variable can be reassigned to another variable. Liveness Analysis can determine when variables/registers hold values that may still be needed in the future.

Before performing liveness analysis, we need to understand the control flow by building a Control Flow Graph (CFG). In a CFG, each node is BB and the edge between two nodes is the potential flow of control (execution flow). Out-edges from node n lead to successor nodes which are defined as set $SUCC[n]$. In-edges to node n come from predecessor nodes which are defined as set $PRED[n]$. Gathering liveness information is a form of data flow analysis operating over the CFG. Liveness of variables "flows around the edges of the graph. The $GEN[n]$ is defined as the variables set in which each variable is read operation in BB n and the $KILL[n]$ is defined as the variables set in which each variable is modified in BB n. $GEN[n]$ contains the variables whose liveness is generated with BB n. These variables have upwards exposed uses in BB n. While $KILL[n]$ contains the variables whose liveness is killed in BB n. Typically, they are the variables appearing on the left hand side of an assignment statement in BB n. The $IN[n]$ and $OUT[n]$ are the alive variable set at the entry and exit of BB n. Based on the $GEN[n]$ and $KILL[n]$ information, $IN[n]$ and $OUT[n]$ can be calculated by Equation 5.1 and 5.2 [40].

$$IN[n] = (OUT[n] - KILL[n]) \cap GEN[n] \tag{5.1}$$

$$OUT[n] = \begin{cases} \phi & \text{if } n \text{ is End Block} \\ \bigcup\limits_{s \in SUCC[n]} IN[s] & \text{Otherwise} \end{cases} \tag{5.2}$$

5.2.2 Extended Liveness Analysis for Offload Regions

It is necessary to use liveness analysis to determine the variables properties for each offload region. If the compiler implementation directly follows the OpenACC specification, the scalar variables in the kernels directive are copied by default. If the variables is copied, compiler generates a series of device memory allocation functions to map the host scalar variables into the device memory space. This leads to memory fragmentation within the GPU memory and interconnect traffic between the host CPU and the GPU. However, if it is not necessary for the scalar variables to be created, the scalar variable are 'read-only' and 'private' to the offload region. In either case, they do not need to be allocated into the device memory. Instead the references to the 'Read-only' scalar variables can be passed as parameters to the kernel functions. 'Private variables' can be privatized and placed into register file within each thread.

The classical liveness analysis [40] is extended to fit into the OpenACC programming mode. Two extensions are made in order to apply in the OpenACC offload

region analysis. First, the entire offload region is treated as Basic Block (BB). The traditional BB is a straight-line code sequence without branches in except to the entry and no branches except the exit. Since the offload region is translated into an outlined accelerator kernel function with only one entry and one exit. Therefore, each kernel function can be considered as BB during the analysis. Next, the variables in the IN and OUT set must be used in the offload region. In the classical liveness analysis, if the offload region dominates other BBs, the variables used in these BBs - even if not used in the offload regions, will be included in the IN/OUT set. As a solution, we propose using an additional $USED$ set to distinguish variables that have been 'used' in the offload region from those that are not. A logical 'AND' operation is applied between IN/OUT and $USED$ sets to filter the unused variables from IN/OUT set (Equation 5.3 and 5.4).

$$IN' = IN \cap USED \tag{5.3}$$

$$OUT' = OUT \cap USED \tag{5.4}$$

$$PARAMETERS = IN' \cup OUT' \tag{5.5}$$

$$PRIVATE = USED \cap (IN' \cup OUT') \tag{5.6}$$

$$INOUT = (IN' \cap OUT') \cap (\neg POINTER) \tag{5.7}$$

$$FIRST\_PRIVATE = (IN' \cap \neg POINTER) \cap \neg INOUT \tag{5.8}$$

$$LAST\_PRIVATE = (OUT' \cap \neg POINTER) \cap \neg INOUT \tag{5.9}$$

The PARAMETERS set which can be computed with 5.5 are the variables appearing as arguments in the kernel functions. Since all the array and pointer variables have to be passed as pointer, the liveness analysis only checks the scalar variables and determine if it is necessary to create pointers to represent the scalar variables. PRIVATE set 5.6 is the thread-private scalar variables. INOUT set 5.7 are the scalar variables requiring data transferring back to host. FIRST_PRIVATE set 5.8 are the copyin scalar variables. LAST_PRIVATE set 5.9 includes the scalar variables copying back to host. When kernel function outline is created, every scalar belonging either INOUT or LAST_PRIVATE set is mapped to a pointer. Before the kernel launching, runtime performs device memory allocation and copies the data from host to device memory. After the offload computing finished, runtime moves the data from device back to host. Each variable in FIRST_PRIVATE, PRIVATE, LAST_PRIVATE and INOUT is privatized in each thread and placed into the register files.

Two more variable sets referred to as READONLY_PTR and READONLY_SCALAR have been introduced to each offload region. READONLY_PTR set contains the read-only buffer pointers and arrays. READONLY_SCALAR set contains read-only scalar variables in the offload region. This extended liveness analysis targets the high-level IR and is built within the PRE-OPT phase(Figure 5.1). While the array IR operations are lowered into pointer operations during the OpenACC IR transformation. Read-only arrays together with pointers are included the READONLY_PTR

set. Both READONLY_PTR and READONLY_SCALAR can be generated from POINTER and FIRST_PRIVATE set by removing the modified variables. The read only array and pointer can be cached in the GPU special Texture Cache Unit and read-only scalar variables can be assigned to Constant Memory and read-only scalar register files. These two data sets can help compiler generated data locality optimized GPU kernels.

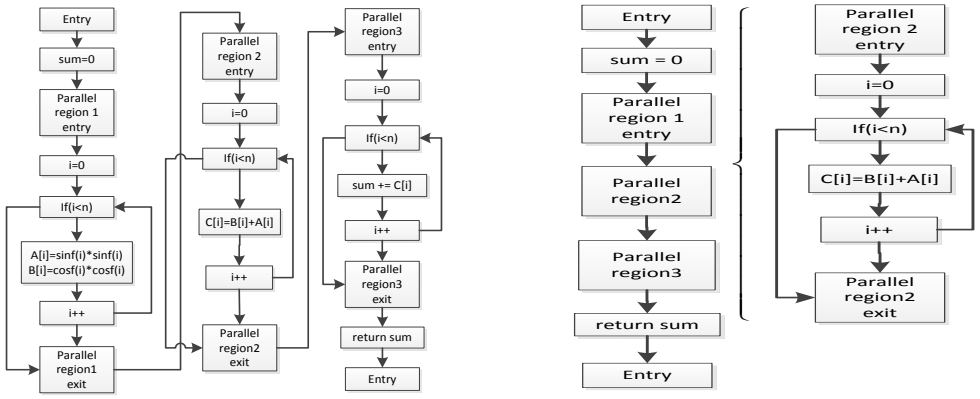
Let us consider a simple vector addition example illustrated in the Figure 5.5. There are three offload regions in this function. If we use the traditional Basic Block definition to build the CFG, it becomes complex. The Figure 5.6(a) shows the respective CFG. Considering each offload region as a Basic Block, the CFG can be simplified as Figure 5.6(b). The simplified version of the CFG is much easier for analysis and debug. Figure 5.7(a) shows the result using the classic liveness analysis. The parallel region1 does not use the variable *sum* at all but it appears in the IN and OUT set. Because parallel region1 dominates path from *entry* to the beyond basic blocks. We propose the USED set for each offload region. It can be easily generated by scanning IR of the each offload region. By applying the extended liveness analysis, results can be computed (Figure 5.7(b)).

5.3 Summary

This chapter described the compiler infrastructure that we used to build our implementation of OpenACC as well as the extended liveness analysis for high-level IR

```
1  float* verify_vectoradd(float* a, float* b, float* c, int n)
2  {
3    int i;
4    float sum;
5    sum = 0;
6    #pragma acc data create(a[0:n], b[0:n], c[0:n])
7    {
8      #pragma acc parallel loop
9      for(i=0;i<n;i++)  {
10       a[i] = sinf(i) * sinf(i);
11       b[i] = cosf(i) * cosf(i);
12     }
13     #pragma acc parallel loop
14     for(i=0;i<n;i++)  {
15       c[i] = a[i] + b[i];
16     }
17     #pragma acc parallel loop reduction(+:sum)
18     for(i=0;i<n;i++)  {
19       sum += c[i];
20     }
21   }
22   return sum;
23 }
```

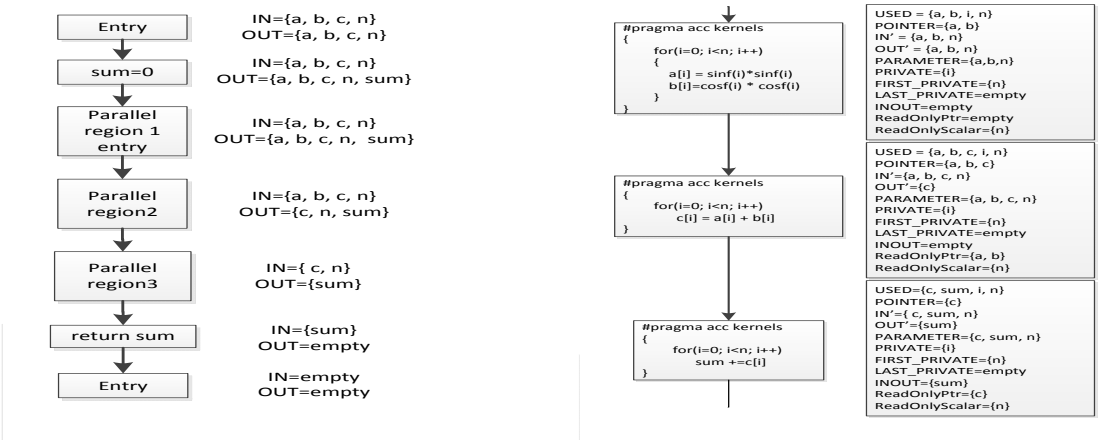
Figure 5.5: Liveness Analysis Example



(a) Classic Control Flow Graph vs Simplified CFG for Offload Region

(b) Control Flow Graph with Simplified Offload Region as Basic Block

Figure 5.6: Classic Control Flow Graph vs Simplified CFG for Offload Region



(a) Original Liveness Analysis Results (b) Extended Liveness Analysis Results

Figure 5.7: Results using Classic and Extended Liveness Analysis for Offload Region : the improved analysis generates more information for the compiler backend optimization

offload region analysis. We selected OpenUH as the baseline OpenACC compiler implementation because of its support for multiple languages (C/C++/Fortran) within the front-end, a modular back-end support for high-level intermediate representations and its well-designed and extensible optimization phases. In the next chapter, we describe the loop scheduling transformations which is one of the core contributions of OpenACC compiler.

Chapter 6

Loop-Scheduling Transformation

Programmers prefer to make use of directives via pragmas to offload computation-intensive nested loops onto the massively parallel accelerator devices. Directives are easy to use but are high level and need a smart translation to map iterations of loops to the underlying hardware. This is the role of the compiler which helps users translate annotated loops into kernels to be run on massively parallel architectures. Therefore, the loop-scheduling transformation serves as a basic and mandatory pass in our OpenACC compiler infrastructure. An earlier version of the work presented in this chapter was published in [60, 69, 61].

Loop scheduling is the problem of distributing the loop iterations across multi-processors [65, 38]. However, the loop scheduling becomes challenging when there are multiple levels of parallelism which is the case in GPU architectures. The compiler has to figure out a way to distribute iterations of multi-level loop nests across these multi-dimensional blocks and threads in the GPUs. One of the major challenges of

this compiler transformation is to create a uniform loop distribution mechanism that can effectively map loop nest iterations across the GPU parallel system. As an example, both NVIDIA and AMD GPGPUs have two levels of parallelisms: block-level (Work-Group) and thread-level (Work-Item). Blocks can be organized as multi-dimensional within a grid, and threads in a block can also be multi-dimensional.

OpenACC provides three levels of parallelism for mapping loop iterations onto the accelerators' thread structures: "gang" for coarse-grain parallelism, "worker" for fine grain parallelism, and "vector" for vector/SIMD parallelism. A number of gangs will be launched on the accelerator. Worker parallelism is fine-grain. Each gang will have one or more workers. Vector parallelism is for SIMD or vector operations within a worker. When executing an offload region on the device, one or more gangs are launched, each with one or more workers, where each worker may have vector execution capability with one or more vector lanes. The gangs start executing in gang-redundant mode (GR mode), meaning one vector lane of one worker in each gang executes the same code, redundantly. When the program reaches a loop or loop nest marked for gang-level work-sharing, the program starts to execute in Gang-Partitioned mode (GP mode), where the iterations of the loop or loops are partitioned across gangs for truly parallel execution, but still with only one vector lane per worker and one worker per gang active.

The OpenACC standard allows the compiler some flexibility of interpretation and code generation for three levels' parallelism. However, all workers have to complete execution of their assigned iterations before any worker proceeds beyond the

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

Figure 6.1: Worker Synchronization for Reduction

end of the loop and all vector lanes will complete execution of their assigned iterations before any vector lane proceeds beyond the end of the loop. In Figure 6.1, reduction at worker-level and vector-level both requires synchronization by the end of their loop [69]. Meanwhile, synchronization is only supported inside each thread-block/work-group. In our implementation, the gang is mapped to thread-block level parallelism. While worker and vector are mapped to the thread topology inside the thread block.

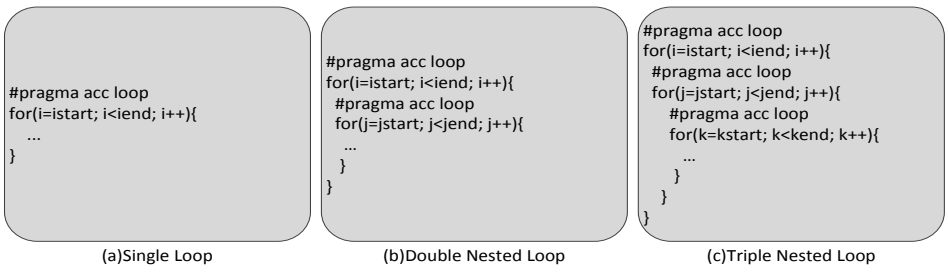


Figure 6.2: OpenACC Loop without scheduling clauses specified

In our design, we propose two different loop scheduling strategies for an OpenACC offload region: kernels and parallel loop scheduling. The difference between parallel and kernels loop scheduling is that we provide more options for the compiler to fine

tune loop scheduling on GPGPUs in kernels region. Both types of loop scheduling can be used for a single loop, double-nested loop and triple-nested loop as shown in Figure 6.2. If the depth of a nested loop is more than 3, the OpenACC `collapse` clause can be used to increase the amount of parallelism. In Figure 6.2, OpenACC loops do not need to specify loop scheduling clauses such as gang, worker and vector and the compiler figures out how to choose loop scheduling for the OpenACC loops. We discuss these cases in detail in the following two subsections. In this latter, we use CUDA language to present the translated kernel code. However, the loop scheduling transformation works both for CUDA and OpenCL since they share similar concepts and can be easily mapped to each other.

6.1 Parallel Loop Scheduling

In the OpenACC specification, parallel region is language *prescription* section which means compiler does exactly what programmers tell. In the loop scheduling, nested gang, nested worker, and nested vector cannot be used, i.e, a gang can only contain worker and vector, and a worker can only include vector. The programmer can create several gangs, and a single gang may contain several workers, and each worker may contain several vector lanes. The iterations of a loop can be executed in parallel by distributing the iterations among one or more levels of parallelism.

```
#pragma acc loop [gang] [worker] [vector]
for(i=0; i<end; i++) {
   statements
}
```

Figure 6.3: General Loop Scheduling in Parallel Region

Table 6.1: OpenACC and CUDA/OpenCL Terminology Mapping in Parallel Loop Scheduling

OpenACC clause	CUDA/OpenCL
gang	blocks in x-dimension of grid.
worker	threads in y-dimension theads in a block.
vector	threads in x-dimension theads in a block.

$$init : iinit = \begin{cases} blockIdx.x + c & \text{only "gang"} \\ threadIdx.y + c & \text{only "worker"} \\ threadIdx.x + c & \text{only "vector"} \\ blockIdx.x * blockDim.y + & \\ threadIdx.y + c & \\ & \text{"gang worker"} \\ blockIdx.x * blockDim.x + & \\ threadIdx.x + c & \\ & \text{"worker vector"} \\ blockIdx.x * blockDim.x * blockDim.y + & \\ threadIdx.y * blockDim.x + & \\ threadIdx.x + c & \\ & \text{"gang vector worker"} \end{cases} \tag{6.1}$$

$$incre : istep = \begin{cases} gridDim.x & \text{only "gang"} \\ blockDim.y & \text{only "worker"} \\ blockDim.x & \text{only "vector"} \\ gridDim.x * blockDim.y & \text{"gang worker"} \\ gridDim.x * blockDim.x & \\ & \text{"worker vector"} \\ gridDim.x * blockDim.x * blockDim.y & \\ & \text{"gang vector worker"} \end{cases} \tag{6.2}$$

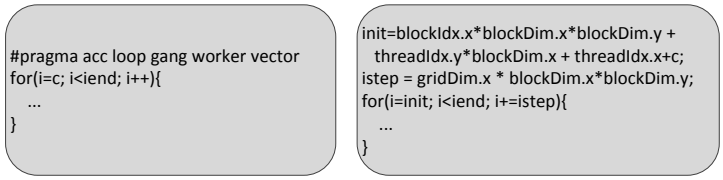
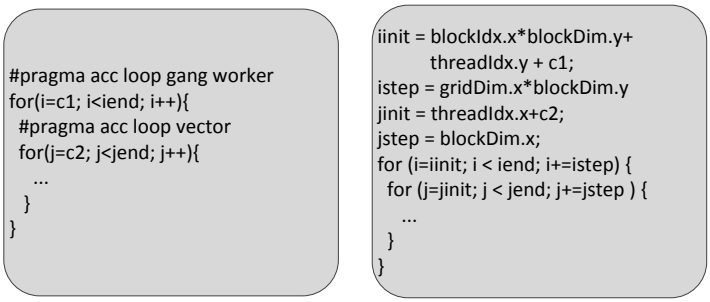


Figure 6.4: Single Loop Transformation in Parallel Region: all the loop scheduling clauses are used in a single loop parallelism.

Table 6.1 shows the CUDA/OpenCL terminology that we use in our parallel region implementation. In parallel region, `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. Equation 6.1 and 6.2 give general transformation rules in terms of loop index initialization and increment for the loop in Figure 6.3. Figure 6.4 demonstrates an example of the loop scheduling transformation. In this example, all the iterations are distributed across gangs, workers and vector lanes. Another



(a) Scheduling-gw-v (b) Scheduling-gw-v in CUDA

Figure 6.5: Two-level Nested Loop in Parallel Region. The scheduling name indicates which loop scheduling clauses are used. In this instance, it means the outer loop is distributed across gang and worker, and inner loop is carried by vector. All of the loop scheduling names follow the same format.

example in Figure 6.5 presents one of the scheduling for two level nested loop.

Table 6.2: OpenACC Loop Schedulings in Parallel Region

Level of Nested Loop	Parallel Loop Scheduling
1	Scheduling-gwv
2	Scheduling-gw-v Scheduling-g-wv Scheduling-g-v
3	Scheduling-g-w-v

There are several advantages by implementing the parallel loop scheduling strageties following Table 6.1. First, this mapping strictly follows the OpenACC standard. Table 6.2 lists all of the available loop schedulings that follow the OpenACC specification. Unlike other OpenACC implementation, our compiler can be fairly compared with commercial compilers by sharing the same application source code. We do evaluate our implementation with commercial compilers. Second, OpenMP 4.0 compiler backend can also share this backend for GPU architecture. In the OpenMP Accelerator Model, three level of parallelisms are available. They are “distribute”, “parallel

for" and "simd". "distribute" means loop iterations are carried by each thread-team, and "parallel for" executes all the iterations by threads in the same team. "simd" then uses SIMD instructions to vectorize the iterations in each thread. The concept are similar to OpenACC three level parallelism.

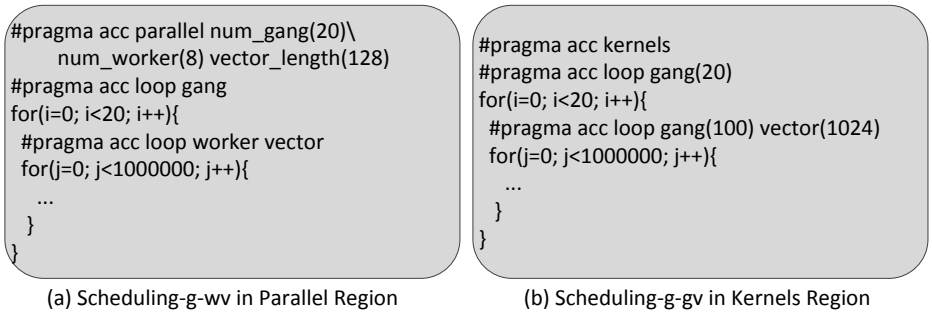


Figure 6.6: Parallel Loop Scheduling Limitation and its solution

However, the parallel loop scheduling cannot exploit the full potential of massively parallel GPU architecture. Both AMD and NVIDIA GPU support multidimensional thread-block and grid. With the OpenACC standard, thread-block can only be maximumly two dimension and thread-block can be only organized into one dimensional grid. This strategy limits the full power of GPU accelerator. First, multi-dimensional grid and thread-block are used to identify the 2D/3D spatial data locality for Texture Memory. Texture Memory is read-only memory that can improve performance and reduce memory traffic when reads have certain access patterns. Stencil-like applications are typical example that requires multi-dimensional thread-block and grid to achieve better performance. Its memory access pattern consumes spatial neighbour points, so the texture memory can be improve the data locality. The Texture Memory optimization will be included in the proposed work. Second, the threads'

scalability becomes a critical issue. The total threads (the number of workers multiplied by the number of vectors) must be less than or equal to a fixed number in one gang (block). For example, each thread-block can have maximumly 1024 threads in an Nvidia GPU. In Figure 6.6(a), the maximum number threads we can create in OpenACC is 20*1024. We could overcome this limitation by utilizing the multi-dimensional topology of the thread block and grid. The solution is to create some extension in kernels loop scheduling to create more threads, in order to increase the Thread-Level Parallelism (TLP). Figure 6.6(b) is one of the kernels loop schedules that can solve the scalability issue since it can create 20*100*1024 threads. Third, the innest most loop is always mapped to x-dimension of thread-block. If the inner most loop index does not repesent consecutively memory access index and the outer loop index does, the performance of parallel loop scheduling will be really bad. Usually, the users can do the loop interchange manually to achieve better performance. However, if the loop interchange is not legal, users may have to reconstruct the data layout in order to coalesce the memory access. Both manually loop interchanging and reconstruct data layout are time-consuming. In next section, we propose kernels loop schediling which is not allowed in the OpenACC specification but does help improve the performance by solving the three disadvantages addressed earlily.

Table 6.3: OpenACC and CUDA/OpenCL Terminology Mapping in Kernels: the int-expr repesents the integer expression that defines the number of the Block/Thread in each dimension.

OpenACC clause	CUDA/OpenCL Description
gangx(int-expr)	Block in X Dimension of Grid
gangy(int-expr)	Block in Y Dimension of Grid
gangz(int-expr)	Block in Z Dimension of Grid
vectorx(int-expr)	Threads in X dimension of Block
vectory(int-expr)	Threads in Y dimension of Block
vectorz(int-expr)	Threads in Z dimension of Block

```
1 #pragma acc loop [gangx|gangy|gangz|] [vectorx|vectory|vectorz]
2 for(i=0; i<end; i++) {
3   statements
4 }
```

Figure 6.7: General Loop Scheduling in Kernels Region

6.2 Kernels Loop Scheduling

In order to take advantage of multi-dimensional grid and thread-block in NVIDIA/AMD GPGPUs, we propose kernels loop scheduling strategies to efficiently distribute loop iterations in an OpenACC kernels region. Figure 6.7 repesents the general form of kernel loop scheduling syntax. In our rules, both *gang* and *vector* can only appear at most one time in the same level of loop. Or users can leave loop scheduling clauses empty and let the compiler generate suitable loop scheduling for the nested loops. Table 6.3 shows the mapping terminology we used from OpenACC to CUDA for kernels directives. The kernels loop scheduling transformation follows Equation 6.3 and 6.4.

$$init : iinit = \begin{cases} blockIdx.x|y|z + c & \text{“gangx|gangy|gangz”} \\ threadIdx.x|y|z + c & \text{“vectorx|vectory|vectorz”} \\ blockIdx.x|y|z * blockDim.x|y|z & \\ \quad + threadIdx.x|y|z + c & \\ & \text{“gangx|gangy|gangz} \\ & \text{vectorx|vectory|vectorz”} \end{cases} \tag{6.3}$$

$$incre : istep = \begin{cases} gridDim.x|y|z & \text{“gangx|gangy|gangz”} \\ blockDim.x|y|z & \text{“vectorx|vectory|vectorz”} \\ gridDim.x|y|z * blockDim.x|y|z & \\ & \text{“gangx|gangy|gangz} \\ & \text{vectorx|vectory|vectorz”} \end{cases} \tag{6.4}$$

```
#pragma acc loop gangy vectory
for(i=c1; i<iend; i++){
  #pragma acc loop gangx vectorx
  for(j=c2; j<jend; j++){
    …
  }
}
```

(a) Scheduling-gv-gv

```
iinit = blockIdx.y * blockDim.y+threadIdx.y+c1;
istep = gridDim.y * blockDim.y;
jinit = blockIdx.x * blockDim.x+threadIdx.x+c2;
jstep =  gridDim.x * blockDim.x ;
for (i=iinit; i<iend; i+=istep) {
  for (j=jinit; j<jend; j+=jstep ) {
    ……
  }
}
```

(b) Scheduling-gv-gv in CUDA

Figure 6.8: Two Level of Nested Loop Scheduling Transformation in Kernels Region

The example in Figure 6.8(a) shows the loop scheduling “gangy vectory” for the outer loop and “gangx vectorx” for the inside the loop. The translated CUDA

version(Figure 6.8(a)) follows the Equation 6.3 and 6.4. In this loop scheduling, both gang and vector are two dimensions which the parallel loop scheduling cannot do. After the mapping, the outer loop thread stride is $gridDim.y * blockDim.y$ and the inner loop thread stride is $gridDim.x * blockDim.x$.

The goal of the kernels loop scheduling is to solve the limitations addressed in parallel loop scheduling section and create as many loop schedulings as possible. From Equation 6.3 and 6.4, we can tell that parallel loop scheduling is a subset of kernels loop scheduling. Due to the rich set of loop scheduling options, either compiler middle-end can have enough search space to tuning the performance or expert users can explicitly specify appropriate loop scheduling for the nested loop.

The loop scheduling strategies proposed for kernels regions are not valid in OpenACC. However, the intention of the kernels construct is to let the compiler analyze and automatically parallelize loops, as well as select the best loop scheduling strategy. So far OpenUH requires the user to explicitly specify these loop schedules, but we are planning to automate this work in the compiler so that the chosen loop scheduling is transparent to the user and the source code also follows the OpenACC standard. Another issue lies with the barrier operation in kernels computation regions. For example, if the reduction is used in the inner loop of scheduling-gv-gv it would require synchronization across thread-blocks, and such synchronization is not supported in both AMD/NVIDIA GPUs. At this situation, the workaround is to use parallel loop scheduling.

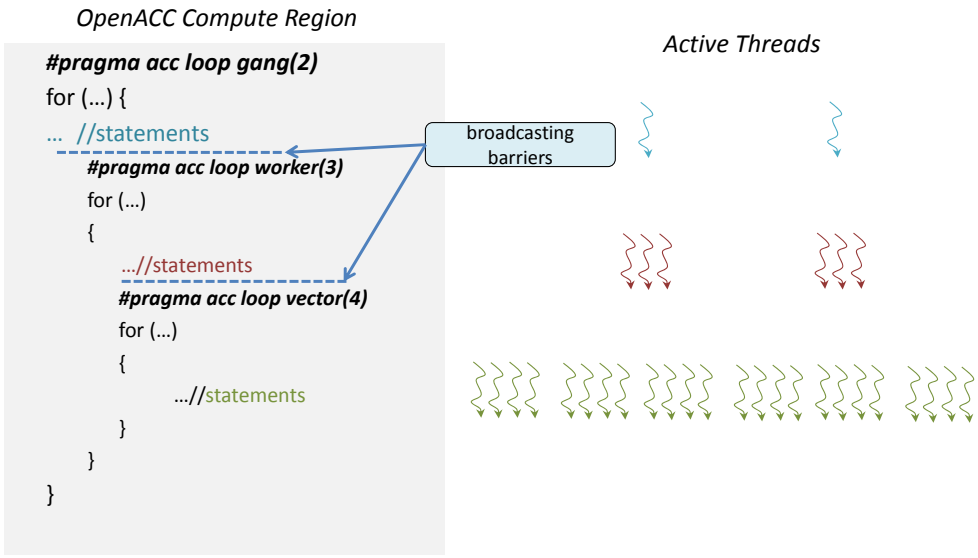


Figure 6.9: Traditional Execution Mode with Synchronization and Broadcasting

6.3 Redundant Execution Mode

Redundant execution is conceptually straightforward. Every thread involved in the redundant execution mode performs redundant computation instead of waiting for master threads computation result with synchronization statement. However, the compute region execution on the device is not fully redundant execution until reaching the vector-level parallelism. The Figure 6.9 shows the traditional execution mode. When the execution has not reached the vector-level parallelism, gang and worker loop parallelism have limited threads actively working on the computation. Results computed at the gang and work parallelism are requiring synchronization and broadcasting to the newly activated threads. The synchronization and broadcasting creates communication overhead.

The traditional execution mode may fit well in the CPU side parallelism. For

example, master thread do the computation and others threads can work on some other computations, like picking up tasks from task queue. The inactive threads do not have to occupy computing resources, such as CPU cores. While on the GPU, this situation does not make any performance benefits. There are several reasons behind it. First, GPU scheduling unit is warp which is a group of threads (typically 32 threads for NVIDIA and 64 for AMD). The cost of scheduling single thread and a warp of threads is the same because both of them consume the same compute resources. Second, threads communication within thread-block relies on Shared Memory and synchronization statements. If each thread perform computation itself, the data is in the register files. Clearly, access latency of Shared Memory is much slower than the register files. Plus the synchronization statements, it is not a decent overhead.

The fully redundant execution mode is proposed to remove such unnecessary overhead. The idea is to privaterize every statement in the gang and worker level parallelism. These statements used to be executed by one thread in each thread block and one thread in each worker. In the gang-level parallelism, each gang only have one active vector lane All thread can perform the computation themselves without synchronization and communication. Basically, the statements in the gang and worker parallelism have to be privaterized for each thread.

However, not all the statements can be privatized and executed redundantly. Consider the OpenACC loop parallelism in Figure 6.10. The loop in both OpenACC loop gang and worker has already assumed that the repsective loop iterations are independent from each other. The only dependence allowed in the loop is reduction

```
  #pragma acc loop gang
for(i =s0; i<n0; i++) {
  S1;
  #pragma acc loop vector
  for(j=s1; j<n1; j++)
  {
    ...
  }
}

```

Figure 6.10: OpenACC Loop Parallelism

```
  #pragma acc loop gang(2)
for(i =s0; i<n0; i++) {
  a[i] += x;
  #pragma acc loop vector(3)
  for(j=s1; j<n1; j++)
  {
    ...
  }
}

```

(a) Array Reduction

```
  #pragma acc loop gang(2)
for(i =s0; i<n0; i++) {
  sum += a[i];
  #pragma acc loop vector(3)
  for(j=s1; j<n1; j++)
  {
    ...
  }
}

```

(b) Scalar Reduction

Figure 6.11: Statement cannot be privatized directly within each thread

operations which requires all the threads synchronization. So the only operation that may prevent the statement privatization is this reduction operation. Both array reduction (Figure 6.11(a)) and scalar reduction operations (Figure 6.11)(b) cannot be privatized and executed redundantly due to the data dependence amongs the threads. Compiler can use the classic dependence analysis to detect such statement and prevent redundant execution mode.

The Figure 6.12 demonstrates the threads computation context in the fully redundant execution mode. All the thread are active from the beginning of the offload kernel. Each thread does the computation itself and store the results in the register files. Synchronization and broadcasting are removed.

Let's consider the an offload loop from sparse matrix-vector loop (Figure 6.13)

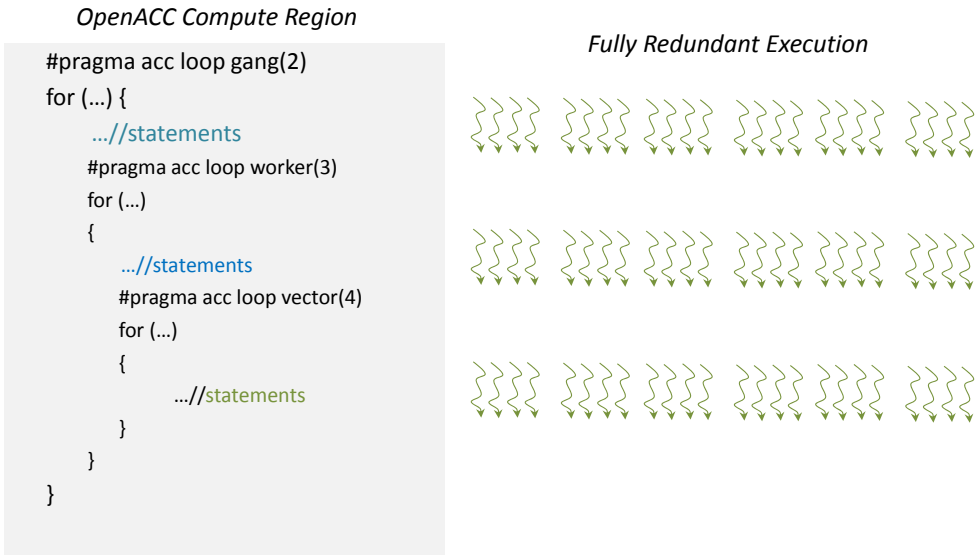


Figure 6.12: Innovative Synchronization Free and Fully Redundant Execution Mode

```
#pragma acc kernels loop gang
  for( i = 0; i < nrows; ++i ){
       val = 0.0;
       nstart = rowindex[i];
       nend = rowindex[i+1];
       #pragma acc loop vector
       for( n = nstart; n < nend; ++n ){
       ...
       }
       r[i] = val;
  }
```

Figure 6.13: sparse Matrix-Vector Loop

```
1     shared float sh_val;
2     shared int sh_nstart, sh_nend;
3     float val;
4     int nstart, nend;
5     for(i=blockIdx.x;i<nrows;i+=blockDim.x){
6       if(threadIdx.x==0) {
7           sh_val = 0.0;
8           sh_nstart = rowindex[i];
9           sh_nend = rowindex[i+1];
10        }
11        __syncthreads();
12        val = sh_val;
13        nstart = sh_nstart;
14        nend = sh_nend;
15        ...//inner loop stmts
16        if(threadIdx.x==0)
17          r[i] = val;
18    }
19
```

```
1     float val;
2     int nstart, nend;
3
4
5     for(i=blockIdx.x;i<nrows;i+=blockDim.x){
6       //removed the thread diverge
7       //and shared memory operations
8
9
10      //no synchronization
11      //no broadcasting from shared memory
12      val = 0.0;
13      nstart = rowindex[i];
14      nend = rowindex[i+1];
15        ...//inner loop stmts
16        //removed the thread diverge
17        r[i] = val;
18    }
19
```

Figure 6.14: (a)Traditional Execution Mode and (b)Redundant Execution Mode with CUDA syntax

6.4 Related Work

Loop Scheduling transformation serves as a basic component in a parallel compiler for high-level directive-based approach. Every compiler has to implement efficient and various strategies to distribute loops iterations across massive threads in accelerators. In this section, both commercial and academic compiler efforts are briefly summarized in terms of loop scheduling transformation.

hiCUDA [31] uses "loop_partition" directive to distribute loop iterations. It has the following syntax:

#pragma hicuda loop_partition [over_tblock [(distr-type)]] [over_thread]
for-loop statement

At least one of the `over_tblock` and `over_thread` clauses must be present. In the `over_tblock` clause, `distr-type` specifies one of the two strategies of distributing

```
#pragma omp parallel for
for(i=1; i<=SIZE ; i ++){
  for(j=1; j<=SIZE; j ++)
    a[i][j] = (b[ i  1 ][j] + b[i+1][j]
    + b[i][ j  1 ] + b[i][j+1])/4 ;
}

```

Figure 6.15: OpenMP original `parallel for` nested loop

```
if(tid<=SIZE){
  for(i =1; i<=SIZE; i++) {
    a[i][tid] = (b[ i  1 ][tid] + b[i+1][tid]
    + b[i][ tid  1 ] + b[i][tid+1]) / 4 ;
  }
}

```

Figure 6.16: CUDA generation from OpenMP parallel region using OpenMPC

loop iterations: blocking (BLOCK) and cyclic (CYCLIC). If it is omitted, the default distribution strategy is BLOCK. The `over_thread` clause does not have such a sub-clause, and the distribution strategy is always CYCLIC. hiCUDA restricts the distribution strategy for the `over_thread` clause to be cyclic. This ensures that contiguous loop iterations are executed concurrently. Since contiguous iterations tend to access contiguous data, this strategy allows for coalescing accesses to the global memory. The inner-most `over_thread` is always mapped to the X-dimension of thread-block. This strategy is similar to the parallel loop scheduling.

OpenMPC [43] interprets the OpenMP 3.0 into CUDA programming model. However, OpenMP 3.0 supports only 1 level of parallelism (nested parallelism are not recommended in OpenMP 3.0). Only one dimension of grid and one dimension of thread-block are supported. This methodology limits the advantage of GPU massively threads architectures which can create multi-dimensional threads topology. OpenMPC cannot handle multiple levels of parallelism. Figure 6.15 and Figure 6.16

```
#pragma hmppcg(CUDA) gridify blocksize "64x1"
#pragma hmppcg(CUDA) permute j,i
for( i = 0 ; i < n; i++ ) {
  for( j = 0 ; j < n; j++ ) {
    prod = 0.0f;
    for( k = 0 ; k < n; k++ )
     prod += A[i][k] * B[k][j];
    C[i][j]= prod;
  }
}
```

Figure 6.17: HMPP Loop Scheduling

present a Jacobi-like loop nest example with OpenMPC framework transformation. OpenMPC compiler uses loop exchange which they called parallel loop-swap to improve the performance of data accesses in the nest loop. The outer and inner nested loops are exchanged, then the loop j iterations are distributed across block and threads which is similar to "gang vector" scheduling. The loop i iterations are carried sequentially by each thread. While Both loops i and j can be parallelized by OpenUH OpenACC without performing loop permutation, if the example in Figure 6.15 cannot perform loop permutation, OpenMPC generates inefficient loop distribution.

HMPP [14] provides fine-control of loop transformation. The loop nest gridification process converts parallel loop nests into a grid of GPU threads. The *gridify* directive can be used to guide the gridification of the loop nest. HMPP implements a 2-dimensional gridification process which is in most of the cases applied automatically. However, in some situations, users may want to specify how to gridify the loops. Figure 6.17 is an example of HMPP offload region. The *blocksize* specifies the number of threads in a block of the gridification. In this case, there are 64 threads in x-dimension and 1 thread in y-dimension. HMPP cannot gridify more than two

```
1  #pragma acc loop gang(n)
2  for (i = 0; i < n; i++) {
3    #pragma acc loop worker(4) vector(128)
4    for (j = 0; j < n; j++) {
5      c = 0.0f;
6      for (k = 0; k < n; k++)
7        c += A[i][k] * B[k][j];
8      C[i][j] = c;
9    }
10 }
```

Figure 6.18: Naive Matrix Multiplication with OpenACC Loop Directives

loops unless the collapse is valid. The loop i and j will be parallelized. The innermost parallel loop is always mapped to the x-dimensional of thread-block. However, the outer loop i represent two out of three array references in coalescing access. The `permute` clause is used to exchange the loop i and j. By comparing to HMPP, our OpenUH implementation covers at most 3 levels of nest loops instead of 2 in HMPP. OpenUH creates more loop scheduling methods so that the loop exchange operation can be avoided in case of unallowed permutation.

PGI delivers two types of mapping in their OpenACC compiler. In the parallel region, PGI interprets each gang to a thread-block in the x-dimension of grid, worker to thread in y-dimension of thread-block and vector to thread in x-dimension of thread-block. While in kernels region, PGI maps each gang to a thread block, vector to threads in a block, and ignore worker [30]. Cray compiler maps each gang to a thread block, worker to warp, and vector to SIMT group of threads [16] in both parallel and kernels region.

The accULL [54] does not cover the loop scheduling transformation in the referened paper. We use a naive matrix multiplication (Figure 6.18) to test this OpenACC

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
if ((i < n) && j < n )
{
  c = 0.0f;
  for (k = 0; k < n; (k++))
    c += A[(i * n) + k] * B[(k * n) + j];
  C[(i * n) + j] = c;
}
```

Figure 6.19: Matrix Multiplication CUDA kernel generated by accULL

compiler. Their compiler ignores the memory coalescing analysis and generates inefficient loop distribution (Figure 6.18). The loop scheduling clauses are also ignored by the compiler. The inner loop is always distributed across y-dimensional of thread blocks and grid.

The Omni OpenACC compiler [58] does not explain the loop scheduling transformation in their paper either. We use the same naive matrix multiplication as accULL to test the Omni compiler. Omni loop scheduling mapping follows OpenACC standard and is similar to the parallel scheduling in OpenUH. However, their loop scheduling is limited compared to our rich set of kernels loop scheduling. Moreover, their transformation introduces too many unnecessary function calls, barriers and thread diverging which hurt performance.

Both Rose-OpenACC [62] and OpenARC [45] map gang to a thread-block in the x-dimension of grid, worker to a thread in x-dimension of thread-block and ignore the vector clause. In this situation, the loop scheduling is very limited and only two levels of parallelism are available.

6.5 Summary

In this chapter, we define the loop scheduling and describe how this transformation is performed to loops in our OpenACC implementation. Basically, it is about how to distribute loop iterations over the GPGPUs threading architectures. The loop scheduling is one of the most important performance factors that have huge impact on the performance of kernel computing. The goal of the loop scheduling targeting GPUs is to maximize the memory coalescing. An innovative redundant execution mode is proposed and implemented in OpenUH to accelerate the kernel execution without synchronization and broadcasting at certain condition. At this point, the baseline of OpenACC compiler implementation is presented. In the next chapter, we cover the data locality optimization in terms of read-only data and register file optimizations.

Chapter 7

Data Locality Optimization

Poor locality of memory access leads to inefficient use of processing capabilities. This direct relation is exacerbated when faced with deep memory hierarchies in GPUs. In order to rectify memory access latencies and improve the **I**nstruction **P**er **C**ycle (IPC), appropriate management of memory resources becomes crucial to ensure performance. The aforementioned memory resources include the set of register files, L1 cache, L2 cache, Read-Only data cache, shared memory, constant memory, texture and global memory. Throughout this work, exploitation of deep memory hierarchy within GPUs is explored as a means to reduce memory access latencies.

Optimizing offload regions written in OpenACC has the following advantages over those written using CUDA or OpenCL:

- Optimizing loop kernels using OpenACC does not require the structure of the loop to be modified. This allows the compiler to normalize loop regions and

use classical optimization techniques (like Loop Nest Optimizations) without affecting the parallelism semantics. This opportunity vanishes once the loop kernels are transformed to lower level representations using OpenCL/CUDA.

- Using high level constructs like array accesses and directives enable the compiler to retain aliasing information, which in turn aids further optimizations. This information is lost in case of lower-level programming constructs within OpenCL/CUDA.

- OpenACC provides for atomic constructs to highlight memory regions participating in atomic operations. Since CUDA/OpenCL provide library interfaces to achieve the same effect, the compiler remains oblivious to its semantics. This further reduces optimization opportunities. Since OpenACC atomic constructs are coupled with specific array accesses, the compiler is aware of the extent of the impact due to the atomic operation.

- Opportunities for locality-based optimizations can further be improved by using OpenACC clauses such as "cache", "tile", and other data clauses. While this can be achieved using OpenCL/CUDA, the programmer is expected to explicitly handle the low-level memory management.

In the following sections, we discuss compiler optimization to solve the data locality problem at some extent. Section 7.1 describes Data Flow Analysis (DFA) to utilize the read-only data cache. The compiler may not find optimal solutions; in these case, the user can use a proposed data clause to identify the read-only data, and the compiler determines from this where the data should be placed. In Section 7.2, we

extend the scalar replacement algorithm to OpenACC *offload*. The new algorithm combines a latency cost model and register information feedback from lower-level vendor's tools to guide the scalar replacement. We name such register optimization as StAtic Feedback-bAsed Register allocation Assistant (SAFARA) which was discussed in the publication [59].

7.1 Read-Only Data Optimization for Offload Region

The GPU architecture contains multiple read-only memory space, each serving a different purpose. The NVIDIA GPU contains three types of read-only memory units: constant memory, texture memory, and Read-Only Data Cache (RODC). RODC actually is texture cache that is used to cache the texture memory. But NVIDIA provides unconventional but convenient way to take advantage of it. In the Figure 7.1, data usually is fetched into L2 cache and then feed into L1 cache. NVIDIA CUDA provides new key words to identify the read-only data which can be fetched from L2 and then reach the register files via read-only data cache. This data path offers additional memory bandwidth which can improve the performance. The AMD GPU also includes constant memory and texture memory. But OpenCL interface that AMD adopts does not have an easy access interface to apply the texture cache for read-only data. If the compiler translates the OpenACC application directly to GPU binary, compiler can use texture fetch instructions to take advantage of such special cache. In this section, we propose a compiler analysis to extract the read-only

data segment and transformations to take advantage of the read-only data segment. The RODC optimization handles data sections referenced using arrays and pointers.

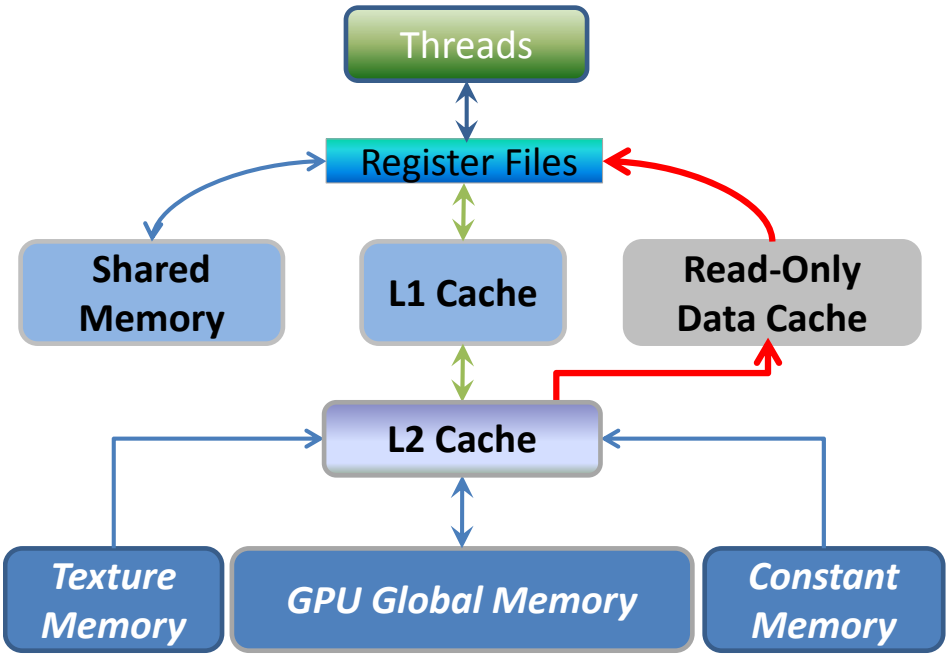


Figure 7.1: Data Path in GPU Memory Hierarchy: the green line the general data path from L2 to register files and the red line the new path for read-only data

Data loaded through the read-only data cache can be much larger and can be accessed in a non-uniform pattern. In the NVIDIA CUDA compiler, the user can give hints by using the "const" modifier to designate certain read-only data and the "__restrict__" keyword to indicate no aliasing. The CUDA compiler generates code to help the hardware to cache the corresponding data into the RODC. We propose and implemented in OpenUH two ways to effectively take advantage of read-only data cache in high-level programming models.

```
#pragma acc data copyin(a,b) \
  copyout(c)\
{
  #pragma acc kernels rodc(a, b)
  {
    #pragma acc loop independent
    for( i=0; i<n; i++)
    {
      c[i] = a[i] + b[i];
    }
  }
}
```

(a) OpenACC Code

```
__global__ void __accrg_vectoradd(
double* __restrict__ c,
const double* __restrict__ b,
const double* __restrict__ a,
int n) {
int i, istep;
i = threadIdx.x +
  blockIdx.x * blockDim.x;
istep = blockDim.x * gridDim.x;
for(; i<n; i+=istep)
  *(c+i) = *(a+i) + *(b+i);
}
```

(b) Translated CUDA Kernel

Figure 7.2: Read-Only Data Cache Optimization Demo

Figures 7.2(a) and (b) give an example of how the proposed clause is used and the translated CUDA code. First, a new data clause is introduced to explicitly identify the read-only data by users. We named it as "rodc," which is only valid in the *kernels* or *parallel* construct. Users can determine which offload region needs the "rodc" clause. Second, the OpenACC compiler can do data flow analysis for the offloaded region, check if an array/pointer buffer will be written, and then generate a read-only array/pointer variables list. In this scenario, aliasing may prevent the compiler from making the optimal decision. However, such aliasing issues can be solved by using the OpenACC loop clause "independent," which tells the compiler that iterations are independent from each other. This is a better solution compared to the new "rodc" clause solution as the code still follows the OpenACC standard and the RODC optimization is applied by the compiler implicitly. Since this optimization is specific to the NVIDIA Kepler architecture, the compiler will bypass when targeting other architectures.

7.2 Register File Optimization

Using compiler directives to program accelerator-based systems through APIs such as OpenACC or OpenMP has increasingly gained popularity due to the portability and productivity advantages it offers. However, when comparing the performance typically achieved to what lower-level programming interfaces such as CUDA or OpenCL provides, directive-based approaches may entail a significant performance penalty. To support *massively parallel* computations, accelerators such as GPGPUs offer an expansive set of registers, larger than even the L1 cache, to hold the temporary state of each thread. Scalar variables are the mostly likely candidates to be assigned to these registers by the compiler.

Hence, scalar replacement is a key enabling optimization for effectively improving the utilization of register files on accelerator devices and thereby substantially reducing the cost of memory operations. However, the aggressive application of scalar replacement may require a large number of registers, limiting the application of this technique unless mitigating approaches such as those described in this paper are taken.

In this section, we propose solutions to optimize the register usage within offloaded computations using OpenACC directives. We first present a compiler optimization called *SAFARA* that extends the classical scalar replacement algorithm to improve register file utilization on GPUs. Moreover, we extend the OpenACC interface by providing new clauses, namely `dim` and `small`, that will reduce the number of scalars to replace. SAFARA prioritizes the most beneficial data for allocation in

registers based on frequency of use and also memory access latency. It also uses a static feedback strategy to retrieve low-level register information in order to guide the compiler in carrying out the scalar replacement transformation. Then, the new clauses we propose will extremely reduce the number of scalars, eliminating the need for more registers.

7.2.1 SAFARA: StAtic Feedback-bAsed Register allocation Assistant for GPUs

Scalar replacement (SR) is a classical optimization that can be applied to improve utilization of register files. In this section we present what we view as limitations of the state-of-the-art algorithm, by Carr and Kennedy [39, 22], and we introduce our new algorithm called SAFARA.

7.2.1.1 The Carr-Kennedy Algorithm

The scalar replacement algorithm[22, 23] includes three phases: (1) a dependence distance-based data reuse analysis, (2) a moderation model of register pressure, and (3) the scalar replacement transformation. The Carr-Kennedy algorithm [23] uses input and flow dependence analysis to find the array references. If all of the reused memory references found in the first phase are replaced with scalar, the performance of the application may slow down because of register spilling. The moderation of register pressure is used to find out the most beneficial memory references that can be replaced with scalars. Once the memory references that are chosen to be

replaced with scalars are determined, the compiler performs the scalar replacement transformation to replace the memory references with scalars. However, the Carr-Kennedy algorithm cannot be directly applied to OpenACC offload regions. In this section, we present its limitations when applied to GPUs.

7.2.1.1.1 Creation of Inter-iteration Dependences in Parallelized Loops

The first limitation of the Carr-Kennedy algorithm is that it may translate an independent loop into a dependent loop that cannot be parallelized. An example of a parallel loop is provided in Figure 7.3, where the array references `b[i]` and `b[i+1]` introduce an input data dependence edge which has a dependence distance of 1. Therefore, the data loaded in `b[i+1]` at iteration i will be used in array reference `b[i]` at iteration i+1. The Carr-Kennedy algorithm will detect the data reuse opportunities and perform the scalar replacement optimization. The loop will be thus transformed into the code shown in Figure 7.4, which has only 1 array reference in the loop body. The loop in Figure 7.4 introduces loop-carried flow dependences across iterations between `b1` and `b[i+1]`. Consequently, the loop cannot be parallelized. This conflicts with the goal of OpenACC which is to expose parallelism to be exploited by the massively parallel accelerator. In fact, executing the loop sequentially by each thread of the GPU will lead to a significant performance penalty. Therefore, in our solution, we will prevent scalar replacement from being performed across iterations, if the loop can be parallelized.

7.2.1.1.2 Cost Model not Adapted to GPUs

The memory access latency in GPUs is different from the one of traditional CPU systems. In the Carr-Kennedy

```
1 for(i=1; i<=SIZE ; i ++)
2   a[i] = (b[i] + b[i+1])/2;
```

Figure 7.3: Before SR: iterations are independent

```
1 b1=b[1]
2 for(i=1; i<=SIZE ; i ++){
3   b2=b[i+1];
4   a[i] = (b1 + b2)/2;
5   b1 = b2;
6 }
```

Figure 7.4: After SR: iterations are dependent

algorithm, the metric used is how many memory accesses can be removed. For this, a model of register pressure moderation is designed to select the most beneficial references to be transformed into scalar variables if limited register files are available. For instance, in Figure 7.5, references to each array `a` and `b` require 3 temporary variables. In the Carr-Kennedy algorithm, when the number of available registers is limited, the array references of `a` have higher priority to be replaced with scalar variables because it is used one more time than `b`. However, in GPUs, another metric should be taken into account due to the memory hierarchy of GPUs; this is the second limitation of this algorithm for our purpose. In fact, since iterations in Loop `j` are distributed across the x-dimension of each thread in each thread block, the memory access in `a` are coalesced within a warp. Meanwhile, the memory accesses in `b` are uncoalesced within a warp. Thus, the memory access latency of `b` is much higher than that of `a`. In this case, replacing array references of `b` will have a better benefit than replacing the references of `a`.

```
#pragma acc loop gang vector
for(j=1; j<=JSIZE ; j ++){
    c[j] = b[j][0] + b[j][1];
    d[j] = c[j] *b [j][0];
#pragma acc loop seq
    for(i=1; i<=ISIZE ; i ++){
        a[i][j] += a[i-1][j] + b[j][i-1] +
             a[i+1][j] + b[j][i+1];
    }
}
```

Figure 7.5: Sample OpenACC program before SR

7.2.1.2 SAFARA: StAtic Feedback-bAsed Register allocation Assistant for GPUs

SAFARA addresses the two limitations presented in the previous section. For the 1st limitation, the scalar replacement approach can be divided into intra-iteration and inter-iteration transformations. If the loop is identified as parallelized in OpenACC, only the intra-iteration SR is performed to avoid to sequentialize it. Otherwise, if the loop is sequential, then inter-iteration SR can be safely applied. As for the 2nd limitation, three new components are integrated.

1. First, during the dependence analysis to retrieve data reuse, we count how many times every array reference is used (read/write). Then, array references are classified into four categories according to the memory hierarchy in the GPU: shared, constant, read-only (available in NVIDIA Kepler GPUs only) and global memory access [1]. Read-only and global memory data accesses can

[1]Note that in our implementation, we only consider read-only and global memory accesses.

be further divided into coalesced and uncoalesced accesses. Each of them has different memory access latency [25].

2. Second, we use GPU tools to pinpoint the register usage information and then feed it back to the OpenACC compiler to perform the SR. The NVIDIA GPU tool we used is called *PTXAS Info*.

3. Third, using the information from the first step, we estimate the cost of each array reference R belonging to a memory space M, using the formula *reference_count(R)* $\times$ *memory_access_latency(M)*.
   Then, all array references can be sorted from higher to lower cost. After that, we select the most beneficial memory references to be replaced by scalar variables.

4. Go to Step 2 until all the registers are used or all the reused references are replaced.

In the following subsections, we explain in details the methodology followed by these steps.

7.2.1.2.1 Array Reference Analysis in SAFARA Memory access pattern analysis is introduced into SAFARA to classify different memory access modes. Basically, the memory access latency depends on where the data is located and how the data is accessed. There are several different memory spaces in modern GPU architectures. For the NVIDIA Kepler GPUs, there are shared memory, read-only global data, read/write global data, constant memory and texture memory. Read-only data

can be placed in the global memory and cached by the read-only data cache in each SM. While building the dependence graph, the compiler performs array index analysis to determine if the memory access is coalesced or not. The index analysis that is used in our algorithm is inspired from [36] which proposes a mathematical model that captures and characterizes memory access patterns inside nested loops. This is used to recognize if the memory access is coalesced or not.

7.2.1.2.2 Iterative Register Information Feedback to SAFARA We use register utilization information from GPU tools to improve the scalar replacement transformation done in SAFARA. In traditional CPU compilers that perform register allocation and directly generate actual assembly code, this register information is available during compile time. However, since GPU architectures change dramatically between generations, compilers for GPUs generate a stable, virtual ISA that spans multiple GPU generations and use pseudo registers. For example, NVIDIA uses "PTX". There are unlimited pseudo register numbers available in the virtual ISA. The compiler cannot determine how many hardware registers have been used. Vendors, including NVIDIA, provide closed-source low-level assembler tools to translate the virtual ISA into actual GPU assembly code and allocate hardware registers.

In our work, we propose to assist the compiler by using feedback information from these tools to calculate how many hardware registers are available. Moreover, backend compilation is performed multiple times. The first time does not perform any scalar replacement; it is only dedicated to invoking the GPU assembler tool to output the hardware register usage information. The following compilation combines

the register usage information and register upper limit specified by the hardware limit (for instance the maximum number that can be used in NVIDIA Kepler GPU is 255 registers per thread) to determine the availability of registers. If there are available registers, the scalar replacement optimization is invoked. The compiler analysis lists all the memory references that can satisfy the scalar replacement requirements. If the number of candidates is less than the available register count, all of them are replaced by scalars. Otherwise, the cost model based on array references selection is invoked.

7.2.1.2.3 Sophisticated Cost Model for Array References Selection In contrast to traditional CPU architectures, overuse of GPU register files causes severe performance degradation due to register spilling as well as lowering of thread concurrency. This raises the issue of how to select good memory references if their number is larger than the number of available registers. We use a more sophisticated cost model to prioritize memory accesses for replacement. It is based on the memory access latency, which is used to estimate the potential cost of different memory accesses. The model consists of two factors: memory access latency L and references count C. The potential access cost is computed as $L \times C$.

7.2.1.2.4 Running Example with SAFARA In the following, we demonstrate the application of SAFARA on the example shown in Figure 7.5. After the first time we apply the first iteration of SAFARA, we suppose that the GPU tool outputs 26 as the number of registers that are used. We suppose that the hardware limit is only 30 registers. So, the number of available registers found by our algorithm is 4. In

the second iteration of SAFARA, the scalar replacement is applied on Array `b` using 3 registers (recall that Array `a` is coalesced and should not be put in registers) and the code will be transformed into the code shown in Figure 7.6. We show here only two iterations for lack of space.

```
#pragma acc loop gang vector
for(j=1; j<=JSIZE ; j ++){
    c[i] = b[j][0] + b[j][1];
    d[j] = c[j] *b [j][0];
    b0=b[j][0];
    b1=b[j][1];
    for(i=1; i<=ISIZE ; i ++){
        b2 = b[j][i+1];
        a[i][j] = a[i-1][j] + b0
              + a[i+1][j] + b2;
        b0 = b1;
        b1 = b2;
    }
}
```

Figure 7.6: Sample OpenACC program after SAFARA

7.2.2 Proposed Extensions to openACC: `dim` and `small` New Clauses

Scalar replacement is a classical memory optimization algorithm to reduce redundant memory access. In the previous subsection 7.2.1, we presented an extension to SR by providing different techniques and we called this extension SAFARA. However, the aggressive application of scalar replacement increases register pressure, which

may lead to low threads occupancy or cause register spilling, and thus hurt performance. To confirm this result, we perform a study on the SPEC benchmark suite and we show experimental results in Figure 7.7. The experimental setup is provided in Section8.1. In this study, we found that SAFARA provides either very small performance improvement or sometimes slows down the application when registers are exhaustively used by threads, because this leads to low threads occupancy; we also found that no register spilling happened based on SAFARA feedback information. Finding the best combination between what is the optimal number of registers to use by each thread and how much scalar replacement we allow the compiler to perform is a complex problem [63]. In this section, we propose a solution at the API level of OpenACC to reduce the number of scalars that will be held potentially in registers. This will save some registers to use by each thread and thus increase threads occupancy.

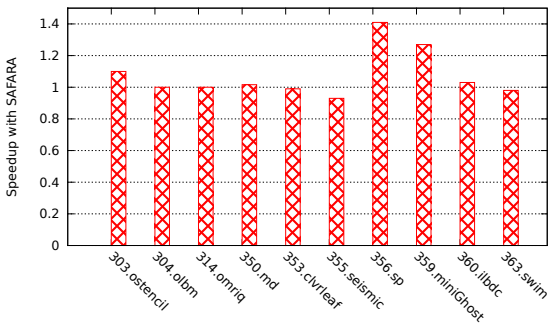


Figure 7.7: Speedup results of SPEC benchmark suite with SAFARA

Array references are frequently used in high-level programming models like OpenMP and OpenACC that are API extensions to C/C++ and Fortran languages. SPEC benchmarks for example contain a lot of array references. When the offload region

is translated into the GPU lower level kernel routines (using CUDA for example), the array reference is represented using a pointer and offset calculation operations. The idea behind the two clauses we want to introduce is to save some register files for scalar replacement by removing unnecessary array offset computations. The programmer can directly use these two clauses to pass array information to the compiler. We provide a motivation example in Figure 7.8 which shows a snippet of one of the offload regions in SPEC Accelerator 355.seismic benchmark. In this code, Loop j iterations are distributed across the y-dimensional threads and Loop i iterations are carried by x-dimensional threads in the GPU. The iterations in Loop i and Loop j are evenly distributed across all the threads and each thread only executes one iteration. The innermost Loop k is executed sequentially and SAFARA can be applied over array references across the k iteration.

```
1  !$acc kernels loop gang(NY/2) vector(2)
2  do j = 2,ny
3    !$acc loop gang((NX-1+63)/64) vector(64)
4    do i = 1,nx-1
5      !$acc loop seq
6      do k=2,nz
7        ...
8        value_dz = (vz_1(i,j,k)-vz_1(i,j,k-1))/h &
9                 + (vz_2(i,j,k)-vz_2(i,j,k-1))/h &
10                + (vz_3(i,j,k)-vz_3(i,j,k-1))/h
11       ...
12     enddo
13   enddo
14 enddo
```

Figure 7.8: Snippet code from SPEC 355.seismic benchmark

7.2.2.1 `dim` Clause

In multiple scientific kernels including SPEC, the arrays are allocatable arrays (case of Fortran) or Variable-Length Arrays (VLA) (case of C/C++). These arrays are dynamically allocated; the dimensional information of the array is held in a dope vector data object generated by the compiler. In the example provided in Figure 7.8, the offset calculation requires five additional compiler-generated temporary variables to hold the lower bound and length for each dimension in Fortran, while in VLAs in C/C++, we need temporary variables to hold the length for each dimension, since the lower bound is always zero. The listing below details the computation of the offsets and reference addresses for the three array references in Figure 7.8. The variables t0,...,t14 hold dimensional information for each array. Note that 15 scalar variables are used to keep the boundary information and calculate the offsets of the three arrays.

```
1 offset0 = (i-t0) + t3 * ((j-t1) + t4 * (k-t2))
2 vz_1(i,j,k) --> *(vz_1 + offset0)
3
4 offset1 = (i-t5) + t8 * ((j-t6) + t9 * (k-t7))
5 vz_2(i,j,k) --> *(vz_2 + offset1)
6
7 offset2 = (i-t10) + t13*((j-t11)+t14*(k-t12))
8 vz_3(i,j,k) -->  *(vz_3 + offset2)
```

However, these three arrays have exactly the same dimensions. If the compiler has this equality of dimensions information, the array references address computation can be optimized into a simplified version which is shown in the listing below:

```
1 offset0 = (i-t0) + t3 * ((j-t1) + t4 * (k-t2))
2 vz_1(i,j,k) --> *(vz_1 + offset0)
3 vz_2(i,j,k) --> *(vz_2 + offset0)
4 vz_3(i,j,k) --> *(vz_3 + offset0)
```

At the compilation time, the compiler has no idea whether these arrays have the same dimension. Therefore, we propose a new clause `dim` to be added to `kernels` and `parallel` directives to specify which arrays share the same dimension(s). At the GPU code generation phase, the compiler can take advantage of this clause information and optimize the offset computation. Note that, in this specific example, the number of registers needed can be reduced to 5, which corresponds to (number of scalars)/(number of arrays). The `dim` clause syntax is shown below:

```
1 Fortran:
2 !$acc kernels/parallel &
3    dim([(lb1:len1,...,lbN:lenN)](A1,...,),...)
4
5 C/C++:
6 #pragma acc kernels/parallel \
7    dim([len1]...[lenN](A1,...,),...)
```

Note that dimension data in the clause syntax is optional. If the user does not specify the dimension data, as follows, the compiler can automatically load lower bounds and length data from one of the array's dope structure:

```
!$acc kernels dim( (vz_1, vz_2, vz_3))
```

However, we recommend providing complete information (dimensions and arrays) because the compiler can simplify further the offset computation, in particular when the lower bound is zero, as below:

```
1 !$acc kernels &
2 dim((0:NX, 0:NY, 0:NZ)(vz_1, vz_2, vz_3))
```

7.2.2.2 `small` Clause

On 64-bit machines, the compiler uses a pointer type of 64-bits size while an offset is also a 64-bit integer. However, if the array size is less than 4GB, array references address computation can be represented with 64-bit addresses and 32-bit integer offsets. The size of register files used for offset computations can thus be reduced by up to half. In fact, small array sizes are common in current applications due to the limited device memory. Note that when the array is a static array in both C or Fortran, the compiler can detect the array size and decide whether 32-bit integers are enough to handle the offset value computation. However, when an allocatable array or VLA is used, the compiler cannot figure out the array size. By default, the compiler will use 64-bit integer to be safe. Therefore, we propose the new clause

small to tell the compiler that the offset of an array can be represented within a 32-bit integer. Here, a small array means the array size is smaller than 4GB and array references of such arrays can be represented with an array address plus a 32-bit integer offset. The `small` clause syntax is shown below:

```
Fortran:
!$acc kernels/parallel &
   small(A1,...,An)

C/C++:
#pragma acc kernels/parallel \
   small(A1,...,An)
```

If we apply this clause to the previous example, as follows, the register number can be reduced up to half:

```
!$acc kernels &
dim((0:NX, 0:NY, 0:NZ)(vz_1, vz_2, vz_3)) &
small(vz_1, vz_2, vz_3)
```

7.3 Related Work

While the original scalar replacement algorithm was proposed more than 20 years ago, computer architectures have evolved considerably since then. Numerous past works exist for improving this algorithm in many aspects. Sastry [55] and Sarkar [57] both proposed new algorithms based on the SSA form. Budiu [21] presented a simplified Carr-Kennedy [23] inter-iteration register promotion algorithm to handle a number of dynamically executed memory acceses. Hall [56] demonstrated an algorithm to increase the data reuse across multiple loops. Baradaran [20] described a register allocation algorithm that assigns registers to array references replaced with scalars along the critical paths of a computation. However, none of these algorithms can effectively work for GPU architectures. While the register moderation model in the Carr-Kennedy algorithm works well for a traditional CPU memory hierarchy, the cost model-based strategy in SAFARA selects the most profitable array reference candidates for mapping to register files through scalar replacement.

Budiu [21] proposed a simplified Carr-Kennedy iter-iteration register promotion algorithm to handle dynamically executed memory accesses. In their approach, the compiler generates a flag represented by a single bit that is associated with each value to be scalarized, as well as code that dynamically updates the flag. The flag can be inspected at run time to avoid redundant load operations, and their algorithm ensures that only the first load and last store take place. Since this algorithm inserts a large number of additional control flow statements throughout the code, the resulting behavior when executed on a GPU is thread divergence. This will produce additional

overhead and significantly degrade performance. In short, this algorithm is not GPU-friendly.

Andión [19] presented a new scalar replacement algorithm for offload computation regions specified using the HMPP directive interface [14]. This work is most similar to our paper. Both works target a GPU offload region expressed using high-level directives. Nevertheless, their algorithm over-utilizes the register files for each thread, which may cause severe performance penalties due to register spilling and GPU low threads occupancy. There are three additional limitations. First, array references with index expressions only consisting of the parallelized loop indices are potential reuse candidates. However, their approach does not handle loop-invariant variables used in array subscripts, which can also be used to estimate the reuse of array references. Second, the array reference access mode is not considered, and the cost of different types of memory access varies. For example, if a read-only array is present in the Read-Only Data Cache, then it will be accessed in a coalesced manner and it is not beneficial to replace them with scalars. Third, all the reused references are replaced with scalars. This does not take into account how many hits each reference induces. Therefore a replacement may not be beneficial in some instances when a low amount of hits occurs.

Regarding improving register usage using extensions to the API and RODC, to the best of our knowledge, our work is the first dissertation to propose such optimizations.

7.4 Summary

In this chapter, we present two aspects optimization targeting on different on-chip memory resources. First, we provided compiler support for Read-Only Data Cache optimization which can potentially improve cases when there was a large number of read-only data/buffer in the offloaded compute regions. Second, we present an extension to the classical scalar replacement algorithm called SAFARA that is based on feedback information regarding register utilization and a memory latency-based cost model to select which array references should be replaced by scalar references. In the register optimization, we propose two new clauses to add to OpenACC, namely *dim* and *small*, to reduce the register usage. In the next chapter, we evaluate the SPEC and NAS OpenACC benchmarks which we have used to study the effectiveness of our OpenACC implementation and optimization algorithms.

Chapter 8

Performance Evaluation with OpenUH

This chapter presents empirical results for our implementation with two sets of benchmarks: SPEC ACCEL suite [37] and NPB OpenACC suite [68]. The evaluation is organized into three parts. The first section demonstrates the data locality optimizations on both NVIDIA and AMD GPUs. It indicates the effectiveness of optimization strategies proposed on these platforms. This is followed by the verification of performance portability across these platforms; performance results of the OpenACC implementation within the OpenUH compiler is presented. Finally, the implementation within OpenUH is compared with state-of-the-art commercial compiler - NVIDIA's PGI OpenACC compiler.

8.1 Experimental Setup

This section describes the characteristics of the test platform.

8.1.1 NVIDIA Test bed

The NVIDIA infrastructure comprises of a K20Xm GPU with 5GB global memory. The host CPU is an 8-core Intel Xeon x86_64 CPU coupled with 32GB of main memory. CUDA 6.5 is used for the OpenUH backend GPU code compilation with "-O3" optimization. The target and optimization flags for OpenUH are listed in Table 8.1.

For the comparative analysis, we used one of the major commercial OpenACC compiler vendors, namely PGI V15.9. We use "-O3,-acc -ta=nvidia,cc35" for the PGI compiler options. To obtain reliable results, all experiments were performed five times and then the average performance was computed.

Table 8.1: OpenUH Compilation Flag

Flag	Summary
-accarch:amd	Targeting AMD discrete GPUs
-accarch:apu	Targeting AMD APUs
-accarch:nvidia	Targeting NVIDIA GPUs
-rodc	Read-Only Data Cache Optimization
-safara	GPU Register Optimization

8.1.2 AMD Test bed

The AMD platform comprises of an APU and a discrete GPU card are used. The AMD APU that we used is R7 GCN-based and integrated with CPU on the same die. APU we used is Kaveri 7850K. The machine has 16GB DDR3 memory which is shared by both CPU and GPU. The CPU and GPU in this APU architecture can access the same unified virtual memory space. The AMD discrete GPU that we used is Firepro W8100 with 8GB global memory. The machine with AMD W8100 includes Intel Xeon CPU E5520 with 16 cores as host CPU and 32GB memory. AMD APP SDK 3.0 is used for OpenUH backend for compiling the OpenCL kernel functions to GPU binary with "-O3" optimization.

Table 8.2: Evaluation Platform

Config	NVIDIA K20mc	AMD W8100	AMD 7850K
Compute Units	14	40	8
Cores	2644	2560	512
GPU Memory	6GB	8GB	16GB
Bandwidth(GB/s)	249.6	320	33
FP32 TFLOPS	4.0	4.2	0.737
FP64 TFLOPS	1.3	2.1	0.046

8.1.3 Benchmarks - SPEC ACCEL and NAS OpenACC

The SPEC ACCEL and NAS OpenACC benchmarks were chosen because the code-size and the complexity are comparable to real world applications. The SPEC OpenACC suite includes both C and Fortran applications and is used to evaluate our compiler implementation and optimization algorithms this dissertation. It comprises of 15 OpenACC benchmarks in SPEC ACCEL OpenACC Suite, ten of which

were used as part of the experimental study[1]. These are listed in Table 8.3. The benchmarks cover the different application domain and range from single GPU kernel upto 116 GPU kernels. The SPEC ACCEL benchmark applications all provide three different input data set sizes: test, train, and ref. The test data is the smallest size and is used to verify the compiler implementation that generates correct results. The reference date set is the largest size and was used to generate benchmark results.

NPB OpenACC benchmarks are written in C. These open-source benchmarks, written using multiple programming models are a good target to evaluate current and upcoming multi/many core hardware architectures. In this benchmark suite, there are eight benchmarks, out of which the following seven were used: EP (Embarrassingly Parallel), CG (Conjugate Gradient), MG (MultiGrid), SP (Scalar Pentadiagonal), LU (Lower-Upper symmetric Gauss-Seidel), BT (Block Tridiagonal) and FT (Fast Fourier Transform)[2]. The suite supports multiple classes which correspond to different sizes of the input data set - S, W, A, B, C, etc. FT was evaluated using "B" and "C" for the rest[3].

[1]Four of these fifteen come from NPB OpenACC Benchmarks (352.ep, 354.cg, 357.csp and 370.bt) and therefore is not include as part of the SPEC suite. One other benchmark - the "351.palm" has been left out of the study due to the inability of the OpenUH compiler to compile programs characterized with intrinsic function in offload region

[2]The benchmarks "IS" has been left out of the study. It requires scan parallel implementation which is not supported by the OpenUH compiler

[3]Four of these benchmarks - EP, CG, BT and SP benchmarks were adopted within the SPEC ACCEL V1.0 [37] suite

Table 8.3: SPEC ACCEL Benchmarks : kernels mean the GPU kernel functions that the compiler generates.

Benchmarks	Langauge	GPU Kernels	Data Size	Application Domain
303.ostencil	C	1	REF	Thermodynamics
304.olbm	C	1	REF	Computational Fluid Dynmaics, Lattice Boltzmann Method
314.omriq	C	1	REF	Medicine
350.md	Fortran	3	REF	Molecular Dynamics
353.clvrleaf	C, Fortran	116	REF	Explicit Hydrodynamics
355.seismic	Fortran	16	REF	Seismic Wave Modeling
356.sp	Fortran	71	REF	Scalar Penta-diagonal solver
359.miniGhost	C, Fortran	51	REF	Finite difference
360.ilbdc	Fortran	1	REF	Fluid Mechanics
363.swim	Fortran	22	REF	Weather

8.1.4 Normalization of results

The execution time depicted in some of the plots in the following sections are normalized so that the results can be accomodated in a single frame. The execution time of benchmarks have a wide range varying from as low as a few seconds to several hundreds of seconds. The formula used was:

$$Norm(Compiler) = \frac{ExeTime(Compiler)}{max(ExeTime(OpenUH), ExeTime(PGI))}$$

8.2 Experimental Results

In this section, we demonstrate our data locality optimizations on both NVIDIA and AMD platform. Two kind of optimizations have been proposed in Chapter 7:

Table 8.4: NAS OpenACC Benchmarks

Benchmarks	GPU kernels	Data Size	Application Domain
BT	46	C	Block Tridiagonal Solver
CG	17	C	Conjuage Gradient
FT	13	B	Fast Fourier Transform
EP	5	C	Embarrassingly Parallel
LU	56	C	Lower-Upper symmetric Gauss-Seidel
MG	18	C	MultiGrid
SP	65	C	Scalar Penta-diagonal solver

Read-Only Data Cache and register optimizations.

8.2.1 Data Locality Optimization

8.2.1.1 Read-Only Data Cache

Read-Only Data Cache (RODC) is applied to the NVIDIA GPU which supports global read only data path. NVIDIA CUDA supports read-only global memory through the same cache used by the texture pipeline since Kepler architecture. Texture Cache is also available in AMD GPUs. However, AMD OpenCL interface offers an Image Buffer that enables caching of global read-only data into the Texture Cache. Typically OpenACC applications are characterized with use of array data structures that are read-only in some offload regions, but written to in other regions. If such data is placed with the Image Buffer during the read-only session, it nees to be copied over to the global memory during the write operations. This excessive data movement between the Image Buffer and the global memory has the potential of severely degrading the performance. Therefore, we only verify RODC

on the NVIDIA platform. This issue can be resolved by generation of fetch instructions from the texture-cache during the conversion to GPU-ISA instructions by the compiler(during code-generation).

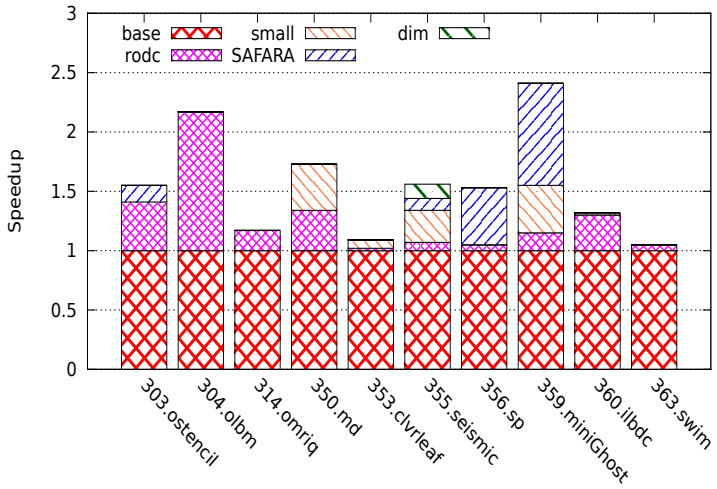


Figure 8.1: SPEC ACCEL SUITE Performance Improvement by RODC and Register Optimization on NVIDIA K20mc

In Figures 8.1 and 8.2, the usage of RODC and Register optimizations shows the speedup in performance for SPEC ACCEL and NAS OpenACC benchmarks on NVIDIA Kepler K20mc. Typically, applications that use a large number of read-only buffers/arrays benefit from the RODC optimization. EP from NAS is the only benchmark does not speedup by the RODC optimization. The execution time of EP is dominiated by one offload region which performs reduction operations. It does not have much read-only array in this offload region.

Figures 8.1 and 8.2 also show the speedup results after applying first the 'dim' and 'small' clauses to reduce the number of required registers and then 'SAFARA' to make the scalar replacement. Note that Benchmarks 303, 304, 314 are C benchmarks

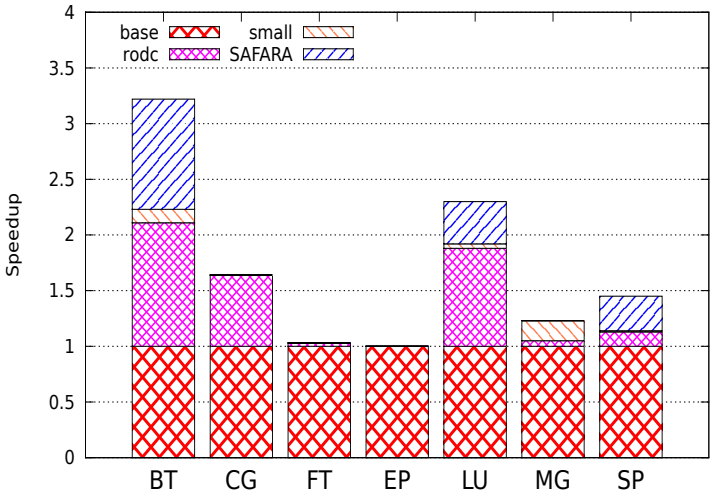


Figure 8.2: NAS OpenACC Benchmarks Performance Improvement by RODC and Register Optimization on NVIDIA K20mc

and pointer operations are used in the offload regions; thus the dim and small clause are used here. The dim clause is used in 355 and 356, which are Fortran applications and include allocatable arrays. By comparing with Figure 7.7, the speedup is boosted up to 1.46x and performance did not slow down anymore after introducing `small` and `dim` clauses (note how 355.seismic overused the register files in Figure 7.7 and the application did slow down). Meanwhile many registers and operations are dedicated for array offset computations. The two clauses helped in reducing the number of variables used in these computations and thus improving the performance.

The seven NAS OpenACC benchmarks are written in C language and use static arrays; so a `dim` clause is not useful in this case. BT, LU and SP have several kernels that contain uncoalesced memory accesses. Thus, SAFARA can help in reducing such costly accesses by prioritizing their placement in register files. However, regarding the `small` clause, among LU, SP, and BT, only BT showed benefit from using this

clause. The reason is not known to us because the actual register allocation is done at a much lower level of the CUDA compiler, which we do not control.

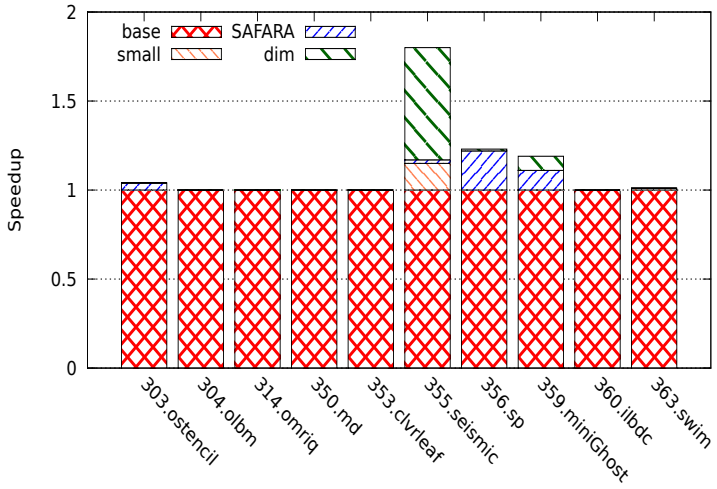


Figure 8.3: SPEC ACCEL SUITE Performance Improvement by Register Optimization on AMD W8100

8.2.1.2 Register Optimization

Register optimization which includes SAFARA and assisting clauses *dim* and *small* is evaluated on both AMD and NVIDIA GPUs.

Since the AMD platform does not provide convenient interfaces to promote read-only data, we only evaluate the Register File optimization (Figure 8.3 and Figure 8.4). Like the SPEC benchmarks' results for NVIDIA, 303.ostencil, 355.seismic, 356.sp and 359.miniGhost which include the data reuse inside the offload region benefit from SAFARA by placing the reused data into register files. 355.seismic is the benchmark that motivates the new clause *small* and *dim*. It receives 63% of performance improvement by providing *dim* clauses to each offload region. The *dim* clause can

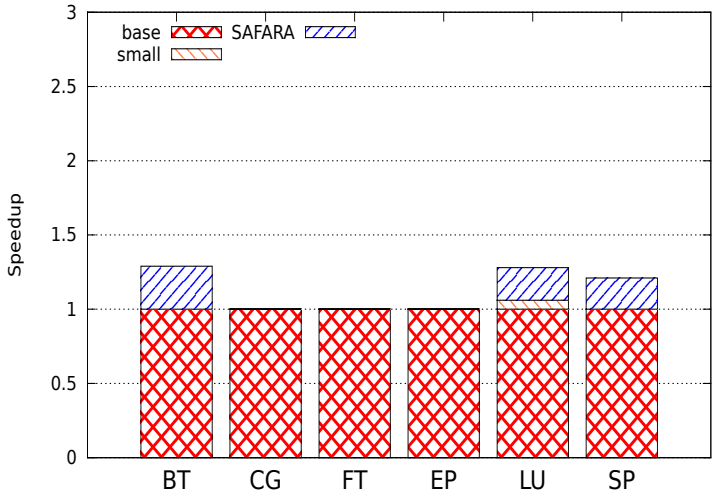


Figure 8.4: NAS OpenACC Benchmarks Performance Improvement by Register Optimization on AMD W8100

largely reduce the register usage and improve the kernel efficient if many allocatable arrays are involved in an offload region. For the NAS benchmarks, BT, LU and SP also have the performance improvement by applying the SAFARA. However, the feedback mechanism is not used in AMD platform since the OpenCL API does not provide the register file information. So once the data reuse is found, scalar replacement is performed. It is possible to cause the register spill and the performance may not be improved as we expected in NVIDIA.

From NVIDIA results in Figures 8.1 and 8.2 and AMD results in Figures 8.3 and 8.4, the AMD platform does not generate as much performance improvement as what we have in NVIDIA platform. However, it does not mean AMD GPU runs slower than NVIDIA GPU. Next section, we discuss the performance different between NVIDIA and AMD GPUs in Section 8.2.2.

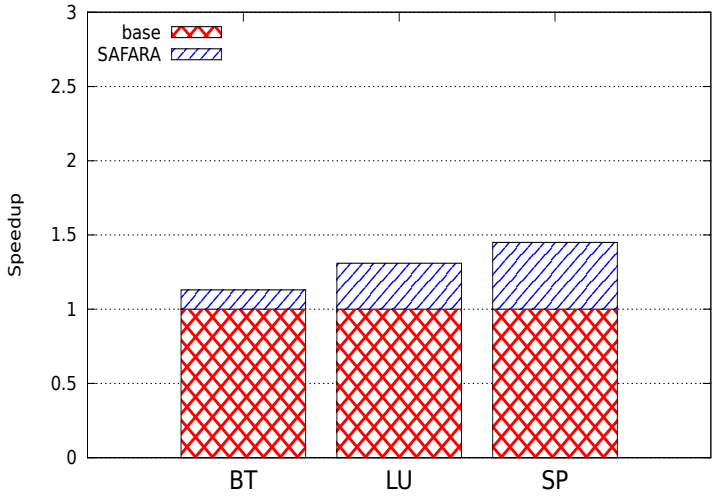


Figure 8.5: NAS BT, LU and SP Benchmarks Performance Improvement by Register Optimization on AMD APU 7850K

We also verified the register optimization on the AMD APU in Figure 8.5. However, we haven't fully supported the APU platform our OpenUH and it has some compilation error when building the SPEC benchmarks. Here, we only show BT, LU and SP from NAS benchmarks and leave the SPEC for the future work. Clearly, Computing capability from APU is much less powerful than discrete GPUs from Table 8.2. Another direction that we can try in the future is to combine the distributed programming models and OpenACC to increase the computing power.

In order to assess whether the `small` and `dim` clauses can effectively reduce the register usage when multiple allocatable arrays or VLAs are used in Fortran or C codes, we introduce another metric: the number of registers used in a kernel with and without the `small` and `dim` clauses. There are 15 kernels in 355.seismic and more than 40 kernels in 356.sp. We chose the 7 hottest kernels in seismic and the 10 hottest kernels in sp to perform this experiment. Note that 355.seismic and 356.sp

are two Fortran applications where allocatable arrays are used. In a compiler, there are multiple optimizations eager to use register files, such as kernel merging, loop unrolling, scalar replacement and memory vectorization [35]. The registers saved by the `small` and `dim` clauses can be used by these optimizations.

For 355.seismic, we take the 7 hottest kernels that constitute together 80% of the total execution time. Table 8.5 shows the register usage optimization results. The register files can be largely reduced by `small` and `dim` clauses if multiple allocatable arrays are used in the same kernel.

Table 8.5: 355.seismic register files usage improvement via `small` and `dim` clause

Kernels	Base	+`small`	w `dim`	Saved
HOT1	128	104	48	80
HOT2	134	105	41	93
HOT3	101	90	47	54
HOT4	90	78	44	46
HOT5	86	79	44	42
HOT6	88	77	40	48
HOT7	76	73	40	36

356.sp has 10 frequently used allocatable arrays with two different dimensional information. However, most of the kernels only use one of them (the ones with NA: `dim` was not used). We chose the 10 hottest kernels (based on the execution time) and investigated the register usage information. From Table 8.6, the register files are largely reduced in the three kernels that access multiple of these arrays. NA corresponds to kernels that use only zero, one allocatable array, or allocatable arrays that do not have equal dimensions; in this case, `dim` should not be used.

Table 8.6: 356.sp register files improvement by `small` and `dim` clause

Kernels	Base	+`small`	w `dim`	Saved
HOT1	72	67	NA	5
HOT2	70	54	51	19
HOT3	82	66	NA	16
HOT4	82	66	59	23
HOT5	74	37	32	42
HOT6	57	57	NA	0
HOT7	95	78	60	35
HOT8	211	152	112	99
HOT9	184	146	114	70
HOT10	60	58	NA	2

8.2.2 Performance Evaluation Using NVIDIA and AMD discrete GPUs

From the Table 8.2, W8100 and K20mc have similar computation ability. In this section, we compare the performance of SPEC (Figure 8.6) and NAS (Figure 8.7) benchmarks on AMD and NVIDIA GPUs with our OpenUH compiler.

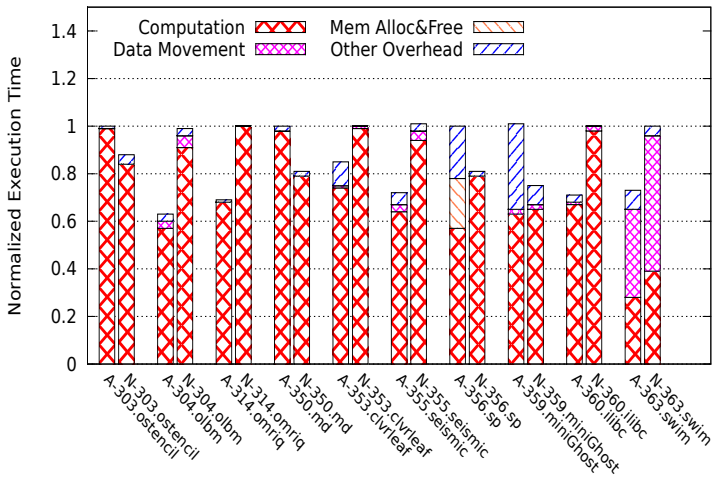


Figure 8.6: SPEC ACCEL SUITE Performance on AMD W8100 and NVIDIA K20mc: "A" is "AMD W8100" and "N" is "NVIDIA K20mc"

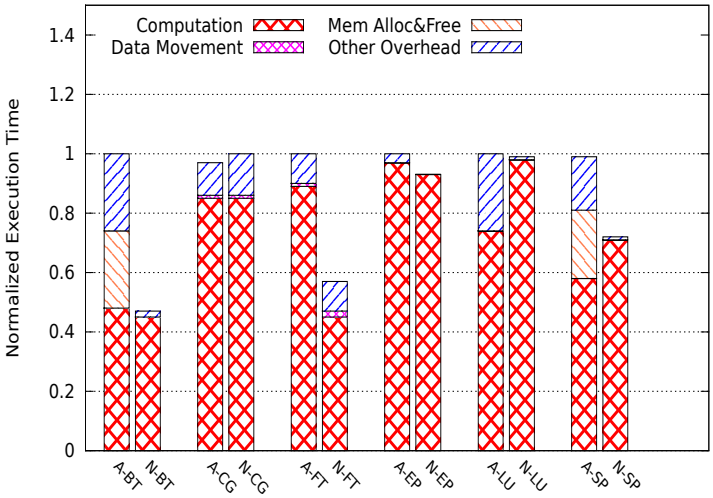


Figure 8.7: NAS OpenACC Benchmarks Performance on AMD W8100 and NVIDIA K20mc : "A" is "AMD W8100" and "N" is "NVIDIA K20mc"

MG from NAS benchmarks uses many subarrays in the offload regions and creating subarray in OpenCL is limited by memory alignment. The start of subarray has to be aligned to 2048bit boundary. CUDA however can create arbitrary size of subarray. As a result, this benchmark can run very efficiently on NVIDIA GPUs and fail on the AMD GPUs due to this OpenCL limitation. The result of MG is not included in the Figure 8.9.

In Figure 8.6 and Figure 8.7, the execution time is split into several parts:

1. Time invested during computation corresponds to execution of the kernel by the device
2. The time spent during data movement between the global and GPU memory
3. Time spent during device memory allocation and free management
4. Other overhead within the device driver

Since source-to-source translation is used within OpenUH for compiling GPU kernel functions and OpenACC runtime (over NVIDIA CUDA and AMD OpenCL driver), our approach may introduce additional overhead at the runtime. This overhead is almost the same for the NVIDIA and AMD since we use the same compiler framework for two platforms. The difference in performance can be attributed to the actual implementation of the CUDA and AMD OpenCL drivers.

From the above figures, we observe the following:

1. All the benchmarks except 363.swim from SPEC are kernel are not data movement intensive applications. It simply means most of the execution time should be spent on the kernel computing.

2. By comparing ratio of computation time against the entire execution time between AMD OpenCL and NVIDIA CUDA, the latter is more efficent. This is because the time invested in computation dominates the other factors on the NVIDIA platform. OpenCL runtime generates a huge overhead on many benchmarks on the AMD GPU. Espcially when the device memory malloc and free are frequently called. Such OpenCL runtime overhead can be removed if the compiler adopts source-to-end compilation path using the GPU low-level driver.

3. Computing time indicates the efficiency of GPU kernel functions. In most of cases, AMD GPU runs close to or even faster than NVIDIA if we only consider the computation time.

8.2.3 Performance Comparison between OpenUH and PGI

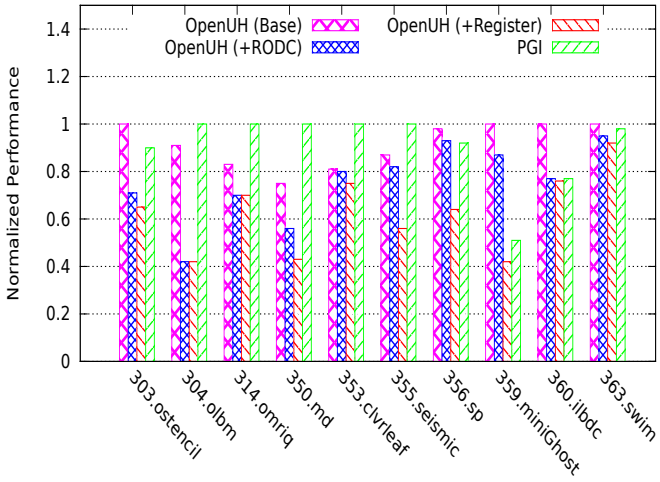


Figure 8.8: SPEC performance comparison between the OpenUH and PGI compilers. The execution time is normalized and the lower, the better

We also compare our implementation in OpenUH with the PGI compiler for both SPEC in Figure 8.8 and NAS in Figure 8.9. We present performance results for the OpenUH compiler with the following configurations:

1. Base version with data locality optimizations disabled
2. RODC enabled
3. Both the data locality optimizations enabled - RODC and register optimization (SAFARA and the two clauses - 'dim'+'small')

From Figure8.8 and Figure8.9, we observe that the OpenUH compiler generates efficient GPU kernels that outperform the PGI compiler while using the data locality optimizations.

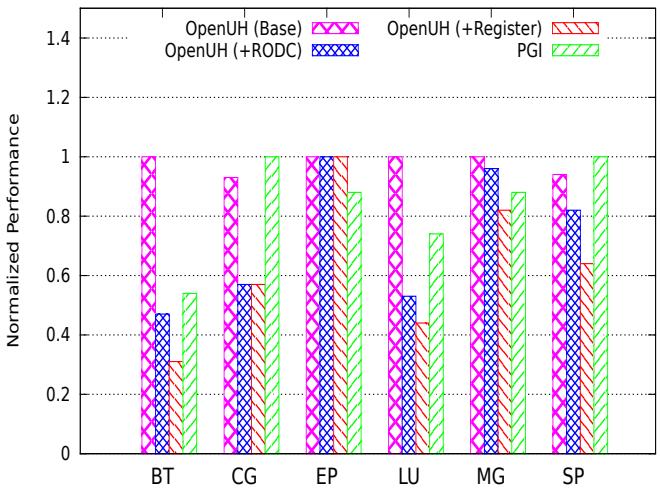


Figure 8.9: NAS performance comparison between the OpenUH and PGI compilers. The execution time is normalized and the lower, the better

Possible reasons for the PGI performance differing from OpenUH: One of the possible reasons is the execution mode in the offload region. OpenUH adopts our Redundant Execution as discussed in Chapter 6. While PGI uses the classical synchronization and broadcasting implementation which causes unnecessary overhead [68]. For EP from NAS benchmarks, the execution time is dominated by one reduction kernel in which the synchronization cannot be eliminated using the redundant execution mode.

8.3 Summary

In this chapter, we evaluated the proposed optimizations within the OpenACC implementation in OpenUH. For this, two sets of widely used benchmarks were used. The evaluation was presented in three parts. First, the impact on speedup due to

data locality optimizations was discussed. It was observed that the proposed optimizations enabled significant reduction in execution time. This was followed by a comparison of performance between the AMD and NVIDIA GPUs. It was empirically shown that the performance portability across two different vendors' platform is achievable using OpenACC. The final part described empirical results between our OpenUH and the state-of-the-art NVIDIA PGI compiler[4]. The results of the evaluation show that the implementation of OpenACC coupled with the proposed optimizations within OpenUH present a competitive edge in performance over that achievable by the PGI compiler.

[4]The PGI compiler has supported OpenACC since 2012

Chapter 9

Conclusion

9.1 Contributions

In this dissertation, we have described the entire framework for designing and optimizing an open source OpenACC compiler serving as research platform in the community. A number of innovative methods are adopt to efficiently generating GPU kernels. The contributions of this dissertation are summarized as follows:

1. We developed a robust and optimized open-source OpenACC compiler based on OpenUH, a branch of the Open64 compiler. The compiler takes C/Fortran applications and targets NVIDIA GPUs and AMD GPUs/APUs. This implementation serves as compiler infrastructure for researchers to explore advanced compiler techniques, to extend OpenACC to other programming languages, or to build performance tools used with OpenACC programs.

2. We designed a rich set of loop-scheduling strategies within the compiler to efficiently distribute kernels or parallel loops to the threading architectures of GPU accelerators. Within the loop scheduling transformation, an innovative redundant execution mode is proposed and implemented in order to reduce unnecessary synchronization and broadcasting overhead.

3. We presented compile-time data locality optimizations to exploit the deep memory hierarchies in GPUs. These optimizations include : 1) the compile-time read-only array/pointer detection for each offload region in order to utilize the read-only data cache; 2) an extension to the classical scalar replacement algorithm called SAFARA that is based on feedback information regarding register utilization and a memory latency-based cost model to select which array references should be replaced by scalar references. Moreover, since the aggressive application of scalar replacement increases register pressure, we proposed two new clauses to add to OpenACC, namely *dim* and *small*, to reduce the register usage.

4. SPEC and NAS OpenACC benchmarks are used to evaluate our compiler and optimizations. With the data locality optimizations, we got up to 3.22 speedup running NAS and 2.41 speedup while running SPEC benchmarks on NVIDIA Kepler GPU; We also achieved up to 1.29 speedup running NAS and 1.8 speedup while running SPEC benchmarks on AMD W8100. The results suggest that these approaches are effective for improving the overall performance of code executing on the GPU. We also compare the design of our compiler framework against NVIDIA PGI compiler. The results indicate that

our compiler generates more efficient code than PGI's compiler.

9.2 Future Work

During the course of this work, we developed a robust and performant implementation for OpenACC programming model within the OpenUH compiler. There are at least three research directions we intend to explore, based on the framework that we developed.

1. A cost model-based loop-scheduling autotuning at compile-time is required.In this dissertation, users have to specify the loop scheduling for each nested loop in order to reach peak performance. However, cost-model can be used to analyze the nested loop and choose a proper loop scheduling in order to achieve decent performance automatically.

2. Data locality optimization is key factor to take full advantage of the GPU memory hierarchy. For example, memory vectorization can be used to take advantage of high bandwidth memory on device. Scratchpad memory on GPU is high-bandwidth and low-latency onchip resource. It is as fast as L1 cache. Both of these two optimization can help reduce the memory access latency and thus improve the performance.

3. Multiple accelerators within the same node may become a trend, hence how to control the data affinity among the accelerators and host is challenging. A high-level abstraction of the data partition among accelerators and host is

necessary to ease the programmer's burden and leverage the communication overhead. Partitioned Global Address Space (PGAS) may be an ideal direction to try. PGAS assumes a global shared memory address space that is logically partitioned and portions of it are local to each processor (accelerator or host). The virtue of PGAS is that portions of the shared memory space may have an affinity for a particular processor. A high-level directive-based approach of the PGAS concept on multiple accelerators system can be an efficient solution to reduce the programming complexity and improve the performance in terms of the data affinity.

4. The component of our framework that targets on AMD GPUs still is not well optimized. There is still a room to work on several aspects : 1) since 8K Scalar register file is divided into 512 entries and is shared for each SIMD unit, the register files optimization to utilize the scalar register files is necessary; 2)the reduction algorithm in our compiler is designed for NVIDIA GPUs; it may not be good fit for AMD architecture.

Bibliography

[1] C++ Accelerated Massive Parallelism. https://msdn.microsoft.com/en-us/library/hh265137.aspx.

[2] CL Offline Compiler : Compile OpenCL kernels to HSAIL. https://github.com/HSAFoundation/CLOC.

[3] CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[4] GCC's implementation of the OpenACC specification and related functionality. https://gcc.gnu.org/wiki/OpenACC.

[5] How does OpenMP relate to OpenACC ? http://openmp.org/openmp-faq.html#OpenACC.

[6] Knights Corner is Your Path to Knight Landing. https://software.intel.com/sites/default/files/managed/e9/b5/Knights-Corner-is-your-path-to-Knights-Landing.pdf.

[7] Offloading Support in GCC. https://gcc.gnu.org/wiki/Offloading.

[8] OpenACC Standard Home. http://www.openacc-standard.org.

[9] OpenCL Standard. http://www.khronos.org/opencl.

[10] OpenMP GPU/Accelerators Coming of Age in Clang. http://llvm.org/devmtg/2015-10/slides/WongBataev-OpenMPGPUAcceleratorsComingOfAgeInClang.pdf.

[11] OpenMP offload infrastructure in LLVM. https://drive.google.com/file/d/0B-jX56_FbGKRM21sYlNYVnB4eFk/view.

[12] OpenUH-Open Source UH Compiler. https://github.com/uhhpctools/openuh.

[13] ORNL Summit Fact Sheet. https://www.olcf.ornl.gov/wp-content/uploads/2014/11/Summit_FactSheet.pdf.

[14] The HMPP Codelet Generator Directives. https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPPCG_Directives_ReferenceManual.pdf.

[15] The OpenMP API Specification for Parallel Programming. http://openmp.org/wp/.

[16] Cray C and C++ Reference Manual. http://docs.cray.com/books/S-2179-81/S-2179-81.pdf, 2014.

[17] Joel C. Adams. HPC: Computational Performance vs. Human Productivity. http://cacm.acm.org/blogs/blog-cacm/180881-hpc-computational-performance-vs-human-productivity/fulltext.

[18] Cody Addison, James LaGrone, Lei Huang, and Barbara Chapman. Openmp 3.0 tasking implementation in openuh. In *Open64 Workshop at CGO*, volume 2009, 2009.

[19] JosM. Andin, Manuel Arenaz, Franois Bodin, Gabriel Rodrguez, and Juan Tourio. Locality-aware automatic parallelization for gpgpu with openhmpp directives. *International Journal of Parallel Programming*, pages 1–24, 2015.

[20] Nastaran Baradaran and Pedro C Diniz. A register allocation algorithm in the presence of scalar replacement for fine-grain configurable architectures. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 6–11. IEEE, 2005.

[21] Mihai Budiu and Seth C Goldstein. Inter-iteration scalar replacement in the presence of conditional control-flow. Technical report, DTIC Document, 2004.

[22] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *ACM Sigplan Notices*, volume 25, pages 53–65. ACM, 1990.

[23] Steve Carr, Steve Carr, Steve Carr, Ken Kennedy, Ken Kennedy, and Ken Kennedy. Scalar replacement in the presence of conditional control flow. *Software Practice and Experience*, 24:51–77, 1992.

[24] B. Chapman, O. Hernandez, Lei Huang, Tien hsiung Weng, Zhenying Liu, L. Adhianto, and Yi Wen. Dragon: an open64-based interactive program analysis tool for large applications. In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 792–796, Aug 2003.

[25] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. Porple: An extensible optimizer for portable data placement on gpu. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 88–100. IEEE Computer Society, 2014.

[26] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). *SIGPLAN Not.*, 47(6):1–12, June 2012.

[27] Deepak Eachempati, Hyoung Joon Jun, and Barbara Chapman. An open-source compiler and runtime implementation for coarray fortran. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 13:1–13:8, New York, NY, USA, 2010. ACM.

[28] VinothKrishnan Elangovan, Rosa.M. Badia, and EduardAyguade Parra. Ompss-opencl programming model for heterogeneous systems. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, volume 7760 of *Lecture Notes in Computer Science*, pages 96–111. Springer Berlin Heidelberg, 2013.

[29] Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman. A prototype implementation of openmp task dependency support. In AlistairP. Rendell, BarbaraM. Chapman, and MatthiasS. Mller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 128–140. Springer Berlin Heidelberg, 2013.

[30] The Portland Group. PGI Accelerator Programming Model for Fortran and C (v1.3), 2010.

[31] 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.

[32] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki. Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 136–143, May 2013.

[33] Lei Huang, Haoqiang Jin, Liqi Yi, and Barbara Chapman. Enabling locality-aware computations in openmp. *Sci. Program.*, 18(3-4):169–181, August 2010.

[34] Lei Huang, Girija Sethuraman, and Barbara Chapman. Parallel data flow analysis for openmp programs. In Barbara Chapman, Weiming Zheng, GuangR. Gao, Mitsuhisa Sato, Eduard Ayguad, and Dongsheng Wang, editors, *A Practical Programming Model for the Multi-Core Era*, volume 4935 of *Lecture Notes in Computer Science*, pages 138–142. Springer Berlin Heidelberg, 2008.

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

[36] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):105–118, 2011.

[37] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W Hwu, et al. Spec accel: A standard application suite for measuring hardware accelerator performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 46–67. Springer, 2014.

[38] Mahmut Kandemir. Lods: Locality-oriented dynamic scheduling for on-chip multiprocessors. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 125–128, New York, NY, USA, 2004. ACM.

[39] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[40] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[41] Ahmad Lashgar and Amirali Baniasadi. Employing software-managed caches in openacc: Opportunities and benefits. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(1):2:1–2:34, February 2016.

[42] Ahmad Lashgar, Alireza Majidi, and Amirali Baniasadi. IPMACC: open source openacc to cuda/opencl translator. *CoRR*, abs/1412.1127, 2014.

[43] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society, 2010.

[44] S. Lee, D. Li, and J. 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, May 2014.

[45] Seyong Lee and Jeffrey S Vetter. Openarc: extensible openacc compiler framework for directive-based accelerator programming study. In *Proceedings of the First Workshop on Accelerator Programming using Directives*, pages 1–11. IEEE Press, 2014.

[46] Seyong Lee and Jeffrey S. Vetter. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 115–120, New York, NY, USA, 2014. ACM.

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

[48] Chunhua Liao, Yonghong Yan, Bronis R de Supinski, Daniel J Quinlan, and Barbara Chapman. Early Experiences with the OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators (IWOMP'13)*, pages 84–98. Springer, 2013.

[49] M. Nakao, J. Lee, T. Boku, and M. Sato. Productivity and performance of global-view programming with xcalablemp pgas language. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 402–409, May 2012.

[50] M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Boku, and M. Sato. Xcalableacc: Extension of xcalablemp pgas language

using openacc for accelerator clusters. In *Accelerator Programming using Directives (WACCPD), 2014 First Workshop on*, pages 27–36, Nov 2014.

[51] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[52] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

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

[54] Ruymán Reyes, Iván López-Rodríguez, Juan J Fumero, and Francisco de Sande. accull: An openacc implementation with cuda and opencl support. In *Euro-Par 2012 Parallel Processing*, pages 871–882. Springer, 2012.

[55] AVS Sastry and Roy DC Ju. A new algorithm for scalar register promotion based on ssa form. In *ACM SIGPLAN Notices*, volume 33, pages 15–25. ACM, 1998.

[56] Byoungro So and Mary Hall. Increasing the applicability of scalar replacement. In *Compiler Construction*, pages 185–201. Springer, 2004.

[57] Rishi Surendran, Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Inter-iteration scalar replacement using array ssa form. In *Compiler Construction*, pages 40–60. Springer, 2014.

[58] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. A source-to-source openacc compiler for cuda. In *Euro-Par 2013: Parallel Processing Workshops*, pages 178–187. Springer, 2014.

[59] Xiaonan Tian, Dounia Khaldi, Rengan Xu, and Barbara Chapman. Optimizing gpu register usage: Extensions to openacc and compiler optimizations. In *Accepted by the 45th International Conference on Parallel Processing (ICPP)*, Philadelphia, PA, USA, 2016. IEEE.

[60] 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. cpe.3648.

[61] Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara Chapman. Compiling A High-Level Directive-based Programming Model for Accelerators. In *LCPC 2013: The 26th International Workshop on Languages and Compilers for Parallel Computing*, 2013.

[62] Tristan Vanderbruggen and John Cavazos. Generating opencl c kernels from openacc. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, page 9. ACM, 2014.

[63] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC'10*, 2010.

[64] 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.

[65] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[66] 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 & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1169–1176. IEEE, 2013.

[67] 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.

[68] Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, Yonghong Yan, and Barbara Chapman. Nas parallel benchmarks for gpgpus using a directive-based programming model. In *Languages and Compilers for Parallel Computing*, pages 67–81. Springer, 2014.

[69] 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 10–20, New York, NY, USA, 2007. ACM.