Scalar-Vector GPU Architectures

A Dissertation Presented
by
Zhongliang Chen
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in
Computer Engineering
Northeastern University
Boston, Massachusetts
December 2016
To my family and friends for their infinite support
Contents

List of Figures v
List of Tables vii
List of Acronyms viii
Acknowledgments ix
Abstract of the Dissertation x

1 Introduction 1

2 Background 5
  2.1 GPU Programming Models .......................................................... 5
    2.1.1 Compute Unified Device Architecture (CUDA) .......................... 6
    2.1.2 Open Computing Language (OpenCL) ........................................ 9
    2.1.3 Other Popular GPU Programming Models ................................. 10
  2.2 GPU Architecture ................................................................. 11

3 Related Work 15
  3.1 Related Work in Workload Characterization ................................. 15
  3.2 Related Work in Architectural Optimization .................................. 16
  3.3 Related Work in Prefetching for GPU Compute Applications ............. 17

4 Workload Characterization 18
  4.1 Static Scalar Analysis ............................................................. 18
  4.2 Dynamic Scalar Analysis ......................................................... 23

5 Scalar-Vector GPGPU Architecture 29
  5.1 Scalar-Vector Multiprocessor Architecture ................................. 29
  5.2 Instruction Dispatch Optimizations ............................................. 32
  5.3 Warp Scheduling Optimizations ................................................ 35
  5.4 Subwarp Execution ................................................................. 36
  5.5 Evaluation ........................................................................... 37
6 Scalar-Assisted Prefetching 45
   6.1 Prefetching Opportunities .................................................. 45
       6.1.1 Regular vs. Irregular Memory Access Patterns .......................... 45
       6.1.2 Scalar Unit Utilization .................................................. 47
   6.2 Prefetching Mechanism ......................................................... 47
       6.2.1 Warp Scheduling Optimization ......................................... 49
   6.3 Evaluation ............................................................................. 50

7 Conclusion 54
   7.1 Future Research Directions ..................................................... 54

Bibliography 56
List of Figures

1.1 Motivating analysis: comparison of the number of dynamic scalar versus vector instructions in the workloads studied in this work. .................................................. 2

2.1 CUDA programming model [1]. ............................................................. 7

2.2 OpenCL programming model [2]. ......................................................... 9

2.3 NVIDIA Kepler GK110 GPU architecture [3]. .................................... 12

2.4 AMD Graphics Core Next compute unit architecture [4]. ...................... 14

4.1 Details of a CUDA vector addition kernel and the resulting PTX code. The annotation “1” or “0” at the beginning of each PTX instruction denotes a scalar and vector instruction, respectively. .................................................. 19

4.2 An example of scalar opportunity analysis (vector addition). .................. 21

4.3 Percentage of static scalar opportunities. ............................................. 22

4.4 The percentage of dynamic scalar instructions. .................................... 25

4.5 The breakdown of scalar instructions. .................................................. 26

4.6 The breakdown of vector instructions. .................................................. 26

4.7 The percentage of scalar instructions varies during execution. .............. 27

5.1 The scalar-vector multiprocessor architecture. The shaded and hatched boxes represent the new or modified hardware components, respectively. ......................... 30

5.2 An example of the modified operand collector architecture. “vr” and “sr” represent vector and scalar register files, respectively. “-” denotes that the value is not checked by the dispatcher. .................................................. 32

5.3 An example showing the performance improvement with the optimized warp scheduling policy. “s” and “v” denote a scalar and vector instructions executing on the corresponding unit, respectively. The hatched area indicates that there is no instruction available to issue due to a lack of warps. .................................................. 33

5.4 The number of scalar instruction increases with smaller warp sizes. ....... 36

5.5 Overview of general heterogeneous GPU architecture implemented in GPGPU-Sim. .................................................. 38

5.6 The utilization of the vector units on the baseline versus the scalar-vector architecture. .................................................. 40

5.7 The utilization of the scalar and vector units. ....................................... 40

5.8 The percent increase in the effective issue width relative to the baseline. .... 41

5.9 The performance improvements of the scalar-vector GPU architecture. ............................................................................ 42

5.10 The power improvements of the scalar-vector GPU architecture. .......... 43
List of Tables

4.1 The benchmarks selected from Parboil [5], Rodinia [6, 7], and CUDA Samples [8], providing a rich range of behaviors for this study. .......................................................... 24

5.1 The simulator configurations used in this work. Our implementation has been calibrated against an NVIDIA Kepler K40 GPU. .......................................................... 37
List of Acronyms

**GPU** Graphics Processing Unit. A specialized computer chip to accelerate image manipulation and display.

**GPGPU** General Purpose computing on Graphics Processing Unit. Use of a GPU to perform general purpose computation traditionally handled by a central processing unit.

**SIMD** Single Instruction Multiple Data. A class of parallel computers in Flynn’s taxonomy. It has multiple processing elements and can perform the same operation on multiple data items simultaneously.
Acknowledgments

The completion of my Ph.D. has been a rewarding journey. Over the past six years, I have received tremendous support from a great number of people. I would like to express my gratitude to all of them for their generous help.

First and foremost, I would like to thank my advisor, Dr. David R. Kaeli, for providing the excellent guidance for my research. He has always been able to offer me invaluable advice whenever I need help. His optimism and diligence have helped to strengthen myself significantly. I would also like to thank Dr. Norman Rubin, who as my mentor at AMD, motivated me to explore the scalar opportunities in GPU compute applications. His wisdom and vision have helped me to build the foundation of this thesis. Also thank you to Dr. Gunar Schirner for being a member of my dissertation committee. He has given very constructive comments, which enable me to greatly improve the quality of this thesis.

I would like to especially thank my fellow students in the Northeastern University Computer Architecture Research group, including Yash Ukidave, Xiang Gong, Yifan Sun, Amir Kavyan Ziabari and Fanny Nina-Paravecino, Xiangyu Li, Shi Dong, Perhaad Mistry, and Dana Schaa. They have been often sharing with me their insightful thoughts about various research topics. Several ideas in this thesis originate from our discussions. They have also provided generous help in my experiments, data analysis, writing, and even life.

Last but not least, I would like to thank my wife for her encouragement and understanding during my Ph.D. study. I am not able to finish it without her support.
Abstract of the Dissertation

Scalar-Vector GPU Architectures

by

Zhongliang Chen

Doctor of Philosophy in Computer Engineering
Northeastern University, December 2016
Dr. David Kaeli, Adviser

Graphics Processing Units (GPUs) have evolved to become high throughput processors for general purpose data-parallel applications. Most GPU execution exploits a Single Instruction Multiple Data (SIMD) model, where a single operation is performed on multiple data at a time. However, neither runtime or hardware pays attention to whether the data components on SIMD lanes are the same or different. When a SIMD unit operates on multiple copies of the same data, redundant computations are generated. The inefficient execution can degrade performance and deteriorate power efficiency.

A significant number of SIMD instructions in GPU compute programs demonstrate scalar characteristics, i.e., they operate on the same data across their active lanes. Treating them as normal SIMD instructions results in inefficient GPU execution. To better serve both scalar and vector operations, we propose a heterogeneous scalar-vector GPU architecture. In this thesis we propose the design of a specialized scalar pipeline to handle scalar instructions efficiently with only a single copy of the data, freeing the SIMD pipeline for normal vector execution. The proposed architecture provides an opportunity to save power by just broadcasting the results of a single computation to multiple outputs. In order to balance scalar and vector units, we propose novel schemes to efficiently resolve scalar-vector data dependencies, schedule warps, and dispatch instructions. Also, we consider the impact of varying warp sizes on our scalar-vector architecture and explore subwarp execution for power efficiency. Finally, we demonstrate that the interconnect and memory subsystem can be the new limiting factor on scalar-vector execution.
Chapter 1

Introduction

Since the early 2000s, single-core Central Processing Units (CPUs) have faced increasing challenges in delivering higher performance because of the power wall and limited instruction-level parallelism in applications. Various parallel computer architecture advances such as multi-core CPUs and many-core Graphics Processing Units (GPUs) have emerged as promising paths forward. Parallel computer architectures can be categorized based on the numbers of instruction streams and data streams, as known as Flynn’s Taxonomy [9].

- SISD: Single Instruction, Single Data
- SIMD: Single Instruction, Multiple Data
- MISD: Multiple Instruction, Single Data
- MIMD: MUltiple Instruction, Multiple Data

A conventional uniprocessor has a single instruction stream and a single data stream, and thus it is a Single Instruction Single Data (SISD) computer. A general computer cluster has multiple instruction streams and multiple data streams, so it is a Multiple Instruction Multiple Data (MIMD) computer. A Multiple Instruction Single Data (MISD) computer would perform multiple instructions on a single data stream, although there is no such examples at the present time. On the contrary, a vector processor such as a GPU is a Single Instruction Multiple Data (SIMD) computer, where a single instruction stream is executed on multiple data streams.

Graphics Processing Units (GPUs) are becoming an increasingly attractive platform for general purpose data-parallel applications. With programmable graphics pipeline and software runtime frameworks such as CUDA [1] and OpenCL [10], GPUs have been applied to many problems.
CHAPTER 1. INTRODUCTION

traditionally solved on general-purpose vector processors [11]. They can deliver higher performance through explicit hardware parallelism. Unlike CPUs, GPUs employ most of the on-chip real estate for computation rather than control logic or cache. Based on a massive parallel architecture, GPUs can be effectively used as many-core processors capable of high compute throughput and memory bandwidth.

The GPU’s SIMD execution model allows multiple processing elements to perform the same operation on multiple data, concurrently. Threads are mapped to SIMD lanes, no matter what the data values are for the source operands. When the input operands to a SIMD instruction are all the same, the resulting execution is redundant and inefficient. In such cases, the SIMD operation can be reduced to Single Instruction Single Data (SISD) execution. We refer to this situation as a scalar instruction. When the operands differ, we use a vector instruction. The point is that we should avoid executing a vector instruction if we can instead use a scalar instruction.

Figure 1.1 shows that scalar instructions account for more than 40% of all dynamically executed instructions across a wide range of GPU compute applications. The applications used in the analysis are listed in Table 4.1. If we continue to use the SIMD/vector hardware for these SISD operations, we are wasting resources and burning unnecessary power.

To address this problem, we propose both compiler-based and microarchitecture-based
CHAPTER 1. INTRODUCTION

scalarness detection mechanisms, delivering a new class of heterogeneous scalar-vector GPU architectures. These approaches can work together to allow more efficient GPU execution without any modification to an application. The scalarness-aware compiler optimization can detect and annotate scalar instructions at compile time. Then the hardware will use the annotations for efficient execution at run time. Alternatively, the microarchitecture can detect and then execute scalar instructions without the compiler’s assistance. The scalar-vector GPU architecture enables scalar instructions to be executed on scalar units so that the vector engines can be used to execute vector operations. Our design and implementation are based on a general SIMD architecture and are not limited to specific NVIDIA or AMD GPUs.

We face the following challenges in designing scalar-vector GPU architectures.

- The scalar instruction detection can be performed at compile time or at run time. Compiler-based approaches have zero hardware overhead but can be too conservative. On the contrary, hardware-based approaches can have higher detection rate but suffer from hardware modifications. The tradeoff should be examined carefully.

- The scalar unit cannot handle all types of scalar instructions in order to keep the hardware complexity low. It is necessary to analyze the types of scalar instructions.

- The scalar and vector execution have to be balanced when there exist enough scalar and vector instructions; neither is under or over utilized.

- When the scalar unit is not beneficial, it should be turned off to save power. This can occur for several reasons, e.g., the number of scalar instructions is limited.

- The warp size is a key design parameter that can affect the ratio of scalar to vector instructions. Its effects should be investigated in depth, which can help in microarchitectural decision-making.

To overcome the challenges above, this thesis will perform a thorough study of scalar and vector execution on GPU architectures and make the following contributions.

- We characterized a wide range of GPU compute benchmarks both statically and dynamically. The information is used to guide our microarchitecture design. We show that GPU compute applications typically have a mix of scalar and vector instructions.

- We proposed a scalarness-aware compiler optimization and analyzed quantitatively the effectiveness of scalar instruction identification by compilers. We also proposed a hardware-based
CHAPTER 1. INTRODUCTION

scalar instruction detection mechanism, which will be in use when the compiler approach is too conservative.

- We designed a scalar-vector GPU architecture and examined the opportunities and challenges for various design alternatives on different microarchitectural components.
  - We designed the scalar unit to handle the most popular types of scalar instructions with modest hardware complexity.
  - We redesigned the instruction dispatcher and warp scheduler to balance the execution of scalar and vector instructions.
  - We evaluated the power saving of the scalar-vector GPU architectures compared to the traditional vector-only architectures.
  - Our scalar-vector architecture requires a rethinking of key design parameters in traditional GPUs. The warp size is a good example. We explored the design space and proposed a dynamic warp sizing scheme to better serve both scalar and vector execution.

Our architecture is capable of effectively utilizing onboard scalar units to service scalar instructions at the lowest cost.

This thesis proposal is organized as follows. Chapter 2 provides an overview of modern GPU programming models and architectures. Chapter 3 reviews the related work in the literature. Chapter 4 analyzes the scalar opportunities in GPU compute applications and motivates the potential for supporting scalar/vector instructions in GPU architectures. Chapter 5 describes our first scalar-vector GPU architecture and the implementation details, and discusses our experimental methodology and simulation results. Finally Chapter ?? summarizes the work completed to date, and lays our plan for the completion of this thesis.
Chapter 2

Background

GPUs are designed to be throughput-oriented processors. They serve as a complement to CPUs which excel at executing latency-sensitive applications. Traditional GPUs have a fixed-function graphics pipeline and are mostly used for efficient graphics manipulation. The introduction of unified graphics and computing architecture [12, 13, 14] has enabled modern GPUs to work well as programmable data-parallel processors which can be used for accelerating general purpose applications. Through the rest of this thesis we will focus on GPUs serving as compute devices. In this chapter, we will present the most common GPU programming models and the typical GPU architecture used today in practice.

2.1 GPU Programming Models

GPU computing uses GPUs to accelerate compute-intensive portions of applications. Programmers can offload data-parallel computations to GPUs, while executing the rest of the code on CPUs. Since GPUs have much higher throughput, computations are executed efficiently and faster than CPUs. The overall performance is therefore improved.

To program GPUs, there are three approaches commonly used: (1) GPU-accelerated libraries, (2) compiler directives, and (3) programming languages such as CUDA and OpenCL. GPU-accelerated libraries, such as cuBLAS [15], provide for ease of programming, requiring the programmer to have little knowledge of GPU programming. Users only need to use existing GPU-accelerated library calls, substituting serial functions with GPU-targeted parallel codes. However, the resulting application performance varies since users cannot guarantee how well the libraries will perform. Programmers will not have access to the source code of the library for analysis or
CHAPTER 2. BACKGROUND

tuning. Compiler directives, such as OpenACC [16] and OpenMP [17], allow programmers to insert hints or syntax into the code so that the compiler can generate parallelized code automatically. These runtimes can provide more flexibility than libraries since the programmer can access GPUs through specific directives. When a programmer can tune the code, application performance can be must faster than using GPU libraries only. Programming languages such as CUDA [1, 18, 19] and OpenCL [10, 20, 21] are the most flexible way to program a GPU, while also pursuing maximum performance. Programmers have complete access to the GPU, and are able to fine-tune the performance at lower code levels. In our preliminary evaluation, we have selected benchmarks written in CUDA since CUDA is more widely used than OpenCL. While we use CUDA, we claim that our proposed architectures are independent of GPU programming models, and can be applied to programs developed in OpenCL and most other GPU programming languages.

CUDA [1, 18, 19] and OpenCL [10, 20, 21] are two major programming languages available on GPUs, both using data-parallel programming models. In the following sections we will review them in detail, as well as discuss a few other popular programming models.

2.1.1 Compute Unified Device Architecture (CUDA)

NVIDIA’s CUDA is designed around three key abstractions [1]: 1) a hierarchy of thread blocks, 2) shared memories, and 3) barrier synchronization, which are shown in Figure 2.1. These system abstractions guide the programmer to partition a problem into multiple subproblems that can be solved independently in parallel by blocks of threads. Each subproblem can be further subdivided into finer elements that can be solved cooperatively in parallel by all threads within the block.

A CUDA program usually has two parts: 1) host code running on the CPU and 2) device code running on the GPU. A host is the system on which the CUDA library runs, e.g., x86 CPUs. A device is the processor that can communicate with the library, which can only be a single NVIDIA GPU.

A function in the device code is called a kernel, which is defined with the __global__ declaration specifier. When called, a kernel is executed multiple times in parallel by different threads, as opposed to only once as with a typical CPU function. CUDA creates a new execution configuration syntax <<<...>> to specify the number of threads executing a kernel. Each thread has a unique thread ID in the thread block that can be obtained in the kernel through the built-in threadIdx variable. Similarly, each thread block has an ID that is accessible through the blockIdx variable. Multiple thread blocks form a grid, which represents the whole computation domain.
CHAPTER 2. BACKGROUND

Figure 2.1: CUDA programming model [1].
CHAPTER 2. BACKGROUND

Thread blocks are required to execute independently, i.e., they are not assumed to execute in any predetermined order. This requirement allows thread blocks to be scheduled in any order and executed on any number of cores, thus enabling scalability.

CUDA has multiple memory spaces with different performance and capacity. Each thread has its own private memory, which has the highest bandwidth but the smallest size. Each thread block has on-chip high-bandwidth shared memory which can be used for efficient inter-thread communication in the block. The GPU device has off-chip global memory accessible by all threads. There are also several specialized memory spaces, including (1) read-only constant memory and (2) read-only texture memory. They are optimized for different memory usages and can be read by all threads.

The threads in a thread block can be synchronized by using the \texttt{__syncthreads()} intrinsic function. This is a barrier where no thread can proceed unless all the threads in the block have arrived. Synchronization is commonly used to ensure a desired ordering between computation and memory accesses.

2.1.1.1 Compilation Workflow

Kernels must be compiled into binary code by the NVIDIA CUDA C Compiler (NVCC) [22] before execution on the GPU. NVCC can perform offline or just-in-time compilation of CUDA C or intermediate PTX (Parallel Thread eXecution) [23] code.

For offline compilation, NVCC separately extracts host code and device code first. Then it compiles the device code into PTX intermediate code in the front-end, and then compiles the PTX code into native SASS ISA code [24], producing the final binary code through NVIDIA’s proprietary Optimized Code Generator [25]. The binary code is merged into a device code descriptor, which is included in the host code. This descriptor will be inspected by the CUDA runtime system whenever the device code is invoked by the host code. NVCC identifies the \texttt{<<<...>>>} syntax in the host code and replaces it with the appropriate runtime library calls to launch kernels.

Just-in-time compilation allows PTX code to be compiled into binary code at run time by the device driver. While it increases load time, the application may benefit from the compiler optimizations performed in the driver.
CHAPTER 2. BACKGROUND

2.1.2 Open Computing Language (OpenCL)

OpenCL [10] is a heterogeneous programming framework, which is managed by the Khronos Group. It is an open standard for general purpose parallel programming across CPUs, GPUs, FPGAs, and other devices, giving programmers a portable language to target a range of heterogeneous processing platforms. Similar to CUDA, OpenCL defines both a device-side language and a set of host-side APIs for device management.

The core of OpenCL defines the platform model, the execution model, the memory model, and the kernel programming model [2]. A platform consists of a host connected to one or multiple OpenCL devices. The host code submits commands (e.g., transferring data, launching a kernel, or host-device synchronization) to OpenCL devices for execution. The execution model includes host code executing on the host and kernels executing on one or more devices. The host manages application-specific OpenCL objects, including devices, kernel objects, program objects, and memory objects, in an OpenCL context. The work of a kernel is carried out by work-groups, which are further divided into work-items. The OpenCL memory model describes memory-related objects and their behaviors. For example, memory regions, memory objects, shared virtual memory, and consistency model are defined in OpenCL 2.0. The kernel programming model defines how parallel computation is mapped to OpenCL devices. OpenCL supports both data-parallel and task-parallel work. Synchronization can be performed between the work-items in a work-group and/or between host commands in a context.
CHAPTER 2. BACKGROUND

OpenCL defines a scalable thread structure for massive parallel computation. Each instance of a kernel is executed by a work-item. A work-group consists of a customizable number of work-items. Work-groups are independent of each other, which results in scalability. All work-groups of a kernel form an index space. Programmers can use built-in function calls to obtain a work-item’s ID, a work-group’s ID, etc., in a kernel.

OpenCL defines various types of memories. The most common types include private memory, local memory, global memory, and constant memory. Private memory is privately accessible by a work-item. Local memory can be accessed only by all the work-items in a work-group. Global memory and read-only constant memory are shared by all work-items. Vendors determine the mapping between memories and underlying hardware components. Generally, private memory is mapped to register files, which have the highest bandwidth and lowest latency among all the memories. Private memory is limited in size. Local memory is usually mapped to on-chip memory, which is larger in size and provides higher bandwidth than global memory. Global memory is off-chip, so it can be large (up to several GBs), but it is slow.

OpenCL accepts kernel source code as a string or binary code. Kernels are compiled at run time if provided as source code. They are first compiled into an Intermediate Representation (IR), such as NVIDIA PTX or AMD IL, and then further compiled into device-specific binary code with the back ends provided by the vendor/implementation. Once compilation is complete, the kernel binary can be launched on the device.

2.1.3 Other Popular GPU Programming Models

2.1.3.1 C++ Accelerated Massive Parallelism (C++ AMP)

The Microsoft C++ AMP (Accelerated Massive Parallelism) library [26], built on top of DirectX 11, enables data-parallel programming in C++. It introduced the restrict(amp) feature to allow programmers to declare that a C++ function or lambdas can be executed on an accelerator. It also provides new classes and APIs to manage the accelerator execution.

C++ AMP shares similar core ideas with CUDA and OpenCL. It models GPU data as multi-dimensional arrays and GPU kernels as lambda functions. A compute domain is defined as the work of a kernel. It is divided into tiles and each tile is further divided into threads created for parallel execution. The threads in a tile can be synchronized using barriers.
CHAPTER 2. BACKGROUND

2.1.3.2 Intel Xeon Phi Programming Models

Intel Xeon Phi is an x86-based many-core coprocessor. To support data-parallel execution, it provides two programming models: (1) a native execution model and (2) an offload model [27, 28]. The native model treats the coprocessor as a standalone multicore processor. It allows cross-compiling C++ or Fortran programs for the coprocessor with very few code changes. However, native execution comes with some constraints on the memory footprint and I/O. The offload model allows programmers to offload highly-parallel work using pragmas from the host processor to the coprocessor. The offload model is more flexible and allows finer performance tuning.

We will use CUDA as an example in this thesis, but our work is compatible with the other languages and/or runtime environments discussed.

2.2 GPU Architecture

A CPU typically consists of a few large and complex cores that are designed for fast serial execution. On the contrary, a GPU has thousands of smaller and efficient cores for massive data-parallel applications. For example, Intel’s Skylake i7-6700K processors have four cores and eight hardware threads [29], while NVIDIA’s Tesla K40 GPUs have 2880 cores [30].

Conventional CPUs are optimized to minimize the latency of a single thread. They use multi-level caches to reduce memory access latency, and utilize complicated control logic to reorder execution, provide ILP (Instruction-Level Parallelism), and minimize pipeline stalls. A large amount of CPU chip area is devoted to caching and control logic. Typically, CPUs have a small number of active threads and thus a limited number of registers.

Unlike CPUs, modern GPUs use a massively parallel execution model to achieve high throughput. Most of the device real estate on a GPU is dedicated to computation rather than caches or control logic [12]. They are organized as an array of independent cores, which are called Streaming Multiprocessors by NVIDIA or Compute Units by AMD. The L2 caches and main memory are generally heavily banked, which allows multiple simultaneous accesses, and provide high bandwidth (100s of GB/s). As an example, Intel’s Skylake i7-6700K processors have a 34 GB/s memory bandwidth [29], while NVIDIA’s Tesla K40 GPUs have 288 GB/s [30].

A GPU core consists of (1) SIMD hardware with a typical width of 16 or 32 elements and (2) memory components including a register file, an L1 cache and a user-managed scratchpad. The register file is much larger on a GPU versus a CPU, since it is used for fast context switching.
CHAPTER 2. BACKGROUND

There is no saving or restoring of the architectural state, and thread data is persistent during the entire execution. The user-managed scratchpad (called Shared Memory by NVIDIA and Local Data Store by AMD) is used for fast communication between the threads in a thread block. GPU scratchpad memory is heavily banked, and provides very high bandwidth of more than 1 TB/s.

![NVIDIA Kepler GK110 GPU architecture](image)

Figure 2.3: NVIDIA Kepler GK110 GPU architecture [3].

As an example, NVIDIA’s Kepler GK110 GPU architecture features up to 15 SMX units, each of which has 192 single-precision CUDA cores, 64 double-precision units, 32 load/store units, and 32 special function units [3], as shown in Figure 2.3. Each CUDA core has fully-pipelined integer and floating-point arithmetic logic units (ALUs) and can execute an integer or floating-point instruction per clock for each thread. Each load/store unit allows source and destination addresses to be calculated per thread per clock. Each special function unit can execute a transcendental instruction, such as sine, cosine, reciprocal, or square root per thread per clock [3]. A GK110 GPU can provide up to 4.29 Tflops single-precision and 1.43 Tflops double-precision floating point performance [30].

When a GPU kernel is executed, software threads created by the programmer are automatically grouped into hardware threads called warps by NVIDIA or wavefronts by AMD. A
warp/wavefront consists of 32/64 threads and executes a single instruction stream on the SIMD hardware. An instruction can be issued multiple times on 16-lane SIMD units. The threads in a warp/wavefront execute in a lock-step fashion.

The SIMD execution model requires that all the threads of a warp/wavefront always execute the same instruction at any given time, i.e., they cannot diverge. If they have to branch to different execution paths in a program, the thread/branch/warp divergence problem occurs. To address thread divergence in a warp/wavefront, a GPU uses an active mask (defined as a bit map) to indicate whether an individual thread is active or not. If a thread is active, its computation (and results) are committed in the updated microarchitectural state. Otherwise, if a thread is inactive, the results are simply masked and discarded.

As mentioned earlier, all the threads executing a kernel are organized as thread blocks, and each block has one or multiple warps. To schedule a large number of blocks of threads running in parallel, GPUs use (1) the global scheduler to distribute thread blocks to GPU cores, and (2) the local/warp schedulers in each GPU core to determine the execution order of the warps generated from the blocks on the core. For example, each GK110 GPU core has four warp schedulers, allowing four warps to be issued and executed concurrently. Then each instruction dispatcher selects the instruction at the current PC from a warp chosen to execute, and sends it to the corresponding pipeline. A GK110 GPU core has eight instruction dispatch units and up to two independent instructions from each warp can be dispatched per cycle. The design can exploit intra-warp ILP in addition to TLP.

Current GPUs that are based on the SIMD execution model can provide high throughput; however, they are not designed to handle control flow instructions efficiently (e.g., conditional branches). The execution of control flow instructions on SIMD hardware usually incur long latencies, which can become a serious bottleneck in some applications. Meanwhile, many control flow instructions are uniform, i.e., all the threads in a warp perform the same evaluation with the same data and have the same branch outcome. So the execution on each SIMD lane is the same and can be reduced to SISD.

In order to improve the performance of control flow instructions and reduce redundant execution, AMD integrates a new scalar unit into its GCN compute unit architecture [4, 31], as shown in Figure 2.4. Unlike standard SIMD units, the scalar units provide fast and efficient SISD execution. With the scalar unit, control flow processing is more efficient and the outcome is distributed throughout the shader cores, which can effectively reduce latency [4]. Furthermore, SIMD units can execute other SIMD instructions concurrently as scalar units execute SISD operations. Scalar execution improves the resource utilization of GPU cores.
Figure 2.4: AMD Graphics Core Next compute unit architecture [4].
Chapter 3

Related Work

Our study of scalar and vector execution on GPU architectures involves two main areas: compiler-assisted workload characterization and microarchitectural optimization. We will present related/previous work in these two areas.

3.1 Related Work in Workload Characterization

Previous work has focused on static (i.e., compile time) detection of divergence in GPU compute applications [32, 33]. Coutinho et al. [32] proposed variable divergence analysis and optimization algorithms. They introduced a static analysis to determine which vector variables in a program have the same values for every processing element. Also, they described a new compiler optimization that identifies, via a gene sequencing algorithm, chains of similarities between divergent program paths, and weaves these paths together as much as possible.

Stratton et al. [33] described a microthreading approach to efficiently compile fine-grained single-program multiple-data threaded programs for multicore CPUs. Their variance analysis discovers what portions of the code produce the same value for all threads and is then used to remove redundancy in both computation and data storage.

Related studies have also pursued compile-time identification of scalar instructions or regular patterns between adjacent threads in GPU applications [4, 34, 35]. The AMD shader compiler can detect and then optimize scalar/vector instruction statically and then generates GCN ISA code consisting of scalar and vector instructions [4]. The compiler can detect several uses of scalar instructions.
CHAPTER 3. RELATED WORK

Lee et al. [34] proposed a scalarizing compiler that factors scalar operations out of the SIMD code. They employed convergence and variance analysis to statically identify scalar values and instructions. Their compiler can identify two-thirds of the dynamic scalar opportunities, which leads to a reduction in instructions dispatched, register accesses, memory address generation and data accesses.

Collange [35] proposed a mechanism to identify scalar behaviors in CUDA kernels. His work described a compiler analysis pass to identify statically several kinds of regular patterns that can occur between adjacent threads, including common computations, memory accesses to consecutive memory locations and uniform control flow.

Leveraging the compiler removes the need for extra hardware that would be required to perform dynamic detection at the microarchitecture level. However, the compiler has to conservatively consider data dependencies present in all possible control flow paths, and as a result, only generates a limited number of scalar instructions. Also, without runtime information, the compiler might not be able to detect most scalar instructions. If many instructions in an application use input-dependent variables, the compiler will not likely to work well. So in this thesis we also study mechanisms used for the detection of scalar instructions at run time.

Previous work has also been done on dynamic detection of regular patterns between adjacent threads in GPU applications. Collange et al. [36] presented a dynamic detection technique for uniform and affine vectors in GPU computations. They concentrate on two forms of value locality in vector variables. The first form corresponds to the uniform patterns present when computing conditions. The second form corresponds to the affine patterns used to access memory efficiently.

We consider both static and dynamic scalar analysis and will carefully compare them in Chapter 4. We will provide a thorough analysis and discussion of the advantages and disadvantages of each approach. Also, we do not identify other patterns at this time since they are not frequent enough to justify additional costs.

3.2 Related Work in Architectural Optimization

Prior studies have proposed taking advantage of scalar instructions in applications to optimize GPU architectures [4, 37, 38, 39, 40, 36]. Most prior work has proposed adding scalar units into the architecture.

AMD’s Graphics Core Next (GCN) architecture [4] adds a scalar coprocessor into each compute unit. The scalar coprocessor has a fully-functional integer ALU, with independent instruction
CHAPTER 3. RELATED WORK

arbitration and decode logic, along with a scalar register file [31]. It supports a subset of the ISA and can execute scalar instructions detected by the compiler.

Yilmazer et al. [37] presented a scalar waving mechanism to batch process scalar instructions as a group on SIMD lanes. This approach improved the utilization of SIMD lanes by executing scalar instructions on them. However, the execution of scalar instructions is still on long-latency SIMD lanes, so their architecture did not handle latency-sensitive operations efficiently.

Yang et al. [38] proposed a scalar unit architecture to perform data prefetching and divergence elimination for SIMD units. However, the scalar units and SIMD units did not work together in their architecture, which can result in the latency and throughput requirements not to met.

Xiang et al. [39] modified the register file to support intra-warp and inter-warp uniform vector instructions. Their design did not include the other hardware components. The instruction dispatching and warp scheduling policies were not considered.

Gilani et al. [40] presented a scalar unit design consisting of a scalar register file and an FMA unit to remove computational redundancy. Given the limited hardwware devoted to scalar operations in their design, it may not work well if applications have a large number of non-FMA scalar instructions.

Some other prior work has exploited scalar instructions without the need to introduce scalar units [41, 36]. Narasiman et al. [41] proposed forming larger warps and dynamically creating subwarps in order to alleviate branch divergence penalty. They used existing architectures to realize their sophisticated warp formation schemes. Their focus was mainly to improve branch divergence.

Collange et al. [36] proposed an architecture to take advantage of two forms of value locality to significantly reduce the power required for data transfers between the register file and the functional units. They also looked at how to reduce the power drawn by the SIMD arithmetic units. Their work focused on variables rather than computations.

3.3 Related Work in Prefetching for GPU Compute Applications
Chapter 4

Workload Characterization

GPU computation consists of a sequence of SIMD instructions, each operating on vector operands in multiple threads. We define a scalar instruction as a SIMD instruction operating on the same data for all the active threads in a warp. We refer to any other SIMD instruction as a vector instruction.

Scalar opportunity analysis can be performed at different abstraction levels. Compiler-level analysis is more flexible and needs zero hardware modifications, but it can only identify scalar opportunities within a thread block or coarser structure since intra-thread-block information is dynamic. Also, it may be conservative since the compiler has to consider all possible control flow paths. Architecture/microarchitecture-level analysis is more informative since it is equipped with run-time information. Working at this level, we can handle scalar opportunities at a finer grain, such as a warp/wavefront, but at the cost of more hardware. In the following we will perform both static and dynamic analysis on a set of benchmarks.

4.1 Static Scalar Analysis

We characterize scalar opportunities on NVIDIA PTX code for the following two reasons: (1) PTX is stable across multiple GPU generations, which makes our approach more general, and (2) there are several existing PTX research tools available to use [42, 43]. However, we claim that our analysis is independent of any specific SIMD programming model and thus applies to most SIMD-based instruction sets, including NVIDIA SASS [24], AMD IL [44], and AMD ISA [45].

Consider the following example of a vector addition kernel. Figure 4.1 shows the CUDA code and its corresponding PTX code. Using a single character, we denote at the beginning of each
Figure 4.1: Details of a CUDA vector addition kernel and the resulting PTX code. The annotation “1” or “0” at the beginning of each PTX instruction denotes a scalar and vector instruction, respectively.
CHAPTER 4. WORKLOAD CHARACTERIZATION

PTX instruction whether it is a scalar or vector instruction. The “1” or “0” denotes a scalar or vector instruction, respectively.

We can see from the CUDA code that the variable \( i \) is initialized to the global thread index at first, which is computed using the thread block dimension, thread block index, and local thread index in a thread block. The corresponding PTX code is using three vector registers \( r_1, r_0, \) and \( r_3 \) to keep track of those three operands, respectively. Since these values will be the same for every warp, the first three PTX instructions are scalar ALU instructions. Later in the kernel, the fourth instruction moves the thread index to \( r_3 \), which will be processing different data for each thread. Thus, a vector instruction is generated. There are four instructions that load the kernel parameters into a register, and so they are scalar memory instructions. We also identify the branch instruction as a scalar operation since we only worry about the active threads, and they all transfer program control to the program counter (PC). We should also note that the floating-point addition in the example may be scalar for some warps, depending on whether the components of \( A \) and \( B \) participating in the computation are the same. This class of scalar instructions can only be identified at run time since the operand values can vary at runtime.

To quantify the scalar opportunities, we determine first if a vector operand contains the same data for all the components corresponding to active threads. If this condition is satisfied, we call it a uniform vector; otherwise, it is divergent. A scalar opportunity requires that all of its source operands are uniform.

We implement the static variable divergence analysis proposed by Coutinho et al. [32], to decide whether an operand is uniform or divergent. It first performs a PTX-to-PTX code transformation in order to handle both data dependence and sync dependence. Then it identifies the variables reached, starting from potentially divergent variables, such as the thread ID and atomic instructions. As shown in Figure 4.2, all the variables reached are marked divergent (black circles); the others are uniform (white circles).

When performing variable divergence analysis on a data dependence graph, we add a tag to each variable indicating whether it is uniform or divergent. Then we run static scalar opportunity analysis on a control flow graph using the previously generated tags, as illustrated in Figure 4.2. A SIMD instruction is recognized as a scalar opportunity (white box) if and only if all of its source operands are uniform.

We added a compiler pass to GPU Ocelot [42] to perform our static analysis. Ocelot is a modular dynamic compilation framework for heterogeneous systems, providing various backend targets for CUDA programs and analysis modules for the PTX virtual instruction set. In these
Figure 4.2: An example of scalar opportunity analysis (vector addition).
CHAPTER 4. WORKLOAD CHARACTERIZATION

experiments, we compiled CUDA source code to PTX code first, and then used Ocelot to generate flags indicating if a static instruction is a scalar opportunity.

![Figure 4.3: Percentage of static scalar opportunities.](image)

We counted the number of static scalar opportunities using Ocelot. As Figure 4.3 shows, 38% of static SIMD instructions on average are detected by the compiler as scalar opportunities. The results imply that scalar opportunities always exist in GPU applications, even if we use SIMD programming models to write and optimize our programs.

In some benchmarks such as SobolQRNG, the percentage of static scalar opportunities is significantly higher than the percentage measured during runtime. This difference implies that scalar opportunities in those benchmarks are likely used during the initialization phase of the code, and thus do not benefit from scalar opportunities in the main loops of these programs. On the other hand, in some benchmarks such as histogram256, the percentage of static scalar opportunities is much lower than the number that are executed. In such cases, scalar opportunities are very likely present in the main loops of these benchmarks.

Static statistics are insufficient since the frequency of scalar opportunities at run time directly determines how well the scalar units are utilized. Assume a program has ten static instructions, where five instructions are non-scalar opportunities in a loop executing 100 iterations, and the others are scalar opportunities outside of the loop. Then the percentage of static scalar opportunities is $5/10 = 50\%$, while that of dynamic scalar opportunities is $5/(5 + 5 \times 100) = 1\%$. Scalar units
CHAPTER 4. WORKLOAD CHARACTERIZATION

will be underutilized if a program has limited dynamic scalar opportunities. Hence, we also count
dynamic occurrences of static scalar opportunities.

Without run-time information, static analysis must be conservative. Specifically, uniform
vectors may be recognized as divergent in variable divergence analysis, and so some scalar opportuni-
ties are not detected. For instance, an instruction subtracting a divergent vector from itself produces
a uniform vector 0. However, because the destination vector has a data dependency on a divergent
source vector in the data flow graph, it is labeled a divergent vector. In such cases, if any following
instruction uses the vector as input, and also uses other uniform vectors as source operands, the
instruction will be classified as a non-scalar opportunity, while it is actually a scalar opportunity.
Another example is when a conditional branch is taken/not taken for all threads, i.e., warp divergence
does not happen, the variables defined between the branches and their immediate post-dominators are
uniform. However, they are recognized as divergent since the compiler has to consider conservatively
that the branch diverges.

Dynamic analysis can generate run-time statistics under those circumstances. Nevertheless,
hardware modification is required for dynamic analysis, which will incur high cost. Also, run-time
information may heavily depend on program inputs, resulting in unique statistics for different inputs.

4.2 Dynamic Scalar Analysis

We designed and implemented our scalar/vector detection logic on the GPGPU-Sim 3.2.2
simulator [43]. We have implemented our dynamic scalar/vector instruction detection logic right
after the operands are fetched and buffered in the operand collector units. We use the predicate values
to determine the active status of threads in a warp. If an instruction works on the same operands
across all the active threads, it is identified as a scalar instruction. Otherwise, it will remain a vector
instruction.

Table 4.1 presents the benchmarks used in our experiments. We selected a wide range
of programs from the Parboil 2.5 [5] and the Rodinia 3.0 [6, 7] benchmark suites, providing us
with a wide range of scalar and vector instruction ratios. Figure 4.4 shows the number of dynamic
scalar versus vector instructions. We can see that 42% of dynamic warp instructions are scalar
instructions on average. Several benchmarks (e.g., K-means) have more scalar instructions than
vector instructions. The main reason why an application has a large number of scalar instructions
executed is due to the loading of constants and kernel parameters, and then computations using those
Table 4.1: The benchmarks selected from Parboil [5], Rodinia [6, 7], and CUDA Samples [8], providing a rich range of behaviors for this study.

<table>
<thead>
<tr>
<th>Application</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first search</td>
<td>BFS</td>
</tr>
<tr>
<td>Back propagation training</td>
<td>BP</td>
</tr>
<tr>
<td>B+ Tree</td>
<td>BT</td>
</tr>
<tr>
<td>Computational fluid dynamics (10 iterations)</td>
<td>CFD</td>
</tr>
<tr>
<td>Distance-cutoff coulombic potential</td>
<td>CUTCP</td>
</tr>
<tr>
<td>Gaussian elimination</td>
<td>GE</td>
</tr>
<tr>
<td>Saturating histogram</td>
<td>HISTO</td>
</tr>
<tr>
<td>HotSpot processor temperature estimator</td>
<td>HS</td>
</tr>
<tr>
<td>Heart wall tracking (1 frame)</td>
<td>HW</td>
</tr>
<tr>
<td>K-means clustering</td>
<td>KM</td>
</tr>
<tr>
<td>Leukocyte tracking</td>
<td>LC</td>
</tr>
<tr>
<td>LU decomposition</td>
<td>LUD</td>
</tr>
<tr>
<td>Cardiac myocyte simulation</td>
<td>MC</td>
</tr>
<tr>
<td>Particle potential and relocation calculation (4 boxes)</td>
<td>MD</td>
</tr>
<tr>
<td>Magnetic resonance imaging - gridding (10% samples and 32x32x32 matrices)</td>
<td>MRIG</td>
</tr>
<tr>
<td>Magnetic resonance imaging - Q</td>
<td>MRIQ</td>
</tr>
<tr>
<td>K-nearest neighbors</td>
<td>NN</td>
</tr>
<tr>
<td>Needleman-Wunsch DNA sequence alignments</td>
<td>NW</td>
</tr>
<tr>
<td>Particle filter</td>
<td>PF</td>
</tr>
<tr>
<td>Sum of absolute differences</td>
<td>SAD</td>
</tr>
<tr>
<td>Single-precision dense matrix multiplication</td>
<td>SGEMM</td>
</tr>
<tr>
<td>Shortest path on a 2D grid</td>
<td>SP</td>
</tr>
<tr>
<td>Sparse-matrix dense-vector multiplication</td>
<td>SPMV</td>
</tr>
<tr>
<td>Speckle reducing anisotropic diffusion (real images)</td>
<td>SRAD1</td>
</tr>
<tr>
<td>Speckle reducing anisotropic diffusion (random inputs)</td>
<td>SRAD2</td>
</tr>
<tr>
<td>3D stencil operation</td>
<td>STENCIL</td>
</tr>
<tr>
<td>Two point angular correlation function</td>
<td>TPACF</td>
</tr>
</tbody>
</table>
constant operands. If programs run on current GPU architectures contain a large number of scalar instructions, but they have no scalar units, execution will be highly inefficient.

We further analyzed the types of scalar and vector instructions, as shown in Figure 4.5 and Figure 4.6. All scalar instructions are either ALU, non-texture memory, or control flow instructions. In order to improve execution efficiency, our scalar unit should be able to execute these three instruction types. On the other hand, most MAD/FMA, transcendental and texture instructions are vector instructions (i.e., their operands change across threads in a warp). We should continue to execute them on SIMD units to keep the scalar unit design fast and streamlined.

Figure 4.7a and Figure 4.7b illustrate how the percentage of scalar instructions varies during the execution for BFS and K-means, respectively. The percentage is computed for each time period. Those plots illustrate two very different behaviors. The percentage of scalar instructions in BFS decreases by 15% as the program runs. At the beginning of BFS, many constants and kernel parameters are loaded as part of program initialization, which is why the percentage is high. Later, vector variables, such as thread identifiers, dominate the instruction mix, so the percentage of scalar operations is reduced. In contrast, the percentage of scalar instructions in K-means increases by 9% after initialization. Inspecting the code, we see that a number of the computations in the main loops work with constants as inputs. Frequent use of those constants produces a stable mix of scalar
CHAPTER 4. WORKLOAD CHARACTERIZATION

Figure 4.5: The breakdown of scalar instructions.

Figure 4.6: The breakdown of vector instructions.
CHAPTER 4. WORKLOAD CHARACTERIZATION

(a) BFS: the percentage of scalar instructions decreases during execution.

(b) K-means: the percentage of scalar instructions increases during execution.

Figure 4.7: The percentage of scalar instructions varies during execution.
CHAPTER 4. WORKLOAD CHARACTERIZATION

instructions.
Chapter 5

Scalar-Vector GPGPU Architecture

Next, we describe the design and implementation details of our scalar-vector GPU architecture. First, we present our scalar-vector multiprocessor design. Second, we will show the optimized instruction dispatcher and warp scheduling units used to balance the scalar and vector execution. Finally, we will consider the effects of warp size on scalar/vector performance and consider subwarp execution for more balanced resource utilization.

5.1 Scalar-Vector Multiprocessor Architecture

Figure 5.1 shows our scalar-vector multiprocessor architecture. We added several new hardware components to support scalar execution, which are highlighted in the shaded boxes. Additionally, we modified the existing components in the hatched boxes to improve performance. The following will describe them in detail.

The scalar detector identifies scalar instructions by comparing the operand values in a warp. As soon as the operands are read from the register file to the buffers in the collector units, the detector starts checking if all active components hold the same value. If so, the operand is labeled with a “1” in the scalar operand bit; otherwise it is labeled with a “0”. If all the source operands for an instruction are labeled as scalar operands, we mark this instruction with a “1” in the “scalar instruction bit”. Otherwise, the instruction is marked with a “0”. This information will be used in the instruction dispatcher.

Since the detector is on the critical path, we make sure that it does not become the bottleneck of the pipeline. We record the register identifiers of the scalar operands in a table indexed by the warp identifier. The comparison logic looks up the scalarness of the operand in the table in parallel
Figure 5.1: The scalar-vector multiprocessor architecture. The shaded and hatched boxes represent the new or modified hardware components, respectively.
with the register read operation by the warp. If the scalarness of the operand is found in the table, the operand collector units will send a single-component read request to the register file, saving register file bandwidth in the memory banks to service other requests. Otherwise, the comparison logic will run as soon as the operand is read back. When there are multiple instructions in the operand collection stage, the comparison logic prioritizes the instruction with the lowest PC.

We add several operand collector units to the scalar functional units so that more active warps and instructions can be handled concurrently. This produces a wider window of instructions for the instruction dispatcher to issue scalar and vector instructions. This ability helps hide latency, since it will be easier to find opportunities for scalar and vector units to run in parallel.

Our scalar execution unit has ALU, memory, and control flow pipelines to serve the corresponding scalar instructions. The scalar ALU is similar to a single-laned SIMD unit, but runs much faster with lower hardware complexity. We added two scalar ALUs to a multiprocessor, each with a 3x faster clock than the clock feeding the SIMD unit, in order to logically provide a 1:1 ratio of SIMD units to scalar units. After execution, the scalar ALUs write the result to the scalar register file.

The scalar memory pipeline can load or store data from/to a specialized scalar data cache to/from a separate scalar register file. This design is very efficient for loading constants and kernel parameters (the most common scalar memory instructions), since we do not have to broadcast the values. The scalar memory pipeline can also load a single component of a vector variable from the vector caches. We maintain the consistency between the vector and scalar register files in the scalar alias table, which will be discussed in the next section.

The scalar control flow pipeline can handle branches efficiently. A scalar branch implies that all the threads of a warp have the same next PC. This PC is computed by the scalar ALU, which runs at a higher clock frequency. Moreover, the branch instruction does not have to update the SIMT stack. Our proposed fast scalar control flow path can therefore reduce branch latency.

In order to better satisfy various scalar/vector resource requirements in our workloads, we introduce a new utilization level table. It uses a sliding window to monitor the utilization of scalar and vector units. If the scalar unit has very low utilization, it is turned off to save power. The following scalar instructions are issued to vector units. Meanwhile, the scalar detector keeps monitoring the percentage of the scalar instructions. When the percentage of scalar instructions reaches a threshold, the scalar unit is turned on to improve execution efficiency. The utilization information is also used by the warp scheduler and the instruction dispatcher to make informed decisions, which will be discussed in Section 5.2 and Section 5.3. An alternative option is to increase the clock frequency.
CHAPTER 5. SCALAR-VECTOR GPGPU ARCHITECTURE

when the scalar unit is turned off. This way we can achieve higher performance within the power budget.

![Modified Operand Collector Architecture](image)

Figure 5.2: An example of the modified operand collector architecture. “vr” and “sr” represent vector and scalar register files, respectively. “-” denotes that the value is not checked by the dispatcher.

Scalar instructions can also be executed on the vector units accompanied by an effective dispatching policy. This allows us to trade off power for performance. In such cases, the scalar unit collectors will need to broadcast scalar data to all of the active components. Specifically, when the utilization of the scalar unit is higher than a set threshold, the scalar unit collectors will broadcast operands and set a “vector fit” bits to inform later dispatch and issue stages. The dispatcher is then enabled to issue scalar instructions to vector units. When the scalar unit utilization is low, the scalar operands do not broadcast and the dispatcher can only issue scalar instructions to the scalar units. Two examples are shown in Figure 5.2. The instruction in Warp 4 does not broadcast its scalar operand, so it can only be executed on the scalar unit. Alternatively, the instruction in Warp 8 can be executed on the vector unit with the “vector bit” set to 1.

### 5.2 Instruction Dispatch Optimizations

The instruction dispatcher accepts instructions from the warp scheduler and issues them to scalar and/or vector units. It exploits the instruction-level parallelism available in an application to improve performance. There are two dispatchers for each warp scheduler in our microarchitecture, and they can issue up to two independent instructions from one warp per cycle.
CHAPTER 5. SCALAR-VECTOR GPGPU ARCHITECTURE

(a) The performance with the round-robin scheduling policy (26 cycles).

(b) The performance with the improved scalar-aware round-robin scheduling policy (21 cycles).

(c) The performance with the further improved scalar-aware greedy scheduling policy (20 cycles).

Figure 5.3: An example showing the performance improvement with the optimized warp scheduling policy. “s” and “v” denote a scalar and vector instructions executing on the corresponding unit, respectively. The hatched area indicates that there is no instruction available to issue due to a lack of warps.
A straightforward design is to have one scalar dispatcher and one vector dispatcher, and have scalar instructions only execute on scalar units. Since the two dispatchers share a single scalar unit, an arbiter is provided to determine which dispatcher has priority for the scalar unit and the other reissues in the next cycle. This policy has low hardware complexity and provides good power efficiency, but the performance benefits may be modest, especially for scalar-dominated workloads.

Alternatively, we can allow scalar instructions to be issued to vector units. This is more flexible in terms of preventing starvation from over-allocation when executing scalar-dominated workloads. When two ready instructions from a warp can be issued, we consider the following four cases. When a scalar instruction is issued to a vector unit, we assume that the operands have already been broadcast by the operand collector. The instructions in the tuple are ordered by their PCs.

\(\text{(Vector, Vector)}\). If subwarp execution mode is off, instructions are issued as normal to the vector units. Otherwise, some or all thread instructions can be issued to scalar units, which will be discussed in Section 5.4.

\(\text{(Scalar, Vector)}\). The scalar and vector instructions are issued to a corresponding unit, if one is available. Otherwise, the instructions need to wait until the next cycle.

\(\text{(Vector, Scalar)}\). The vector instruction is issued to a vector unit, if one is available. Otherwise, it waits until the next cycle. The scalar instruction is issued to a scalar unit, if one is available. Otherwise, it can be issued to a vector unit with low utilization.

\(\text{(Scalar, Scalar)}\). Both instructions are issued to scalar units if two are available. If only one scalar unit is available, the first instruction is issued to the scalar unit and the second to a vector unit, but only if a vector unit is available and its utilization is low. When there are no scalar units available, the first instruction is issued to a vector unit if one is available and its utilization is low, and the second waits until the next cycle.

We believe that the policies proposed above can help resolve future data dependencies earlier and do not over-commit vector units, resulting in more balanced execution. In order to fully exploit the instruction-level parallelism available, we propose a scalar aliasing scheme to reduce Write-After-Read and Write-After-Write data hazards. When an operand from the vector register file is identified as an operand for a scalar operation, the collector unit allocates a scalar register to store a copy of the data and records the scalar alias in the alias table. The following instructions that use the vector register will try to read the value of the scalar alias from the scalar register file, saving the bandwidth to the vector register file for other vector read requests. Figure 5.2 shows a scalar alias example. The instruction from Warp 15 reads the source operands from vector registers 4 and 8. The detector identifies them as scalar values, and the collector unit allocates scalar registers 15 and 16.
as their scalar aliases, respectively. So the future instructions that need to read vector register 4 can redirect their requests to scalar register 15.

The vector register maintains consistency with its scalar alias in the scoreboard. If an instruction is executed on the scalar unit, it writes the result to the scalar register as well as to the corresponding vector register as needed. The entry in the alias table is also maintained. On the other hand, if an instruction is executed on the vector unit and its destination operand has a scalar alias, the entry in the alias table will be removed in the write-back stage.

5.3 Warp Scheduling Optimizations

The Kepler SMX architecture can be split into two warp clusters, each with two warp schedulers sharing a pool of up to 32 warps [3]. Compared to the Fermi SM which allowed 48 warps to be managed by two schedulers, the Kepler pool design has a smaller size, which implies less latency hiding.

Figure 5.3a shows an example of the pipeline performance with the round-robin scheduling policy. Each of the four warps has five instructions. Each instruction encounters a data dependency with the previous instruction. A scalar instruction takes one cycle to complete and a vector instruction takes six cycles to complete. With a round-robin warp scheduler, there is no instruction available to execute in the shaded cycles since the result from the previous instruction has not yet been written back.

In order to reduce the stalls caused by long-latency instructions, we propose a new warp scheduling algorithm. All the warps in a scheduler are divided into two groups: the scalar group and the vector group. A warp is assigned to a group based on which type of the unit it is executed on.

Figure 5.3b shows our scalar-aware round-robin scheduling algorithm. We give priority to scheduling warps on the scalar units because: 1) the scalar pipeline is faster, and thus the generated result can resolve data dependencies earlier, and 2) the ready warps are issued while other warps wait for the results, which reduces the number of idle cycles.

The algorithm works as follows. The scheduler selects the warps in the scalar group (if any are waiting) in a round-robin fashion. If the warp chosen is executed on the scalar unit, it will be kept in the scalar group; otherwise, it will be moved to the vector group. If there is no warp in the scalar group, the scheduler will check the vector group and schedule them also in a round-robin fashion. The example shown in Figure 5.3b uses this algorithm and saves five idle cycles. There is still one idle cycle when all the warps are executed on the vector units while waiting for the results.
We further optimize the algorithm by allowing earlier execution of the vector group, as shown in Figure 5.3c. We use a greedy algorithm here for each warp so that Warp 2 executes vector instructions and generates results earlier. Meanwhile, the scheduler also checks the oldest warp in the vector group and issues it if it is ready to run. This effectively prevents the vector group from starving. Using this algorithm further removes one idle cycle. Our optimized warp scheduling algorithm improves thread-level parallelism and reduces the number of idle cycles when executing long-latency instructions.

5.4 Subwarp Execution

![Graph showing the percentage of scalar and vector instructions](image)

Figure 5.4: The number of scalar instruction increases with smaller warp sizes.

Warp size can impact the execution efficiency of our scalar-vector GPU. An extreme case occurs when the warp size is one, thus all instructions will be scalar. We have observed that as the warp size decreases, more instructions become scalar, as is shown in Figure 5.4.

The best warp size for the scalar-vector balance varies across workloads. If we choose a size that is too large, scalar units will be underutilized. On the contrary, if the warp size is too small, vector units will starve. Therefore, we propose a subwarp execution mechanism to dynamically balance the utilization of scalar and vector units.
CHAPTER 5. SCALAR-VECTOR GPGPU ARCHITECTURE

We consider two subwarp patterns: 1) consecutive and 2) odd/even. For the consecutive pattern, one warp is split into multiple subwarps consisting of consecutive threads. For the odd/even pattern, one warp is split into two subwarps based on whether the thread IDs is either odd or even. One vector warp can be split into multiple scalar subwarps, which can be used to improve performance or reduce power utilization, or scalar subwarps and vector subwarps, which can be used to reduce power utilization.

The operand collector can perform warp partitioning based on the utilization information. When the scalar units have low utilization, the operand collector can break a long-latency vector warp instruction into multiple scalar and vector subwarp instructions. The destination vector registers will not be released until all the subwarps complete.

5.5 Evaluation

Table 5.1: The simulator configurations used in this work. Our implementation has been calibrated against an NVIDIA Kepler K40 GPU.

<table>
<thead>
<tr>
<th>Category</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-level architecture</td>
<td>15 multiprocessors at 750 MHz</td>
</tr>
<tr>
<td>Multiprocessor architecture</td>
<td>Maximum 2048 threads/16 thread blocks, 32 threads/warp</td>
</tr>
<tr>
<td></td>
<td>192 cores, 4 warp schedulers, 8 dispatchers, 64 K 32-bit registers</td>
</tr>
<tr>
<td></td>
<td>16 KB L1 d-cache, 48 KB shared memory, 64 KB constant cache</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Single-stage bufferfly architecture at 750 MHz, 32 B flit size</td>
</tr>
<tr>
<td>L2 cache</td>
<td>1536 KB at 750 MHz</td>
</tr>
<tr>
<td>GDDR5 DRAM</td>
<td>1500 MHz command clock, 288 GB/s bandwidth</td>
</tr>
<tr>
<td></td>
<td>6 memory controllers, 16 banks</td>
</tr>
<tr>
<td></td>
<td>Timing: tCCD = 2, tRRD = 6, tRCD = 12, tRAS = 28, tRP = 12,</td>
</tr>
<tr>
<td></td>
<td>tRC = 40, tCL = 12, tWL = 4, tCDLR = 5, tWR = 12</td>
</tr>
<tr>
<td></td>
<td>4 bank groups with tCCDL = 3 and tRTPL = 2</td>
</tr>
</tbody>
</table>

We used the GPGPU-Sim 3.2.2 simulator [43] and calibrated the baseline against the NVIDIA Kepler K40 GPU architecture. The key parameters are listed in Table 5.1. GPGPU-Sim is a detailed cycle-level GPU performance simulator. As shown in Figure 5.5, it has several Single Instruction Multiple Thread (SIMT) cores connected via an on-chip interconnection network to lower level memory. A SIMT core models a pipelined SIMD multiprocessor, where each lane corresponds to a basic single-ALU processing element. A SIMD instruction is executed on a SIMT core as follows. First, the instruction is fetched from the instruction cache, decoded, and then stored in the instruction buffer. The instruction buffer is statically partitioned so that all warps running on the
SIMT core have dedicated storage to place instructions. Then the issue logic checks all the valid instructions, which are decoded but not issued, to establish issue eligibility. A valid instruction can be issued if the following three requirements are all satisfied: (1) its warp is not waiting at a barrier, (2) it passes the Write After Write (WAW) or Read After Write (RAW) hazards check in the scoreboard, and (3) the operand access stage of the instruction pipeline is not stalled. Memory instructions are issued to the memory pipeline. The other instructions always prefer SIMD units to special function units (SFU) unless they have to be executed on special function units. The pipeline also maintains a SIMT stack per warp to handle branch divergence. Moreover, an operand collector offers a set of buffers and arbitration logic used to provide the appearance of a multi-ported register file using multiple banks of single-ported RAMs. The buffers hold the source operands of instructions in collector units. When all the operands are ready, the instruction is issued to an execution unit.

We modified the simulator to implement our scalar-vector architecture and report on speedup achieved by our scalar configurations relative to the baseline. Figure 5.5 shows most of the major modifications we made to GPGPU-Sim to model our scalar=vector architecture. These changes include:

- **Execution unit.** We added a configurable number of scalar units, each of which is pipelined and can execute all types of ALU instructions (except transcendental). They have the same speed as SIMT units, i.e., execute one instruction per cycle. Each unit has an independent issue port from the operand collector, and share the same output pipeline register as other execution
CHAPTER 5. SCALAR-VECTOR GPGPU ARCHITECTURE

units that are connected to a common writeback stage.

- **Operand collector.** We added a configurable number of collector units for each scalar unit. The collector units have a similar structure to others structures, but incur fewer register reads since they merely need the operands for one thread.

- **Issue logic.** We allow schedulers to issue instructions to scalar units, in addition to SIMD units; otherwise, the instructions will never run in parallel since both can execute an instruction per cycle. The issue width of our simulator is configurable. Moreover, scalar opportunities should be able to run on SIMD units as well for flexibility, though we may choose to restrict this option when optimizing for power.

- **Configuration options.** The configurable parameters described above are added as new configuration options for GPGPU-Sim.

We used GPUWattch [46] as an architecture-level power model to collect the power results. The power model of the scalar units is based on an in-order single-core processor pipeline. We added several new activity counters specific to our scalar-vector design to allow for more accurate power calculations, including scalar unit accesses, scalar register file accesses, and scalar cache hit/miss rate. They are passed by GPGPU-Sim to the underlying modified McPAT [47] power modeling framework in GPUWattch.

Table 4.1 lists the workloads used in our work. We selected from Parboil 2.5 [5] and Rodinia 3.0 [6, 7] benchmark suites all the benchmarks that can run on GPGPU-Sim, providing a range of workloads with scalar-dominated, as well as vector-dominated, behaviors.

We use the total cycles executed as the primary performance metric. Reporting instructions per cycle is not accurate, since the scalar execution reduces the effective instruction count. Additionally, we consider auxiliary metrics such as scalar and vector unit utilization and the effective issue width. These metrics will be used to better understand the effects of our proposed warp scheduling and instruction dispatching optimizations.

To begin our evaluation, we compare the utilization of the vector units of the baseline versus utilization of the scalar-vector architecture, as shown in Figure 5.6. Most benchmarks have significantly lower vector-unit utilization on the scalar-vector architecture than the vector-only baseline, since the scalar units introduced handle part of the computation.

We next investigate if we have balanced the load on the scalar and vector units by comparing the utilization of the scalar and vector units, as shown in Figure 5.7. In general, a number of the
CHAPTER 5. SCALAR-VECTOR GPGPU ARCHITECTURE

Figure 5.6: The utilization of the vector units on the baseline versus the scalar-vector architecture.

Figure 5.7: The utilization of the scalar and vector units.
benchmarks have similar or slightly higher vector utilization. The reason for this is that scalar instructions are allowed to be executed on vector units if the scalar unit is overutilized, while vector instructions can also be executed on scalar units if they can be split into multiple subwarp scalar instructions. So we achieve a more balanced distribution of the loads across the scalar and vector units.

Figure 5.8: The percent increase in the effective issue width relative to the baseline.

We define the effective issue width as the average number of instructions issued per cycle in a multiprocessor. We consider this metric to investigate the effectiveness of our instruction dispatcher and warp scheduling schemes. As we can see in Figure 5.8, the effective issue width is improved by 37%. There are two main reasons: (1) our scalar aliasing scheme is able to resolve data dependencies earlier so that the dispatchers can issue more instructions per cycle, which increases the instruction level parallelism, and (2) our warp scheduling policy reduces the number of pipeline stall cycles, which improves thread-level parallelism.

Figure 5.9 presents performance improvements achieved for the scalar-vector design relative to our vector-only baseline. We report on the number of cycles saved for each workload. Most benchmarks perform significantly better on the scalar-vector architecture than the baseline. For a few benchmarks (e.g., STENCIL), performance degrades slightly. We have observed a higher vector cache miss rate in those benchmarks. The reason is that a portion of data is saved in the scalar
CHAPTER 5. SCALAR-VECTOR GPGPU ARCHITECTURE

Figure 5.9: The performance improvements of the scalar-vector GPU architecture.

cache on the scalar-vector architecture. If they are used by the following vector instructions, vector cache misses will result. To address this, we plan to explore a novel scalar/vector cache design in the next steps in this thesis.

Our subwarp execution improves the power efficiency of GPU architectures. We evaluated the power savings with the 8-thread subwarp execution (i.e., each 32-thread warp is split into four 8-thread subwarps). Figure 5.10 shows that power consumption is improved by 18% on average. The workloads dominated by scalar instructions (e.g., TPACF) enjoy significant power savings. A few workloads (e.g., MC) did not receive any real improvement since (1) the extra power introduced by the new components offset the power saved by scalar execution, and/or (2) shader power accounts for a small part of the overall power usage.

Scalar opportunities can put pressure on the multiprocessor pipeline, interconnection network and memory subsystem, as shown in Figure 5.11. Multiprocessor pipeline stalls can be caused by shared memory bank conflicts, non-coalesced memory accesses, or serialized memory accesses. Interconnection network stalls happen when DRAM channels cannot accept requests from the interconnect. Memory stalls result from interconnection network congestion when DRAM channels cannot send packets. In the figure, a positive number implies that stall cycles are increased on scale-vector GPU architectures over the baseline, while a negative number suggests that they are
CHAPTER 5. SCALAR-VECTOR GPGPU ARCHITECTURE

Figure 5.10: The power improvements of the scalar-vector GPU architecture.

Figure 5.11: Stall cycles difference on scalar-vector architecture over the baseline.
CHAPTER 5. SCALAR-VECTOR GPGPU ARCHITECTURE

decreased.

The benchmarks show a range of different results. BlackScholes and quasirandomGenerator place more pressure on multiprocessor pipelines, while MersenneTwister places less pressure. Moreover, for binomialOptions, scalar units relieve much of the pressure on the interconnect. In contrast, several other benchmarks including dwtHaar1D and scalarProd experience a higher number of interconnect stalls. Additionally, some benchmarks such as MonteCarlo place additional stress on memory.

When designing a scalar-vector GPU architecture, we need to keep in mind that when we add scalar units to the microarchitecture, we may need to increase our interconnect and memory bandwidth to guarantee efficient data delivery to these units. We need to consider the entire data path so that we do not create another hotspot in the microarchitecture. This will be considered in our next steps in this thesis.
Chapter 6

Scalar-Assisted Prefetching

In this chapter, we explore a prefetching mechanism with scalar units. Prefetching fetches the data before it is requested to reduce memory latency, which is a widely used technique in modern CPUs to reduce compulsory misses. Several GPU prefetching mechanisms have been proposed [48, 49, 50, 51]. They all use the SIMD units to prefetch data, sacrificing compute throughput.

We propose a scalar unit assisted prefetching mechanism, which uses scalar units to prefetch data when they are idle, leaving the precious SIMD units for computations. This way we achieve more efficient execution and higher performance than with SIMD-based prefetching.

6.1 Prefetching Opportunities

We first identify the prefetch opportunities available from both the software and hardware. Many GPU compute applications exhibit regular memory access patterns, which can benefit from conventional stream or stride prefetching [52, 53]. Also, the scalar units on the scalar-vector GPU architecture are idle when there are no scalar instructions to execute. We can use them to assist in prefetching, without sacrificing SIMD compute throughput.

6.1.1 Regular vs. Irregular Memory Access Patterns

Regular memory access patterns are observed in many GPU compute applications. They refer to a series of memory addresses that mostly depends on static information. For example, a vector addition program has a regular memory access pattern. The memory address to access an
element is statically determined as long as the data structure to store the vector is known. It has no relation with the element value.

On the contrary, a memory access pattern is called irregular if the memory addresses depend on the program input values. A commonly cited example of irregular accesses is the reference stream associated with a linked list. We cannot predict how the data values will be stored before we know the values of the elements and have knowledge on the order associated with the particular data structure.

There are two types of memory access patterns on the GPU architecture: intra-warp and inter-warp patterns. The intra-warp pattern refers to the memory accesses generated at different PCs in an individual warp. Since a warp has one single instruction stream, intra-warp prefetching on GPU is similar to single-thread CPU prefetching, which has been extensively studied. On the other hand, the inter-warp pattern refers to the memory accesses generated at the same PC across different warps, which is a new class of pattern observed primarily on many-threaded GPU architectures. Since a GPU frequently switches warps to hide latency, inter-warp memory access patterns heavily affect performance.

![Figure 6.1: The utilization of the scalar units.](image-url)
We analyzed both inter-warp and intra-warp memory access patterns in the benchmarks listed in Table 4.1. We can see from Figure 6.1 that regular access patterns are very common across various benchmarks. We begin with focusing on prefetching for regular memory access patterns in this chapter.

### 6.1.2 Scalar Unit Utilization

![Figure 6.2: The utilization of the scalar units.](image)

We have balanced the load on the scalar and vector units in Section 5.1. Figure 6.2 shows the utilization of the scalar units across all the benchmarks studied. On average, the scalar units achieve approximately 35% utilization. Thus, they are available 65% of the time, so explore how to use them for prefetching during idle periods.

### 6.2 Prefetching Mechanism

A prefetching mechanism has two essential performance challenges: accuracy and timeliness [52, 54]. Accuracy is determined by how well we predict the location of data ahead of demand.
CHAPTER 6. SCALAR-ASSISTED PREFETCHING

Low accuracy implies that useless data is prefetched, wasting resources such as cache space, memory bandwidth, and energy. On the other hand, timeliness is determined by how well we predict the time when data will be required by the program. If data are fetched too early, they may be replace before they are used. Alternatively, if data is fetched too late, a cache miss will result. Both cases produce ineffective prefetching.

Figure 6.3: The scalar-vector multiprocessor architecture. The shaded and hatched boxes represent the new or modified hardware components, respectively.

Figure 6.3 shows the flow of our prefetching mechanism on a scalar-vector multiprocessor architecture. We added a new table to collect memory access information for the prefetcher’s use, which is highlighted in the shaded boxes. Additionally, we modified several existing components in the hatched boxes to enable prefetching. The following describes design modifications in detail.

The new prefetching request table records all the information related to prefetching memory
requests. The key fields include the warp ID, memory address, and a time stamp. The coalescer reads the information in the prefetching request table, computes prefetching addresses, and generates requests. Prefetching and normal memory requests are pushed into a single access queue.

Prefetched data is stored in the L1 scalar or vector cache, depending on whether the instruction is for a scalar or vector operation. If the instruction is a scalar operation, the scalar memory pipeline will perform the same as in the normal scalar-vector GPU architecture. The scalar unit prefetches the data and stores it in the scalar cache. If the instruction is a vector operation, the scalar unit will compute the memory address for one lane, and the coalescer will generate a memory request for all the lanes. The prefetched data is stored in the vector cache.

We also add logic to monitor the accuracy of the prefetcher. We use the L1 data cache eviction rate as the metric. When the eviction rate is higher than a predefined threshold, the prefetching accuracy is considered low and the prefetcher is turned off.

### 6.2.1 Warp Scheduling Optimization

Prefetching performance heavily depends on the warp scheduling policy that is used. For inter-warp prefetching, one warp prefetches data for another. In order for the prefetched data to be used in time, the time between the prefetch and the use of the prefetch needs to be managed careful. Our approach is to focus on the scheduling of use of the prefetched data. If it is executed too early, the data may not be ready. If it is executed too late, the data may have been evicted. So we need to optimize the warp scheduling policy to guarantee that warps are executed at the right time.

We allow the prefetcher to pass on information about the prefetched data to the scheduler. Then scheduler can update the priority of the warps in the warp pool, so that the warps have completed the prefetch of the data are more likely to be executed first.

Figure 6.4a shows an example of pipeline performance assuming a round-robin scheduling policy. Each of the four warps generates a prefetching request for another. Without the prefetching information, the warp scheduler keeps the original policy, and all warps are executed too early with no prefetched data available. The prefetches are useless and the related hardware resources are wasted.

Figure 6.4b shows our prefetching-aware warp scheduling algorithm. Equipped with the added information about the prefetched data, we give priority to scheduling the warps whose data are ready. This way the warps are executed at the right time, neither too early nor too late, as long as there are enough warps in the warp pool. If there are only a few warps, the prefetched data will
6.3 Evaluation

Figure 6.5 presents the L1 cache hit rate improvements achieved with and without our prefetching mechanism. All the benchmarks have significantly higher hit rates with the prefetching mechanism. The reasons are twofold. When the prefetching accuracy is high, prefetched data is more likely to be useful, which improves the hit rate. When the prefetching accuracy is low, the prefetcher is turned off to avoid polluting the cache.

Figure 6.6 presents performance improvements with our prefetching scheme. We report on the number of cycles saved for each workload. All the benchmarks perform significantly better with the prefetching mechanism. We can see there is a positive correlation between how regular the memory access patterns are (in Figure 6.2) and the amount that performance is improved. So our prefetching mechanism effectively exploits regular memory access patterns in the GPU, patterns that are common in compute applications.
Figure 6.5: The utilization of the scalar units.

Figure 6.7 presents the change in the global memory bandwidth with the prefetching mechanism. All the benchmarks have higher global memory bandwidth.
Figure 6.6: The utilization of the scalar units.
Figure 6.7: The utilization of the scalar units.
Chapter 7

Conclusion

To summarize, this thesis has made the following contributions.

- We have characterized a wide range of GPU compute benchmarks, and showed that GPU compute applications typically have a mix of scalar and vector instructions. This information is used to guide our microarchitectural design.

- We proposed a scalarness-aware compiler optimization and a hardware-based scalar instruction detection mechanism. We examined the advantages and disadvantages of each approach.

- We designed a scalar-vector GPU architecture and examined the opportunities and challenges for various design alternatives on several microarchitectural components.

- We proposed a scalar unit assisted prefetching mechanism on the scalar-vector GPU architecture. We achieved more efficient execution and higher performance than the vector-based prefetching.

7.1 Future Research Directions

Our scalar-vector architectures can still be further tuned given the impact of scalar execution. As we can see in Section 5.5, the interconnect network can be a performance hotspot in the scalar-vector GPU architecture. A mix of scalar and vector memory requests make the interconnect traffic more complicated. One interesting research direction in the future is to examine and optimize the interconnect network. We plan to investigate its performance problem and explore the design tradeoffs, in order to achieve the efficiencies provided by scalar opportunities in common applications.
CHAPTER 7. CONCLUSION

Also, our work is focused on GPU compute workloads only. Modern GPUs run various types of applications, such as graphics and machine learning. A future direction is to investigate reasons in source for scalar opportunities, study the performance of those applications on the scalar-vector GPU architecture, and future optimize the architecture. This way we will achieve higher overall efficiency.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


