A MapReduce Framework for Heterogeneous Computing Architectures

Marwa K. Elteir

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Wu-chun Feng, Chair
Heshan Lin
Ali Raza Ashraf Butt
Eli Tilevich
Xiaosong Ma

August 15, 2012
Blacksburg, Virginia

Keywords: Graphics Processing Unit, Heterogeneous Computing, Programming Models, MapReduce, Atomics

A MapReduce Framework for Heterogeneous Computing Architectures

Marwa K. Elteir

ABSTRACT

Nowadays, an increasing number of computational systems are equipped with heterogeneous compute resources, i.e., following different architecture. This applies to the level of a single chip, a single node and even supercomputers and large-scale clusters. With its impressive price-to-performance ratio as well as power efficiency compared to traditional multicore processors, graphics processing units (GPUs) has become an integrated part of these systems. GPUs deliver high peak performance; however efficiently exploiting their computational power requires the exploration of a multi-dimensional space of optimization methodologies, which is challenging even for the well-trained expert. The complexity of this multi-dimensional space arises not only from the traditionally well known but arduous task of architecture-aware GPU optimization at design and compile time, but it also arises in the partitioning and scheduling of the computation across these heterogeneous resources. Even with programming models like the Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL), the developer still needs to manage the data transfer between host and device and vice versa, orchestrate the execution of several kernels, and more arduously, optimize the kernel code.

In this dissertation, we aim to deliver a transparent parallel programming environment for heterogeneous resources by leveraging the power of the MapReduce programming model and OpenCL programming language. We propose a portable architecture-aware framework that efficiently runs an application across heterogeneous resources, specifically AMD GPUs and NVIDIA GPUs, while hiding complex architectural details from the developer. To further enhance performance portability, we explore approaches for asynchronously and efficiently

distributing the computations across heterogeneous resources. When applied to benchmarks and representative applications, our proposed framework significantly enhances performance, including up to 58% improvement over traditional approaches to task assignment and up to a 45-fold improvement over state-of-the-art MapReduce implementations.

Dedicated to my wonderful husband, Atef, for his endless love, support and understanding

Acknowledgments

All praise be to God for giving me the patience and power to survive in my pursuit of graduate studies.

I would like to thank many people, this dissertation would not be possible without their help. First and foremost, I would like to thank my advisor, Dr. Wu-chun Feng, for his continuous guidance, valuable advices and encouragements. I am also really grateful to him for accepting me in the Synergy Lab and for allowing me to participate in the group meetings while being remotely. This greatly helps me achieve sustainable progress during my graduate study. I also owe a lot of thanks to my co-advisor, Dr. Heshan Lin, for his generous and patient guidance. He has always been available at anytime to discuss my research progress. He has taught me many important skills that had and will have great impact on me. I really appreciate his effort and assistance.

I am thankful to my committee members: Dr. Ali R. Butt, Dr. Xiaosong Ma and Dr. Eli Tilevich for their support, feedback, and flexibility when I was scheduling my preliminary exam and my final defense between semesters.

I would like to thank all members of the Synergy Lab for their invaluable discussions during the group meetings. These meetings have always been a source of inspiration. I especially would like to thank Tom Scogland, Ashwin M. Aji, Umar Kalim, Konstantinos Krommydas, Kenneth Lee, Lokendra Singh, Balaji Subramaniam, Shucai Xiao and Jing Zhang.

Very special thanks go to Dr. Sedki Riad and Dr. Yasser Hanafy for making the VT-MENA program available for Egyptian students. It has been my great fortune to join this program.

Finally, I would like to thank the special ones. Thank you my wonderful husband for your support and understanding, for taking care of our kids while I had to attend meetings and

concentrate on my work, and for always encouraging me during my hard times. I would like to tell you that I could not make it without your support. I would like to thank my kids, Nour and Ahmed, for bearing a busy mum, I'm really happy to reach this stage so I can requite you. I would like to express my especial gratitude to my parents for all what they did for me, for praying for me during my hard times, and for raising me up to have endless trust that God must reward me for my efforts. Without this belief, I cannot survive until this moment. I'd like to especially thank my mother for taking care of my kids and regularly helping me to save me time to work on my PhD. I am also thankful to my twin sister Safa and my dearest friends Amira Soliman and Dina Said for their love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past few years, the graphics processing unit (GPU) has become a commodity component of most computers. The success of the GPU as a computing resource comes from its low cost, high computing power, and power efficiency compared to multicore CPUs. Nowadays, we are moving towards greater heterogeneity along all levels of computing. At the level of a single chip, fusing the CPU and the GPU on a single chip with one shared global memory is a reality e.g., AMD Fusion [6], Intel Xeon Phi [38], and Nvidia's Tegra chip [61] for mobile phones which combines ARM processors with GeForce graphics cores. At the level of a single node, the integration of heterogeneous computing resources including traditional multicore CPU and accelerators such as GPUs in a single node has become mainstream [78]. The heterogeneity even spans up to the level of supercomputers and clusters. For the June 2012 list of the The TOP500 list [85], the fifth supercomputer is built from Intel Xeon X5670 CPUs and NVIDIA 2050 GPUs. In addition, the National Science Foundation announced HokieSpeed [10], a supercomputer consisting of 200+ GPU-accelerated nodes. Each of the nodes is equipped with a dual socket Intel X5645 6C 2.40 GHz CPU and 2 NVIDIA Tesla M2050/C2050 GPUs (Fermi). This configuration enabled HokieSpeed to debut as the greenest commodity in the U.S. on the Green500 in Novemeber 2011. These

emergent systems provide the necessary computing power required by high-performance computing (HPC) applications from diverse domains including scientific simulation [22], bioinformatics [75, 54], image analysis [26], and databases [60, 65].

Although several researchers have reported tremendous speedups from porting their applications to GPUs, harnessing the power of heterogeneous resources is still a significant challenge. This is largely attributed to the complexities of designing optimized code for heterogeneous architectures as well as the partitioning and scheduling of the computation among these resources.

In this dissertation, we propose a framework for efficiently exploiting the computing power within and across heterogeneous resources. Our overarching goal is to deliver a transparent parallel programming environment for heterogeneous resources. Towards achieving this goal, we adopt a three-step approach. The first step involves developing a portable framework across different compute resources including multicore CPUs and accelerators such as NVIDIA GPUs, AMD GPUs, APUs, Cell, and FPGA. In particular, the developer should write his code once, and then the framework transparently exploits the architecture details of different devices to efficiently run this code. Letting these heterogeneous resources working together to accelerate an application requires careful distribution of the computation. Thus, in the second step, we focus on developing an efficient workflow that concurrently distributes the computation among heterogeneous resources. Furthermore, since different applications have different computing patterns and input/output characteristics, the framework should dynamically identify the combination of resources that is best suited for the target application based on its characteristics. Finally, in the third step, a performance model should be derived to estimate the execution time of the application based on its computing and input/output characteristics as well as the capabilities of the heterogeneous devices. Based on this model, the framework should adaptively choose the appropriate set of resources to execute a given application. In this dissertation, we leverage the power of the MapReduce

programming model to address the first and second steps. We leave the third step for future work. For the first step, our case study focuses on the portability across AMD GPUs and NVIDIA GPUs.

In the rest of this chapter, we provide the necessary context for understanding the research performed in this dissertation. Specifically, Section 1.1 and 1.2 discuss the problems that we seek to address, the research objectives, and the research contributions we make to address the above first and second steps, respectively. Section 1.3 outlines the remainder of this dissertation.

1.1 Portable Architecture-Aware MapReduce Framework

1.1.1 Problem Statement

Fully exploiting the computational power of a graphics processing unit is a complex task that requires exploring a multi-dimensional space [62, 8, 54, 73, 72], including proper adjustment of the number of threads executing a kernel, making use of low-latency memories, e.g., local memory and registers, avoiding divergence, coalescing memory accesses, using vector types, and so on. This task is further exacerbated by the complexity of debugging and profiling GPU kernels.

Although all graphics processors share the same high-level architecture i.e., made up of several compute units, where each unit contains multiple processing elements executing in a SIMD fashion, in addition to a hierarchical memory system, GPUs from different vendors have their own characteristics. For example, the AMD GPU that we study adopts vector cores instead of scalar ones. It also has two memory paths with significantly different band-

width; each specialized in handling specific memory transactions. Additionally, it contains only one branch unit per processing element. Considering these subtle architecture details is crucial for designing efficient code. Generally, different optimization methodologies [54] have to be explored for different devices, thus complicating the task of designing an optimized code for heterogeneous resources.

Programming models such as NVIDIA's Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) facilitate GPU programming. Although these programming models enhance the programmability of GPUs, the developer still needs to manage the data transfer between host and device and vice versa, orchestrate the execution of several kernels, and more arduously optimize the kernel code. Higher-level programming models like MapReduce can help to mask a lot of these complexities. It shows great success in hiding complexities of programming large scale clusters of thousands of commodity computers [41, 1]. Currently, there are several MapReduce implementations targeting other systems [68, 17, 12], however all of them focus on multicore CPUs and NVIDIA GPUs. We lack a MapReduce framework that targets parallel computing devices more generally, e.g., CPUs, AMD GPUs and NVIDIA GPUs, while still delivering performance portability across other heterogeneous devices, such as NVIDIA GPUs.

1.1.2 Methodology and Contribution

Our objective is three-fold: (1) facilitate programmability of heterogeneous resources, (2) efficiently exploit the computing power of heterogeneous resources, and (3) deliver functional and performance portability across heterogeneous architectures. To achieve these goals we leverage the MapReduce programming model and OpenCL programming language to design and develop an efficient and portable framework across different resources. The MapReduce programming model [41] offers high-level APIs to express the functionality of an applica-

tion and hide the architectural details of the underlying system, thus achieving the first goal. Currently, OpenCL has been adopted by many processor vendors [8, 63, 34, 19], so developing our framework using OpenCL achieves the portability goal. This portability is just functional portability. To ensure performance portability, we propose and implement efficient architecture-aware techniques to execute the MapReduce tasks. First, we consider the architectural details of AMD GPUs since these GPUs have not been studied before in the context of MapReduce, then we further optimize our framework to work efficiently on other devices as well, such as NVIDIA GPUs.

In particular, we investigate the applicability of the state-of-the-art MapReduce designs to AMD GPUs. These MapReduce designs depend on executing atomic-add operations to coordinate output writing from thousands of concurrently running threads. Our investigations show that using global atomic operations can cause severe performance degradation on AMD GPUs i.e., up to 69.4-fold slowdown [8]. This is attributed to the fact that including atomic operations in the kernel forces all memory transactions in this kernel to follow a slow memory path rather than a fast one. So the designed framework should completely avoid the use of global atomic operations. Consequently, we design and implement [50] a software-based atomic operation that does not impact the used memory path. Using this software atomic operation, we implement a MapReduce framework that behaves efficiently and significantly outperforms state-of-the-art MapReduce implementations on AMD GPUs. The main shortcoming of the proposed software-based atomic is that it supports applications running limited number of threads.

To address the limitation of the proposed software-atomic operation, we propose and implement an atomic-free design for MapReduce, StreamMR [51], which can efficiently handle applications running any number of threads. We introduce several techniques to completely avoid the use of atomic operations. Specifically, the design and mapping of StreamMR provides efficient atomic-free algorithms for coordinating output from different threads as

well as storing and retrieving intermediate results via distributed hash tables. StreamMR also includes efficient support of combiner functions, a feature widely used in cluster-based MapReduce implementations but not well explored in previous GPU-based MapReduce implementations. StreamMR significantly outperforms the state-of-the-art implementation of MapReduce, e.g., up to 45-fold faster than MapCG on AMD GPUs. We further optimize StreamMR [48] to work efficiently on other heterogeneous devices that do not suffer from the penalties associated with the use of atomic operations, e.g., NVIDIA GPUs. Specifically, we propose a mechanism for improving the scalability of the reduce phase with the size of the intermediate output. With the highly scalable reduce phase, StreamMR outperforms MapCG on a NVIDIA GPU by up to 3.5-fold speedup.

1.2 Optimized MapReduce Workflow

1.2.1 Problem Statement

The traditional approach [41] for scheduling the map and reduce tasks across resources is to force a barrier synchronization between the map phase and the reduce phase. So, the reduce phase can only start when all map tasks are completed. There are several cases where this barrier synchronization can result in serious resource underutilization. First, when distributing the computation across heterogeneous resources whether within node or across nodes, it is highly expected that the faster compute resources will finish their assigned map tasks earlier, but these resources cannot proceed to the reduce processing until all the map tasks are finished, thus wasting resources. Second, since different resources are appropriate for different computations i.e., sequential code is suitable for CPU, while data-parallel code is suitable for GPU, so for efficient execution of the map and reduce computation, we may end up scheduling the map computation on one resource i.e., GPU, and the reduce computation

on another resource, i.e., CPU, which leaves the CPU unutilized during the map phase. Even in homogeneous environments, we have noticed that a compute node/resource may not be fully utilized by the map processing due to the fact that a map task alternates between computation and data accessing. Based on the above, we have explored other approaches for scheduling the map and reduce tasks across resources.

1.2.2 Methodology and Contribution

We aim at improving the resource utilization by efficiently distributing the map and reduce tasks across the heterogeneous resources. Our solution starts by initially assigning the map tasks to the resources, and then improving the resource utilization through overlapping the computation of the map and reduce phases. Specifically, we propose two approaches to cope with such heterogeneity [49]. The first proposed approach is the hierarchical reduction, which overlaps map and reduce processing at the inter-task level. It starts a reduce task as soon as a certain number of map tasks complete and aggregates partially reduced results following a tree hierarchy. This approach can be effective when there is enough overlap between map and reduce processing. However, it has some limitations due to the overhead of creating reduce tasks on the fly, in addition to the extra communication cost of transferring the intermediate results along the tree hierarchy. To cope with this overhead, we proposed an incremental reduction approach, where all reduce tasks are created at the start of the job, and every reduce task incrementally reduces the received map outputs. Both approaches can effectively improve the MapReduce execution time. The incremental reduction approach consistently outperforms hierarchical reduction and the traditional synchronous approach. Specifically, incremental reduction can outperform the synchronous implementation by up to 58%. As a part of this investigation, we derive a rigorous performance model that estimates the speedup achieved from each approach.

1.3 Organization of the Dissertation

The rest of this dissertation is organized as follows: in Chapter 2, we present some background and discuss the related work. In Chapter 3, we present our investigations towards developing MapReduce implementation for AMD GPUs. In Chapter 4, we propose StreamMR, across-platform MapReduce implementation optimized for heterogeneous devices. We explore two different approaches for distributing map and reduce tasks among different resources in Chapter 5. In Chapter 6, we conclude the dissertation. Finally, we discuss potential future projects in Chapter 7.

This page intentionally left blank.

Chapter 2

Background and Related Work

2.1 GPU Architecture

All GPU devices share the same high-level architecture. They contain several SIMD units; each has many cores. All cores on the same SIMD unit execute the same instruction sequence in a lock-step fashion. All SIMD units share one high-latency, off-chip memory called global memory. Host CPU transfers the data to global memory through a PCI-e path. There is also a low-latency, on-chip memory that is shared by all cores in every SIMD unit named local memory. In addition to the local and global memory, there are two special types of memories i.e., image memory and constant memory that are also shared by all compute units. Image memory is a high-bandwidth memory region whose reads may be cached. Constant memory is a memory region storing data that are allocated/initialized by the host and not changed during the kernel execution. Access to constant memory is also cached. Below, we present the main differences between AMD and NIVDIA GPU architectures.

2.1.1 AMD GPU

For the AMD GPU, each core within the SIMD unit is a VLIW processor containing five processing elements, with one of them capable of performing transcendental operations like sine, cosine, and logarithm. So, up to five scalar operations can be issued in a single VLIW instruction. Double-precision, floating-point operations are executed by connecting two or four processing elements. Each core also contains one branch execution unit that handles branch instructions. This makes AMD GPUs very sensitive to branches in the kernel, whether divergent or not.

As shown in Figure 2.1, for the AMD Radeon HD 5000 series of GPUs, the local memory, Local Data Store (LDS), is connected to L1 cache. Several SIMD units share one L2 cache that is connected to the global memory through a memory controller. There are two independent paths for memory access: FastPath and CompletePath [8]. The bandwidth of the FastPath is significantly higher than the CompletePath. Loads and stores of data whose size is multiple of 32 bits are executed through the FastPath, whereas advanced operations like atomics and sub-32 bit data transfers are executed through the CompletePath.

Executing a memory load access through the FastPath is performed by a single vertex fetch (vfetch) instruction. In contrast, a memory load through the CompletePath requires a multi-phase operation and thus can be multiple times slower to the AMD OpenCL programming guide [8]. The selection of the memory path is done automatically by the compiler. The current OpenCL compiler maps all kernel data into a single unordered access view. Consequently, including a single atomic operation in a kernel may force all memory loads and stores to follow the CompletePath instead of the FastPath, which can in turn cause severe performance degradation of an application as discovered by our work [50]. Note that atomic operations on variables stored in local memory do not impact the selection of memory path.

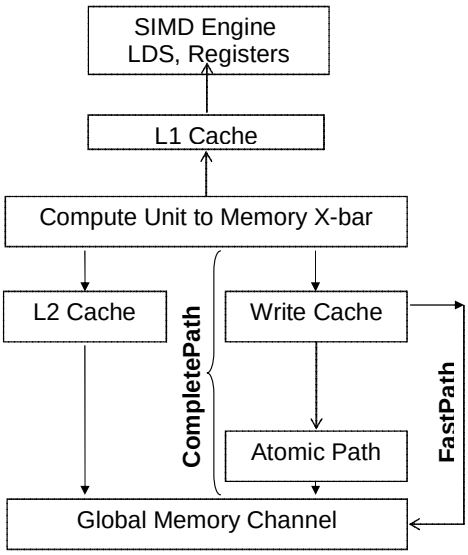

Figure 2.1: AMD GPU memory hierarchy

2.1.2 NVIDIA GPU

For an NVIDIA GPU and Fermi in particular [64], each core within each SIMD unit contains two processing elements, one for executing integer operations, and another one for floating-point operations. Thus each core can perform one single-precision Fused Multiply Add (FMA) operation in each clock cycle and one double-precision FMA in two clock cycles. In addition, each SIMD unit has 4 Special Functions Units (SFUs) to handle transcendental and other special operations such as sine, cosine, and logarithm. Only four of these operations can be issued per cycle in each SIMD unit. So, compared to the AMD GPU, the NVIDIA GPU needs more cycles to execute double-precision, floating-point operations and special operations. Fermi GPU has a chip-level scheduler named GigaThread scheduler which enables Fermi to execute multiple concurrent kernels, a feature that is unique to Fermi GPU.

2.2 GPU Programming Models

GPUs are originally designed for data-parallel, graphics-based applications. However, the introduction of some programming models have made general-purpose computing on GPUs (i.e., GPGPU) possible [28]. These programming models are NVIDIA's Compute Unified Device Architecture (CUDA)[62], AMD's Brook [7], and OpenCL [31]. Here we present the most commonly used models.

2.2.1 CUDA

CUDA [62] is the first programming model proposed and developed by NVIDIA to support general-purpose computing on NVIDIA GPUs. It provides a set of extensions to C programming language to differentiate between the functions running on host and device. Using CUDA terminology, the device code is called a kernel and it must be launched to the device before execution. The kernel runs as a multithreaded program, with threads grouped into blocks. Each block is assigned to one SIMD unit; however a SIMD unit can run multiple blocks concurrently. A group of threads named warp is scheduled together; the scheduling of these warps to the physical SIMD units is done at runtime by CUDA thread scheduler. CUDA only supports synchronization between threads of the same block. To synchronize all threads, another kernel must be launched.

2.2.2 OpenCL

OpenCL [31] is a programming model that aims at supporting heterogeneous computing. It was initially developed by Apple, and then submitted to Khronos Group to develop its specification, which released in 2008. Currently, most major processor vendors released an OpenCL implementation for their products including NVIDIA[63], AMD[8], IBM[34], and

Intel[19]. For AMD GPUs, OpenCL becomes its main programming language.

Using OpenCL terminology, each thread of a kernel is called a *workitem* and executed on a single core. Multiple workitems are organized into a *workgroup*. One or more workgroups can run concurrently in a SIMD unit. The resource scheduler executes each workgroup as several *wavefronts* (a wavefront is similar to the warp concept in CUDA). To hide memory latency, it switches between the wavefronts whenever anyone is waiting for a memory transaction to complete. Also synchronization primitive is provided to synchronize threads in a single workgroup only.

2.3 MapReduce Programming Model

MapReduce is a high-level programming model aims at facilitating parallel programming by masking the details of the underling architecture. Programmers need only to write their applications as two functions: the *map* function and the *reduce* function. All of the input and outputs are represented as *key/value* pairs. Implementing a MapReduce framework involves implementing three phases: the *map* phase, the *group* phase, and the *reduce* phase. Specifically, the MapReduce framework first partitions the input dataset among the participating parties (e.g. threads). Each party then applies the map function to its assigned portion and writes the intermediate output (map phase). The framework groups all of the intermediate outputs by their keys (group phase). Finally, one or more keys of the grouped intermediate outputs are assigned to each party, which will carry out the reducing function and write out the result key/value pairs (reduce phase).

MapReduce was first proposed by Google [41] in 2004. It seeks to simplify parallel programming on large-scale clusters of computers. The success of MapReduce on porting scientific and engineering application to large-scale cluster motivated other MapReduce implementa-

tions on different platforms. In the following, we discuss different MapReduce implementations categorized based on the target platform.

2.3.1 MapReduce on Large-Scale Clusters

Several research efforts have been done to enhance the performance of the original MapReduce and add more functionality. In [69] a programming language named Sawzall was built over MapReduce framework. It aims at automatically analyzing a huge distributed data files. The main difference between it and the standalone MapReduce framework is that it distributes the reduction in a hierarchical topology-based manner. The reduction is performed first per machine, followed by reduction per rack, followed by final global reduction.

Furthermore, in [33], the authors believe that the original MapReduce framework is limited to be used with applications like relational data processing. So they presented a modified version of the MapReduce named MapReduceMerge framework which works exactly the same as the original framework, however the reduce workers produce a list of key/values pairs that are transmitted to the merge workers to produce the final output.

Moreover, Valvag et al. [87] developed a high-level declarative programming model and its underlying runtime, Oivos, which aims at handling the applications that require running several MapReduce jobs. This framework has two main advantages compared with MapReduce. First, it handles the overhead associated with such type of applications including monitoring the status and progress of each job, determining when to re-execute a failed job or start the next one, and specifying a valid execution order for the MapReduce jobs. Second, it removes the extra synchronization when these applications are executed using the traditional MapReduce framework, i.e., every reduce task in one job should complete before any of the map tasks in the next job can start.

Steve et. al, [76] realized that the loss of intermediate map outputs may result in a significant

performance degradation. Although using HDFS (Hadoop Distributed File System) improves the reliability, it results in considerably increasing the job's completion time. As a result, they proposed some design ideas for a new intermediate data storage system.

Zahria et al. [52] proposed another speculative task scheduler named LATE (Longest Approximate Time to End) to cope with several limitations of the original Hadoop's scheduler. It aims at improving the response time of the jobs by speculating the tasks that are expected to finish late. It is mainly applicable to heterogeneous environments or virtualized data centers like Amazon EC2 [4].

Condie et al. [86] extended the MapReduce architecture to work efficiently for online jobs in addition to batches. Instead of materializing the intermediate key/value pairs within every map task, they proposed pipelining these data directly to the reduce tasks. They further extended this pipelined MapReduce to support interactive data analysis through online aggregation, and continuous query processing.

2.3.2 MapReduce on Multicore CPUs

Phoenix [18] is the first implementation for MapReduce on small-scale multicore and multiprocessor systems. The Phoenix runtime system forces barrier synchronization between the map and reduce phases. It dynamically schedules the map and reduce tasks to the cores, thereby achieving balance among the cores. To enable locality, Phoenix adjusts the granularity of the map and reduce tasks so that the data manipulated by each task fits in the L1 cache. The reported performance is very close to a manually optimized pthreads code, however the scalability is limited. The next generation of Phoenix [68] enhances the scalability on large-scale systems with NUMA characteristics by adapting the runtime to be NUMA-aware. The grouping of the intermediate results in Phoenix is done by hashing technique instead of sorting. To the best of our knowledge, they are the first to propose

hashing for the grouping phase of MapReduce.

Recently Hong et. al [17] proposed a portable implementation for MapReduce, MapCG, that can run efficiently in both CPU and GPU. The key contribution in the CPU implementation is the use of an efficient memory allocator that greatly reduces the number of calls to *malloc()*. This is important especially for applications emitting large intermediate and final output.

2.3.3 MapReduce on GPUs

Mars [12] is the first MapReduce implementation on GPUs. One of the main challenges of implementing MapReduce on GPUs is to safely write the output to a global buffer without conflicting with output from other threads. Mars addresses this by calculating the exact write location of each thread. Specifically, it executes two preprocessing kernels before the map and reduce phases. The first kernel counts the size of the output from each map/reduce thread by executing the map/reduce function without writing the generated output to the global buffer. The second kernel is a prefix summing that determines the write location of each thread. Each thread then reapplies the map/reduce function and safely writes the intermediate/final output to the predetermined location in the global buffer. After the map phase, Mars groups the intermediate output by their keys using bitonic sort. After similar preprocessing kernels (counting and prefix summing), the reduce phase starts, where every thread reduces the values associated with certain key and finally writes the generated key/value pair to the final output. One main disadvantage of Mars' preprocessing design is that the map and reduce functions need to be executed twice. Such a design was due to the fact that atomic operations were not supported on the GPUs at the time Mars was developed.

Recently Hong et al. proposed MapCG [17], an implementation for MapReduce on both CPU and GPU. Its GPU implementation depends on using atomic operations to safely write

the intermediate and final output. Also, MapCG designed a memory allocator to allocate buffers from the global memory for each warp. Moreover, MapCG uses hash tables to group intermediate output from map function, which is shown to be more efficient than the sorting used in Mars.

There is another study on accelerating MapReduce on GPUs [25] that is orthogonal to our work. In [25], Ji et al. proposed several techniques to improve the input/output performance by using shared memory as a staging area. They also depend on atomic operations to coordinate the writing of the output to the global buffers. The sorting technique is exploited to group the intermediate results. Our investigation shows that MapReduce implementations that depend on global atomic operations [17, 25] can cause severe performance degradation on AMD GPUs.

Instead of implementing MapReduce as a runtime framework on GPUs [12, 17, 25], the authors of [13] implemented a code generation framework that generates the application code from two user-defined functions, i.e., map and reduce function. To facilitate the deployment of this framework on GPUs, several constraints are forced on the map and reduce phases. The size of the intermediate output is assumed to be known; in addition each map function only produces one output. Also, the reduce function should be associative to enable a hierarchical reduction phase. All of these constraints limit the applicability of this framework.

2.3.4 MapReduce on Heterogeneous Architectures

Here we discuss the implementations of MapReduce on the Cell Broadband Engine (Cell BE) and CPU/GPU co-processing based implementations. Cell BE [81] is an example of heterogeneous computing resource. It contains one general-purpose Power Processing Element (PPE), and eight Synergistic Processing Elements (SPE), each has SIMD unit.

There are two implementations for MapReduce on Cell BE [47, 9]. Both implementations

exploit the PPE to control the MapReduce runtime, i.e., task instantiation, task scheduling, data transfer, and synchronization. The actual map and reduce computations are handled by homogeneous cores, i.e., SPEs, so the challenges of managing different heterogeneous cores are not handled in these implementations. To handle applications with nondeterministic output size, the work in [47] uses an approach similar to the one proposed by [17, 25]. Specifically, the PPE is responsible for allocating memory for all SPEs, a mechanism that requires significant communication and synchronization between the PPE and SPEs. On the other hand, [9] uses a SPE-centric approach similar to Mars' counting phase [12], where every SPE runs a run-ahead map task to measure the buffer needed by each task. Two different mechanisms are proposed to schedule the tasks: (1) overlapping tasks across adjacent MapReduce stages [47] and (2) enforcing a barrier between any two stages citepapa-icpp10-cellmapreduce. Along with the dynamic scheduling of tasks, the barrier approach balances the tasks among SPEs and minimizes the control overhead, thereby achieving better performance.

CPU/GPU coprocessing MapReduce implementations are proposed in [89, 17]. In [89], the map and reduce tasks are statically distributed among the CPU and GPU so that the data assigned to GPU is S times larger than that assigned to CPU, where S is the speedup of the GPU over CPU. In [17], the co-processing details are not presented. Both implementations do not report significant improvement over the GPU-only implementation. It should be noted that using CPUs with a larger number of cores and better scheduling mechanism, the performance may be improved [26, 83].

In [58], Linderman et al. proposed a MapReduce framework named Merge that can automatically distribute the computation across multiple heterogeneous resources. It also provides a predicate dispatch-based library for managing function variant for different architectures. Although the framework is generic and can support any architecture, it does not take into account the data transfer overhead when making scheduling decisions. All of the results

reported show a significant speedup when using multiple CPUs and one integrated GPU (Intel Graphics Media Accelerator X3000). Other discrete GPUs need to be tested to show the scalability of the framework.

2.3.5 MapReduce on Clusters of Accelerators

Several studies focus on enabling MapReduce in a cluster of accelerators, i.e., GPUs, Cells, or both. GPMR [40] is a MapReduce implementation for a cluster of GPUs. The CPU in each node is responsible for scheduling the map and reduce phases on the GPU and managing the communication with other nodes. The design of GPMR is concerned mainly with minimizing the communication cost between different GPUs through the use of combiner and local reduction. GPMR also introduces several application-specific optimizations to improve the program performance.

In [67], a Monte Carlo simulation is formulated using Massive Unordered Distributed (MUD) formalism. This enables running it as a MapReduce application on a cluster of GPUs, leveraging Hadoop [1] and NVIDIA CUDA. The evaluation shows that 4-node GPU cluster can achieve more than 4-fold speedup compared to Hadoop cluster of 62 nodes. Although trying to architect a general framework, MITHRA, to handle any MapReduce application, the grouping phase is not implemented, thus limiting the generality of the framework. Also the user should provide a CUDA implementation of their map and reduce functions, knowledge of GPU programming and optimization. It should be noted that the same idea of leveraging Hadoop to enable MapReduce on a cluster of GPUs was explored by the Mars's authors in [89].

In [45, 46], a scalable MapReduce implementation is proposed for a cluster of accelerators, i.e., GPUs and Cells. The implementation follows a centralized approach to manage all nodes; so a single manager is responsible for assigning the MapReduce tasks to all accelerators and

merging the results into the final output. For better scalability, several handler threads are launched in the manager; each handles one accelerator. To cope with accelerators of different capabilities, the manager dynamically adjusts the granularity of the assigned tasks by monitoring the accelerator performance to tasks of different sizes. This implementation adopts existing MapReduce implementations [12, 47] to run the assigned tasks on the GPU and Cell. So, it only targets NVIDIA GPUs i.e., there is no support for AMD GPUs. Also, the MapReduce computations are handled by the accelerators only; the CPUs control their attached accelerators, thus leaving some resources unutilized.

2.3.6 Limitations of Previous MapReduce Solutions

Despite of the diverse MapReduce implementations existing in the literature, none of these efforts target AMD GPUs. All of them target either multicore CPUs, Cells, or NVIDIA GPUs. The architecture of AMD GPUs has unique characteristics that require revisiting the existing MapReduce implementations to attain the expected performance. Also, the MapReduce implementations on heterogeneous resources do not exploit the computing power of general-purpose cores. In [47, 9], the PPE is only used to control the execution, and the actual computations are distributed among SPEs. Although this is acceptable for Cells, it is not the case for multicore CPUs that are expected to have 10s of cores [35, 36]. The CPU/GPU coprocessing implementation of MapReduce is not deeply studied in the literature. CPU should act as an active member in the computation by processing part of the map and reduce tasks. Furthermore, each of the above implementations targets a specific platform and is optimized for its architecture, having one implementation that is portable among different platforms is another challenge.

2.4 Other Runtime Solutions

Over the past few years, there have been many efforts to address the challenges of programming heterogeneous resources. In [30], a runtime supported programming and execution model named Harmony is proposed. Starting from an application code, Harmony builds an acyclic dependency graph of the encountered kernels. It then uses dynamic mapping to map each kernel to the heterogeneous resources. Although a single kernel can be mapped to the CPU and the GPU concurrently, it is not clear how the ratio of CPU-to-GPU computation is adjusted. Also, to be able to run a given kernel on different architectures, the programmer should implement it using different languages, i.e., C for CPU kernels and CUDA for GPU kernels. StartPU [14], on the other hand, aims at more generally studying the problem. It provides a uniform interface to support implementing different scheduling algorithms, and then allowing the user to use the most appropriate strategy. This approach may be efficiently handle different classes of applications; however it places additional purden on the programmer. To adjust the task granularity, StarPU supports either using a pre-calibration run or dynamically adapting the ratio at runtime. Qilin [15] proposes a runtime system that depends on adaptive dynamic mapping to map the code to the computing resources, either GPU or CPU or both. It conducts training for the first time a program is run and curve fitting to adjust the ratio of computations assigned to CPU vs. GPU. Also the programmer should provide two versions of the kernel; one in thread building blocks (TBB) for CPU and one in CUDA for GPU. In [59], Michela Becchi et. al. propose a runtime system that targets legacy kernels. It uses a performance model to estimate the execution time of a certain function call on both CPU and GPU, based on profiling information obtained from runs with different data sizes. The runtime also optimizes the data transfer by deferring all transfer until necessary. Although this system takes into account the data transfer overhead, it does not utilize it to guide the scheduling decision. Also, a function call is either executed on CPU or GPU i.e., no co-processing of a single function on both CPUs and GPUs is supported.

Recently, Scogland et al. [83] extend OpenMP for accelerators to enable porting an existing OpenMP code to heterogeneous environments. It provides new OpenMP directives for the programmer to indicate the code region to be parallelized across the heterogeneous cores. At runtime, the performance of CPU to GPU on different number of iterations is used to adjust the ratio between the CPU-to-GPU work.

There are three main shortcomings of these previous solutions. First, none of them consider the data transfer overhead while making the scheduling decision, which can greatly impact the performance especially for discrete GPUs. Second, all of them target NVIDIA GPUs, none of them support AMD GPUs. Finally, except for [83], the programmer should provide two implementations; one to run on CPU and other one to run on GPU.

This page intentionally left blank.

Chapter 3

Optimized MapReduce Framework for AMD GPUs

3.1 Overview

Currently, all existing MapReduce implementations on GPUs focus on NVIDIA GPUs. So the design and optimization techniques in these implementations may not be applicable to AMD GPUs, which have a considerably different architecture than NVIDIA ones as discussed in Chapter 2. For instance, State-of-the-art MapReduce implementations on NVIDIA GPUs [17, 25] rely on atomic operations to coordinate execution of different threads. But as the AMD OpenCL programming guide notes [8], including an atomic operation in a GPU kernel may cause all memory accesses to follow a much slower memory-access path, i.e., CompletePath, as opposed to the normal memory-access path, i.e., FastPath, even if the atomic operation is not executed. Our results show that for certain applications, the atomic-based implementation of MapReduce can introduce severe performance degradation, e.g., a 28-fold slowdown on AMD GPUs.

Although Mars [12] is an existing atomic-free implementation of MapReduce on GPUs, it has several disadvantages. First, Mars incurs expensive preprocessing phases (i.e., redundant counting of output records and prefix summing) in order to coordinate result writing of different threads. Second, Mars sorts the keys to group intermediate results generated by the map function, which has been found inefficient [17].

In this chapter, we propose a MapReduce implementation for AMD GPUs. The main design goal is to avoid the use of global atomic operations. To achieve this goal, we start by developing an efficient software-based atomic operation that can efficiently and safely update a shared variable, and at the same time, does not affect the performance of other memory transactions. This software-based atomic operation is then used to develop an efficient MapReduce framework for AMD GPUs.

The rest of this chapter is organized as follows: In Section 3.2, we have quantified the performance impact of atomic operations to simple kernels and MapReduce implementations, respectively. The design of our software-based atomic add is described in Section 3.3 and 3.4. In Section 3.5, we have derived a model of kernel speedups brought by our software atomic operations. Performance evaluations are then presented in Section 3.6. We conclude in Section 3.7.

3.2 Quantifying the Impact of Atomic Operations on AMD GPUs

We seek to quantify the performance impact of atomic operations on memory access time, we run the simple kernel code, shown in Figure 3.1, on the Radeon HD 5870 GPU. The code includes only two instructions; the first is an atomic add operation to a global variable, and the second is a memory transaction that reads the value of the global variable and writes it

to an element of an array.

```
__kernel void Benchmark (__global uint *out,
                         __global uint *outArray)
{
    int tid = get_global_id(0);

    // Safely incrementing a global variable
    atom_add(out,tid);

    /* Writing the value of the global variable
       to an array element */
    outarray[tid]=*out;
}
```

Figure 3.1: A simple copy kernel with atomic add operation

We measure the kernel execution time of three versions of the aforementioned kernel, as shown in Figure 3.2. The first version contains only the atomic operation. The second contains only the memory transaction. The third contains both. Ideal represents the sum of the execution times of the atomic-only and the memory transaction-only versions.

By analyzing the ISA code, we found that the number of CompletePath memory accesses is 0 and 3 for the second and third versions, respectively. As a result, the memory access time increases significantly by 2.9-fold and 69.4-fold for 8 and 256 workgroups, respectively, when including the atomic operation. Note that, as the number of memory transactions in the kernel increases, the impact of accessing the memory through the CompletePath is exacerbated, as discussed in Section 3.6.

The above results suggest that using atomic operations can severely impact the memory access performance. To quantify the performance impacts of using atomic operations in MapReduce implementations on an AMD Radeon HD 5870 GPU. We first implement a basic OpenCL MapReduce framework based on Mars. In its original design, Mars uses preprocessing kernels, i.e., counting and prefix summing kernels, to calculate the locations of output records in global memory for each thread. We add a feature that allows threads in different wavefronts to use atomic operations (instead of using preprocessing kernels) to compute the output locations.

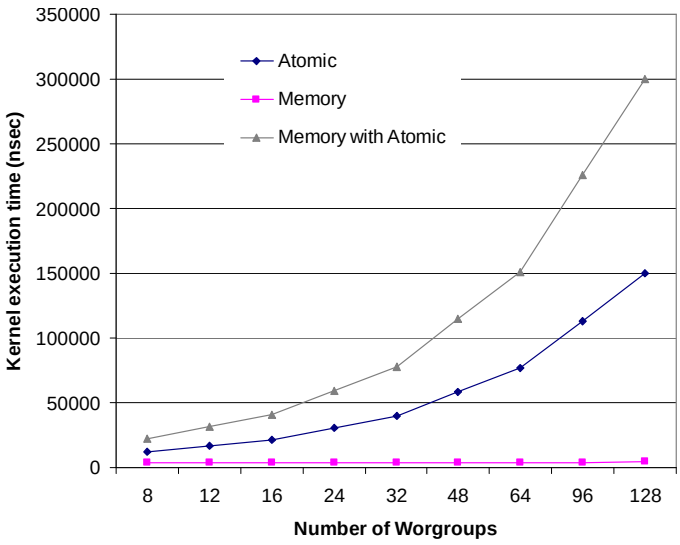

Figure 3.2: Kernel execution time of the simple copy kernel

We compare the performance of the basic OpenCL MapReduce implementation (named Mars) and the atomic-based implementation (named AtomicMR), focusing on the execution time of two MapReduce applications: Matrix Multiplication (MM) and KMeans (KM). Specifically, we run MM for matrix sizes of 256 X 256, 512 X 512, and 1024 X 1024, and KM for number of points 4K, 16K, 64K. As shown in Figure 3.3 and Figure 3.4, the performance of atomic-based MapReduce framework is significantly worse than Mars. More specifically, the average slowdown is 28-fold and 11.3-fold for Matrix Multiplication and KMeans, respectively. These results suggest that atomic-based MapReduce implementations are not suitable for AMD Radeon HD 5000 series.

It is worth noting that, our atomic-based implementation uses atomic operations at the granularity of a wavefront, i.e., one master thread in the wavefront is responsible for allocating more buffer for all threads in this wavefront. In KMeans and Matrix Multiplication, each map thread writes to the global buffer once, so atomic operation is called once per wavefront by a master thread. This implementation using atomics at the wavefront level fairly mimics the map phase of the MapCG[17] implementation.

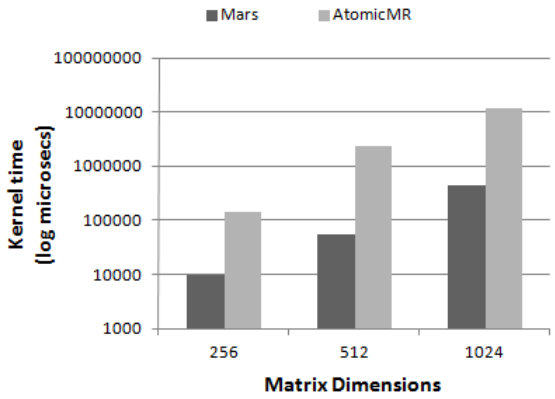

Figure 3.3: Performance of atomic-based MapReduce vs. Mars using Matrix Multiplication

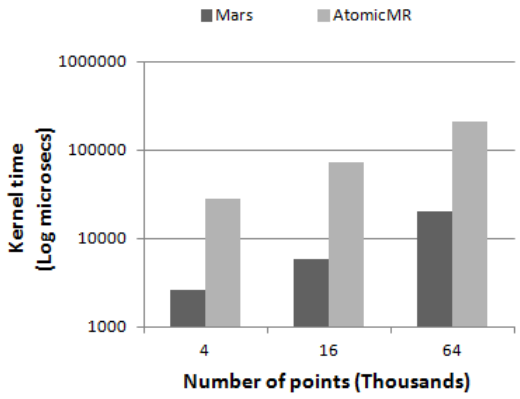

Figure 3.4: Performance of atomic-based MapReduce vs. Mars using KMeans

3.3 Software-Based Atomic Add

Implementing atomic add on GPUs is tricky because of the lack of efficient synchronization primitives on GPUs. One straightforward approach uses a master-slave model to coordinate concurrent updates at the granularity of threads. As shown in Figure 3.5, three arrays, i.e., *address array*, *increment array*, and *shared variable array*, are maintained in global memory. Each thread executing the software atomic add operation writes the increment values to a shared variable to the increment array and the address of the shared variable to the address array. Note that storing the address of a shared variable enables support for multiple shared variables in a kernel. A dedicated master/coordinator thread, which can be run in a separate workgroup, continuously spins on the address array. Once the master thread detects any thread executing the atomic operation, it updates the corresponding shared variable using the address and the increment value stored. Once the update is finished, the master thread resets the corresponding element of the address array to 0, signaling the waiting thread, busy waits on its corresponding element until the update is finished. Since only one thread is doing the update, the atomicity is guaranteed.

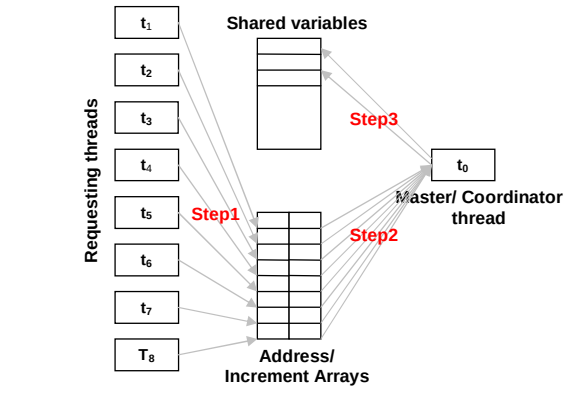

Figure 3.5: High level illustration of handling the software atomic operation

However, in this basic implementation described above, the master thread can easily become a performance bottleneck because of the serialization of update calculation as well as the excess number of global memory accesses. In addition, maintaining one element per thread in the address and increment arrays can incur space overhead for a large number of threads. To address these issues, we introduce a hierarchal design that performs coordination at the level of wavefronts and parallelizes the update calculation across the current threads executing the software atomic add. Specifically, the increment array maintains one element per wavefront, so does the address array. Each wavefront first calculates a local sum of the increment values requested by the participant threads in the fast local memory,[1] then it stores the local sum to the increment array in the global memory. The first workgroup is reserved as the coordinator workgroup. Threads in the coordinator workgroup read the address and increment arrays in parallel and collaboratively calculate the update value. Note that the coordinator workgroup does not participate in the kernel computation, otherwise deadlocks may occur when threads diverge in the coordinator group. Such a hierarchical design can greatly reduce global memory transactions as well as parallelize the update computation.

One challenge in the hierarchal design is to support divergent kernels, in which case not all threads participate in the software atomic add. In this case, care must be taken to avoid potential deadlocks and race conditions. As we will explain in Section 3.4, we use system-provided atomic operations on local variables to coordinate between threads within a wavefront, leveraging the fact that atomic operations on local variables will not force memory access to take the CompletePath.

To guarantee that the coordinator will always be executed, our current implementation assumes that the number of workgroups used in the kernel does not exceed the maximum number of concurrently running workgroups. For the Radeon HD 5870, we have found that for a simple kernel, each compute unit (of the 20 compute units) can run up to seven

[1]Local memory in OpenCL is equivalent to shared memory in CUDA.

workgroups, so the maximum number of workgroups supported by our implementation in this case is 140. This value can be easily calculated following a similar methodology to the one proposed by the CUDA occupancy calculator [21]. While we leave support for an arbitrary number of workgroups for future work, the current design is useful in practice by adopting a large number of threads.

3.4 Implementation Details

By default, the atomic add operation returns the old value of the global variable just before executing the atomic operation. To support this feature, in addition to the hierarchical design described above, an old value of the shared variable is returned to each wavefront, which then calculates a return value for each participating thread with backtracking.

3.4.1 Data Structures

Four global arrays are used in our implementation. The number of elements of each array equals the number of wavefronts of the kernel, so each wavefront reads or writes to its corresponding element of these arrays. The first array is the *WavefrontsAddresses* array; whenever a wavefront executes an atomic operation to a shared variable, it writes the address of this variable to its corresponding element in this array. The second array is the *WavefrontsSums* array, which holds the increment of every wavefront to the shared variable. The third array is the *WavefrontsPrefixsums* array, which contains the old value of the global variable just before executing the atomic operation and is used by the requesting wavefront to generate the return value from the atomic add operation, i.e., to mimic the system-provided atomic. The final array is the *Finished* array. Whenever a wavefront finishes its execution, it sets its corresponding element of this array to one.

3.4.2 Requesting Wavefront

Any thread executing our software-based atomic add operation passes through four steps, as shown in Figure 3.6. In the first step, the thread collaborates with other threads concurrently executing the atomic add operation to safely increment the wavefront's increment using local atomic add (line 13, 15, 16, and 18). In the second step, only one thread called the *dominant thread* writes the increment and address of the shared variable to the global memory (lines 22-26), i.e., *WavefrontsSums*, and *WavefrontsAddresses*, respectively. Since threads of any wavefront may diverge, the atomic operation may not be executed by all threads in the wavefront. Consequently, instead of fixing the first thread of the wavefront to write to the global memory, the first thread executing the local atomic add operation is chosen to be the *dominant thread* (line 14, 15, 17, and 18). In the third step, the thread waits until the coordinator workgroup handles the atomic operation and resets the corresponding element of the *WavefrontssAddresses* array (lines 29-32). Once this is done, the *WavefrontsPrefixsums* array contains the prefix sum of this wavefront, and every thread in the wavefront then generates its prefix sum and returns (line 36).

3.4.3 Coordinator Workgroup

For convenience, the functionality of the coordinator workgroup is described assuming the number of wavefronts of the kernel equals the number of threads of the coordinator workgroup. However, the proposed atomic operation handles any number of wavefronts that is less than or equals the maximum number of concurrent wavefronts. Each thread of the coordinator workgroup is responsible for handling atomic operations executed by a specific wavefront. All threads in the coordinator group keep executing four consequent steps until all other wavefronts are done.

As shown in Figure 3.7, in the first step (lines 16-19), each thread loads the status of its

```
1  int  software_atom_add(__global int *X,  int Y,
2  __local int *LocalSum, __local int *ThreadsNum,
3  __global int *WavefrontsAddresses,
4  __global int *WavefrontsSum,
5  __global int *WavefrontsPrefixsum)
6  {
7     //Get the wavefront global and local ID
8    int wid = get_global_id(0) >> 6;
9    int localwid = get_local_id(0) >> 6;
10
11    /* Safely incrementing the wavefront increment and
12       threads number */
13    LocalSum [localwid]  =  0;
14    ThreadsNum [localwid]  =  0;
15    mem_fence (CLK_LOCAL_MEM_FENCE);
16    int threadSum  =  atom_add(&LocalSum [localwid],Y);
17    int virtualLid  =  atom_inc(&ThreadsNum [localwid]);
18    mem_fence(CLK_LOCAL_MEM_FENCE);
19
20    /* The first thread only writes the sum back to the
21       global memory */
22    if (virtualLid  ==  0)      {
23         WavefrontsSum[wid]  =  LocalSum[localwid];
24         WavefrontsAddresses [wid]  =  X;
25     }
26     mem_fence(CLK_GLOBAL_MEM_FENCE);
27
28   //Wait until the coordinator handles this wavefront
29     while(1)  {
30         mem_fence(CLK_GLOBAL_MEM_FENCE);
31         if (WavefrontsAddresses [wid]  ==  0) break;
32     }
33
34    /* Generate the retrun value and re-initialize the
35       variables */
36    int  ret =  WavefrontsPrefixSum[wid]  +  threadSum;
37    if (virtualLid  ==  0) {
38       LocalSum [localwid]  =  0;
39       ThreadsNum [localwid]  =  0;
40       mem_fence(CLK_LOCAL_MEM_FENCE);
41     }
42     return ret;
43 }
```

Figure 3.6: Code snapshot of software atomic add operation

wavefront into the local memory. The thread *lid* reads the status of the wavefront *lid*. More specifically, it reads *WavefrontsAddresses[lid]*, and *WavefrontsSums[lid]* and stores these variables into the local memory, i.e., *Address[lid]* and *LocalSum[lid]*, respectively, as shown in lines 16, and 17. All threads are then synchronized (line 19) before the next step to ensure that the status of all wavefronts have been loaded.

In the second step (lines 23-36), the prefix sum of each wavefront and the increment of each unique address are generated. Each thread *lid* checks whether the wavefront *lid* executes the atomic operation or not by examining the address *Address[lid]* (line 23). If it is the only wavefront executing atomic operation to this address, the prefix sum is simply the value of this address (line 34), and the increment is the wavefront's increment represented by *LocalSum[lid]*. If there are several wavefronts concurrently executing atomic add for this address, the prefix sum of each wavefront and the increment of this address are generated using local atomic add operation, i.e., atomic add to a local memory variable (lines 25-33). Note that the increment of the first of these wavefronts called *dominant wavefront* holds the increment of this address and the other wavefronts increments are set to zero (line 29) to ensure correctly incrementing the shared variable. All threads are again synchronized (line 36) to ensure that the increments of all wavefronts are used to calculate the increments of the global variables.

In the third step (lines 40-46), the global variables are safely updated and the blocked wavefronts are released. Specifically, each thread *lid* checks whether the wavefront *lid* executes the atomic operation or not by examining the address *Address[lid]* again (line 40). If it is a requesting wavefront, the thread *lid* sets *WavefrontsAddresses[lid]* to zero (line 44) to release this wavefront. If it is a *dominant wavefront*, its global variable is safely updated (line 41). Also, the local address and increment of this workgroup are reinitialized (line 42, and 43).

Finally, each thread re-evaluates the termination condition by calculating the number of the finished wavefronts (lines 50-54). If all wavefronts are done, the thread terminates.

```
1  void AtomicCoordinator(__local int *Address,
2  __local int *LocalSums,
3  __global int *WavefrontsAddresses,
4  __global int *WavefrontsSums,
5  __global int *WavefrontsPrefixsums,
6  __global int *Finished)
7  {
8   //Get thread ID in workgroup, and number of wavefronts
9    int lid = get_local_id(0);
10   int wavefrontsPerWorkgroup = get_local_size(0) >> 6;
11   int wavefrontsNum = get_num_groups(0) *
12                       wavefrontsPerWorkgroup;
13
14   while (1)    {
15     //1- Read the status of the wavefronts
16     Address[lid]   =  WavefrontsAddresses [lid];
17     LocalSum[lid]  =  WavefrontsSums[lid];
18     __global int * X  =  (__global int*)Address[lid];
19     barrier(CLK_LOCAL_MEM_FENCE);
20
21     /* 2- Safely generate the wavefronts prefixsums and
22           the increment of each unique variable     */
23    if ((lid < wavefrontsNum) && (Address[lid] > 0 )){
24     int  replaced   =   0;
25     for (int  k = 1;  k < lid ;  k++){
26      if (Address[lid]   ==   Address[k]) {
27       int temp = atom_add(&LocalSum[k], LocalSum[lid]);
28       WavefrontsPrefixSum[lid]  =  *X +  temp;
29       LocalSum[lid]  =  0;
30       Replaced  =  1;
31       break;
32      }
33     }
34     if (replaced == 0) WavefrontsPrefixsum[lid]  = *X;
35    }
36    barrier(CLK_LOCAL_MEM_FENCE);
37
38    /* 3- Safely increment the global variable and
39         release the blocked wavefronts */
40    if ( Address[lid]  >  0) {
41       if ( LocalSum[lid] > 0) *X  +=  LocalSum[lid];
42       Address[lid]   =   0;
43       LocalSum[lid]   =   0;
44       WavefrontsAddresses [lid]   =   0;
45    }
46    mem_fence(CLK_GLOBAL_MEM_FENCE);
47
48    //4- Check for exiting
49    int count  =  0;
50    for(int i = wavefrontsPerWorkgroup; i <
51           wavefrontsNum;  i++)
52              if (Finished[i]  ==  1) count++;
53    if (count == wavefrontsNum - wavefrontsPerWorkgroup)
54              break; //All wavefronts are done
55  }
56 }
```

Figure 3.7: Code snapshot of coordinator workgroup function

3.4.4 Discussion

We have taken great care in our design to ensure its correctness. Within a requesting wavefront (Figure 3.6), one design challenge is to select the dominant thread in divergent kernels. Since all threads within a wavefront are executed in a lock-step manner, using atom_inc on a variable in local memory can guarantee only one thread is chosen as the dominant thread. Our implementation also maintains separate local sums for different wavefronts; if a local sum is shared between wavefronts, a race condition can occur when threads from different wavefronts try to update the same local sum.

Another design challenge is to ensure that data is correctly exchanged between different workgroups. According to [88] and [75], the correctness of implementing a GPU primitive that requires inter-workgroup communication cannot be guaranteed until a consistency model is assumed. Xiao et al. [75] solved that by using __threadfence() function that ensures the writes to global memory by any thread is visible to threads in other blocks (i.e., workgroup in OpenCL). OpenCL does not have an equivalent to the __threadfence function. The mem_fence function in OpenCL only ensures that the write of a thread is visible to threads within the same workgroup. Fortunately, mem_fence guarantees the order that the memory operations are committed [31]. That means, for two consecutive memory operations A and B issued by a thread to a variable in the global memory, if mem_fence is called between them, once B is visible to threads in other workgroups, A will be visible as well because A is committed to the global memory before B. The correctness of our implementation in data exchange between different workgroups is achieved by the memory consistency provided by mem_fence.

Finally, although our implementation allows different wavefronts to concurrently execute atomic operation to different variables, threads within the same wavefront should concurrently execute the atomic operation to the same variable, since the status of each wavefront is represented by only one element in the global arrays. We believe that this requirement

can be satisfied by restructuring the code and utilizing the shared memory.

3.5 Model for Speedup

In this section, we derive a model representing the speedup of our software-based atomic over the system-provided atomic for both divergent and non-divergent kernels. For simplicity, this model assumes that there is only one wavefront per workgroup.

In general, any GPU kernel involves three main steps; reading the input data from the global memory, doing some computations, and writing the results back to the global memory. The first and third steps are memory accesses, the second step can be divided into general computations and atomic-based computations. So the total execution time of atomic-based kernels is composed mainly of three components: memory access time, atomic execution time, and computation time. The software-based atomic operation affects only the first and second terms. The total execution time can be represented as:

$$T = t_m + \sum_{i=1}^{n} t_{ai} + t_c \tag{3.1}$$

Where t_m is the memory access time, t_a is the atomic execution time, n is the number of calls to atomic operation in the kernel, and t_c is the computation time.

For simplicity, we ignore the possible overlapping between the computation and memory accesses. The memory access time depends on the used path whether i.e., complete path or fast path, however the atomic execution time depends on the threads divergence. Assuming the kernel executes c, and f memory transactions through the complete and fast path respectively, and time to execute each transaction is t_{comp} or t_{fast} for CompletePath and FastPath, respectively. Then equation 3.1 can be represented as:

$$T = (c.\ t_{comp} + f.\ t_{fast}) + \sum_{i=1}^{n} t_{ai} + t_c \tag{3.2}$$

When our software-based atomic is used instead of the system-provided atomic, the first and the second term of equation 3.2 are affected. For the first term, all of the memory accesses except for stores of non-32 bit data are executed through the fast path. Then the memory access time becomes $(c + f + o).t_{fast}$, where o is the extra memory transactions executed by the requesting and coordinator workgroups. So the memory access speedup can be represented as:

$$\begin{aligned} S_m &= \frac{c.\ t_{comp}+f.\ t_{fast}}{(c+f+o).t_{fast}} \\ &= \frac{c.\ x.\ t_{fast}+f.\ t_{fast}}{(c+f+o).t_{fast}} \\ &= \frac{c.\ x+f}{c+f+o} \end{aligned} \tag{3.3}$$

Where S_m is the memory access speedup, x = $\frac{t_{comp}}{t_{fast}}$ is the speedup of a single memory access when using fast path relative to the complete path. Since including a single system-provided atomic in the code may force most of the memory accesses to follow the complete path, so f is very small compared to c and can be removed from equation 3.3, so S_m becomes $\frac{x}{1+\frac{o}{c}}$. Since x is significantly larger than one [8], then if o is less than c, the memory access speedup using our software atomic becomes significantly larger than one. For memory-bound applications, where the memory access time represents a major factor of the total execution time, using our atomic can significantly speedup the performance as the experiments have shown.

To derive the speedup of atomic operations, we need to consider the details of handling atomic operations using system-based and software-based approaches. Executing one system-provided atomic operation concurrently by several threads is done serially, and hence requires

$N \cdot t_1$, where N is the number of threads concurrently executing the atomic operation. For non-divergent kernels, N equals the total number of threads in the kernel. Moreover, t_1 is the time to modify a global variable through the CompletePath. By neglecting the computations embedded within the atomic operation, t_1 can be replaced with t_{comp}, where t_{comp} is the time to execute a memory transaction through the CompletePath. So, the time required to execute the system-provided atomic, $t_{a_{system}}$, can be represented as:

$$t_{a_{system}} = N \cdot t_{comp} \tag{3.4}$$

Executing a software-based atomic operation can be represented by:

$$t_{a_{software}} = t_{RWGI} + t_{CWG} + t_{RWGP} \tag{3.5}$$

Where t_{RWGI} is the time needed for the requesting workgroup to generate its increment and updates the global arrays (section 3.4.2), T_{CWG} is the time required by the coordinator workgroup to generate the prefix sums and update the shared variables (section 3.4.3), and finally t_{RWGP} is the time needed by the requesting workgroup to generate the prefix sum and return from the atomic operation (section 3.4.2).

Since the wavefront's increment is calculated using atomic add operation to shared memory (lines 13-18 in Figure 3.6), then t_{RWGI} can be represented by $2 \cdot N_c \cdot t_l + 2 \cdot t_{fast}$, where N_c is the number of threads per workgroup concurrently executing the atomic; t_l is the time to modify a variable in the shared memory; and t_{fast} is the time to execute memory transaction through the FastPath. And $2 \cdot t_{fast}$ is the time for writing the address and the increment to the global arrays (line 22-26 in Figure 3.6). Moreover, t_{CWG} can be represented by $5 \cdot t_{fast} + N_{cwg} \cdot t_l + \frac{N_{wg}}{2} \cdot t_l$, the first term corresponds to reading the workgroups increments, and addresses, writing the prefix sums to the global memory, updating the workgroup's address and shared variable. The second term corresponds to the time needed to generate the global increment using atomic add operation to the shared memory (line 27 in Figure

3.7), where N_{cwg} is the number of concurrent workgroups executing the atomic operation. The third term is time needed to check the value of local addresses (lines 25 and 26 in Figure 3.7), where $\frac{N_{wg}}{2}$ is the average number of comparisons until reaching the *dominant wavefront*. Finally, t_{RWGP} equals $2 \cdot t_{fast}$ because it requires only reading the address and the prefix sum from the global memory.

From the above discussion:

$$t_{a_{software}} = ((2 \cdot N_c + N_{cwg} + \frac{N_{wg}}{2}) \cdot x_l + 9) \cdot t_{fast} \tag{3.6}$$

where $x_l = \frac{t_l}{t_{fast}}$ and it is less than one by definition. For non-divergent kernels, we can substitute N in equation 3.4 with $N_c \cdot N_{cwg}$ and t_{comp} by $x \cdot t_{fast}$,where x = $\frac{t_{comp}}{t_{fast}}$ is the speedup of a single memory access when using FastPath relative to the CompletePath. Then $t_{a_{system}}$ can be represented by:

$$t_{a_{system}} = N_c \cdot N_{cwg} \cdot x \cdot t_{fast} \tag{3.7}$$

By comparing equation 3.6 by equation 3.7, we can see that the atomic operations speedup $\frac{t_{a_{system}}}{t_{a_{software}}}$ increases significantly as the number of workgroups increases. Furthermore, for divergent kernels, the speedup is smaller than that of non-divergent kernels, because $t_{a_{system}}$ is proportion to the number of threads concurrently executing the atomic operation, but $t_{a_{software}}$ remains almost the same.

3.6 Evaluation

All of the experiments are conducted on a 64-bit server with Intel Xeon e5405 x2 CPU and 3GB RAM. The attached GPU device is ATI Radeon HD 5870(Cypress) with 512MB of

device memory. The server is running the GNU/Linux operating system with kernel version 2.6.28-19. The test applications are implemented using OpenCL 1.1 and built with AMD APP SDK v2.4.

In all experiments, three performance measures are collected. The first is the total execution time in nano-seconds. The second is the ratio of FastPath to CompletePath memory transactions, and the third is the ALU:Fetch ratio that indicates whether the kernel is memory-bound or compute-bound. Stream kernel analyzer 1.7 is used to get the second and third metrics. For the second metric, the equivalent ISA code of the OpenCL kernel is generated, then all memory transaction are counted. MEM_RAT, and MEM_RAT_CACHELESS transactions are considered as CompletePath and FastPath transactions respectively [8]. Note that these metrics do not capture runtime information. For instance, the absolute numbers of memory transactions following different paths are not revealed by these metrics. Each run is conducted using 64, 128, and 256 threads per workgroup, and the best performance is used to generate the graphs.

We conduct two sets of experiments to evaluate the performance of our software-based atomic. The first set uses micro benchmarks to generally show the benefits of using the software-based atomic on AMD GPU. And the second set evaluates the performance impact of using atomic operations in MapReduce design. To achieve this, we first implement a baseline MapReduce framework based on Mars. We then implement a single-pass output writing design using atomic operations. The atomic operations are applied in both the thread level and the wavefront level.

3.6.1 Micro Benchmarks

The first micro benchmark aims at identifying the overhead of executing the system-provided atomic operation. The code of this microbenchmark is simple. Each thread only executes

the atomic operation to increment a global variable by the global index of the thread. The kernel does not include any memory transaction, for our goal is to measure the overhead of executing the atomic operation by itself.

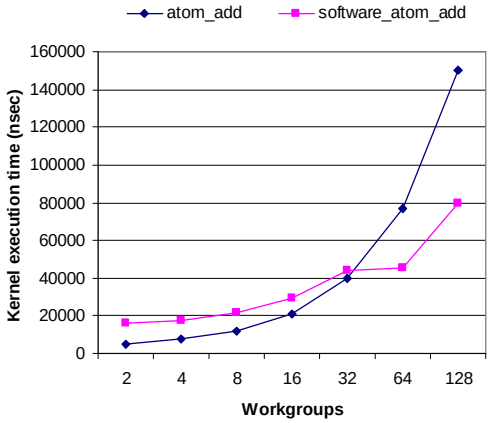

Figure 3.8: The execution time of system and software-based atomic

As shown in Figure 3.8, for small numbers of workgroups, (e.g., less than 32 workgroups), the performance of our software-based atomic is slower than the system-provided atomic by 0.5 fold on the average. As the number of workgroups increases, the speedup of our atomic increases until reaching 1.9 folds for 128 workgroups. This can be explained by the model discussed in Section 3.5. As indicated in equations 3.6 and 3.7, the execution time of the system atomic operation increases linearly with the number of concurrent threads. However, the execution time of the software-based atomic is proportional to the number of concurrent wavefronts. Consequently, as the number of workgroups increases, our atomic add implementation can significantly outperforms the system one.

The second micro benchmark aims at studying the impact of atomic operations on the performance of the memory transactions. The code of this micro benchmark looks very similar to the previous one, with another memory instruction being added.

As shown in Figure 3.9, the speedup of our atomic add implementation with regard to the system-provided atomic add operation increases significantly as the number of workgroups

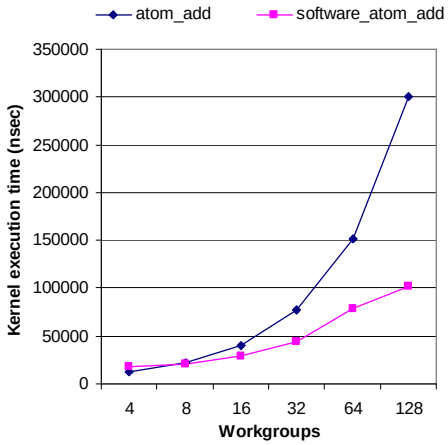

Figure 3.9: The execution time of system and software-based atomic when associated with memory transactions

grows. This is due to that the performance of CompletePath is much worse than FastPath. Although our atomic add implementation performs more loads and stores to global memory compared to the system atomic add. Also, the ratio of complete to FastPath transactions is 3:0 and 0:10 for the system-provided atomic add and our software-based atomic add, respectively.

3.6.2 MapReduce

We use three test applications that are commonly used in other MapReduce studies such as Mars and MapCG. These applications involve both variable and fixed sized output. in addition one of them execute only the map phase and the others executes both map and reduce phases. These applications include:

- **Matrix Multiplication (MM)**. MM accepts two matrices X and Y as input and outputs matrix Z. Each element $z_{i,j}$ in Z is produced by multiplying every element in row i of X with the corresponding element in column j of Y and summing these

products. The MapReduce implementation of MM includes only the map phase, where each map task is responsible for calculating one element of the output matrix. Since all map threads access the same number of elements of X and Y and executes the same number of operations, then matrix multiplication is an example of non-divergent kernels whose threads execute the atomic operation at the same time.

- **String Match (SM)** SM searches an input keyword in a given document and outputs all matching locations. The MapReduce implementation of SM includes only the map phases. Each map task reads a chunk of the input document, character by character, and outputs the locations of any found matching words. String match is an example of divergent kernels.

- **KMeans (KM)**: KM is an iterative clustering algorithm. Each iteration takes a set of input points and a set of clusters, assigns each point to a closest cluster based on the distance between the point and the centroid of the cluster, and recalculates the clusters after. The iteration is repeated until clustering results converge (In our results we run only one iteration). The MapReduce implementation of KM include both map and reduce phases. The map function attaches the assigned points to their closest clusters, and the reduce function calculates the new coordinates of a cluster based on the attached points. Note that, KMeans also is an example of non-divergent kernels whose threads execute the atomic operation at the same time.

Matrix multiplication performance is shown in Figure 3.10. As we can see, the speedup of using software-based atomic add over the system atomic add increases as the input matrices get larger. Specifically, the speedup improves from 0.62 folds for a 8X8 input to 13.55 folds for a 256X256 input. The main reason is that for larger inputs, there will be more memory access, exacerbating the memory performance of using CompletePath. By analyzing the ISA, we realize that the ratio of the FastPath to CompletePath memory accesses is 30:0 and

3:28 for software-based atomic and system-provided atomic implementations, respectively.

Note that, since the number of workgroups is constrained by the maximum number of concurrent workgroups, for matrices of dimensions greater than 64X64, every thread manipulates several elements in the output matrix instead of one element.

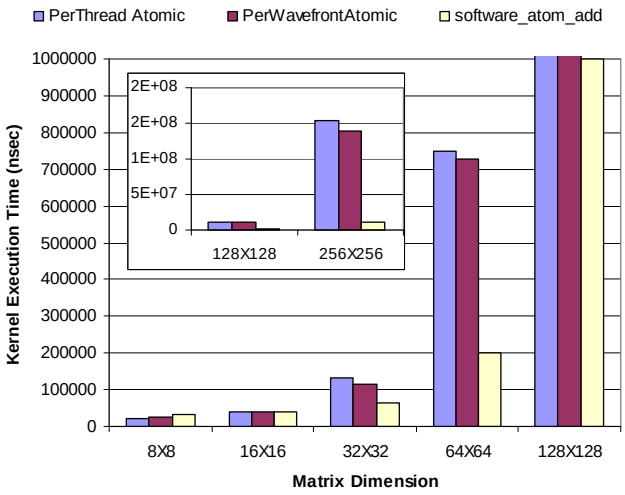

Figure 3.10: The execution time of Matrix multiplication using system and software-based atomic operation

For String Match, we run String match using a dataset of size 4 MB [66] to search different keywords. For each keyword, we vary the number of workgroups from 32 to 128. As shown in Figure 3.11, the performance of the software-based atomic is better than that of the system-provided atomic in almost all cases for the first three queries. More specifically, the average speedup is 1.48 folds.

Two reasons contribute to this small speedup compared to the other MapReduce applications; first, measuring the ALU:Fetch ratio indicates that this application is compute-bound since the ratio is highly greater than one i.e., 17.55. Second, string match is an example of divergent kernels where the atomic operation speedup is not significant as shown by the model in section 3.5.

Note that, for the fourth query, the performance of our atomic is significantly worse than the system-provided atomic. This query returns significantly higher number of matches compared to the other queries. Specifically, the number of matches is 7, 87, 1413, and 20234 for first, second, third, and fourth query respectively. A larger number of matches requires more memory transactions to write the matches as well as more computations. We realize that writing the matches are done through the FastPath even when system-provided atomic is used, so increasing the number of matches only contributes to increase of the compute-boundness of the application. Note that the number of read operations is the same for four queries. In other words, the software atomic approach does help improve the memory read performance, thus we observe performance improvements for the first and second queries with less computation. For the fourth query, with more amounts of computation, the overhead incurred by the software atomic approach for writing results start to offset the benefit of using FastPath for read accesses.

By analyzing the ISA of both kernels using the software-based atomic and the system-provided atomic, we realize that the ratio of FastPath to CompletePath memory accesses is 12:0 and 1:19 for the software-based atomic and the system-provided atomic, respectively. This result also reveals one important fact that is not explicitly mentioned in the AMD OpenCL Guide [8]; although in [8], they mentioned that non-32 bits memory transaction are executed through the CompletePath, in the kernel that uses the software-based atomic, all transactions are executed through the FastPath although input file is read character by character. In-depth mapping of OpenCL kernel instructions to ISA instructions have shown that only stores of char are executed through the CompletePath (loads of char are executed through the FastPath).

For KMeans, we run it for different number of points ranging from 512 to 8192. As shown in Figure 3.12, the speedup gets improved from 15.52 folds for 512 points to 67.3 folds for 8192 points. Again, this is because of there are more memory accesses for larger inputs,

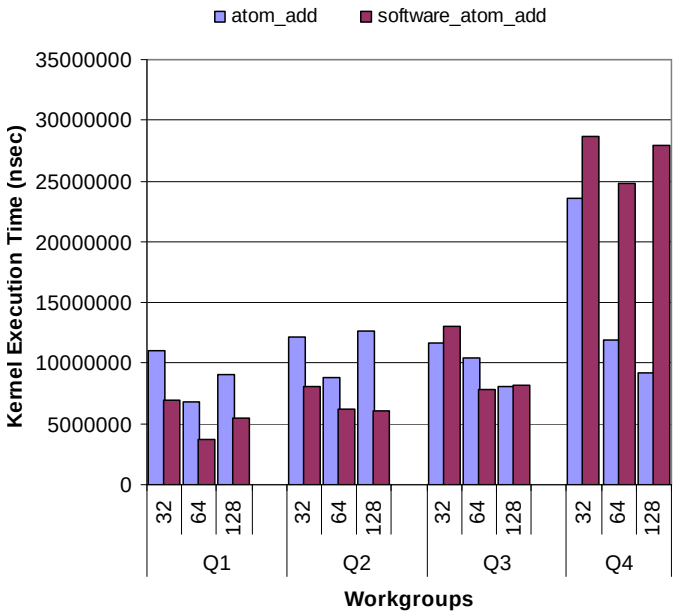

Figure 3.11: The execution time of string match using system and software-based atomic

amortizing the overhead of the software atomic add.

Although the performance of executing the system-provided atomic operation may be better than the performance of the software-based atomic in case of divergence as illustrated by the model in section 3.5, the results show that the performance of per thread atomic implementation of MapReduce almost equals the per wavefront atomic implementation for both matrix multiplication and KMeans. This returns to the high number of memory accesses in theses kernels which makes the execution time of this atomic operations insignificant compared to the memory access time.

3.7 Chapter Summary

In this chapter, we first quantify the effects of using the system-provided atomic operations on the performance of simple kernels and MapReduce implementations using AMD GPUs.

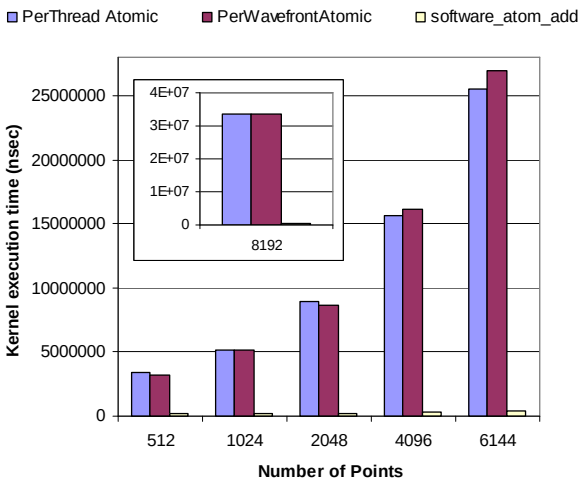

Figure 3.12: The execution time of map phase of KMeans using system and software-based atomic operation

Then we propose a novel software-based atomic operation that can significantly improve the performance of memory-bound kernels. Using this software-based atomic operation, we have developed an efficient MapReduce implementation for AMD GPUs. We evaluate this MapReduce framework using three applications that follow different divergence patterns and ALU:Fetch ratio. The experimental results show that for memory-bound kernels, our software-based atomic add can deliver an application kernel speedup of 67-fold compared to one with a system-provided atomic add. The main shortcoming of the proposed software-based atomic is that it supports limited number of workgroups. In the next chapter, we propose an atomic-free design for MapReduce that can efficiently handle applications running any number of workgroups.

This page intentionally left blank.

Chapter 4

StreamMR: An OpenCL MapReduce Framework for Heterogeneous Devices

4.1 Overview

To address the limitations of the MapReduce implementation introduced in Chapter 3, we propose StreamMR, an atomic-free MapReduce framework optimized for AMD GPUs. The design and mapping of StreamMR provides efficient atomic-free algorithms for coordinating output from different threads as well as storing and retrieving intermediate results via hash tables. StreamMR also includes efficient support of combiner functions, a feature widely used in cluster MapReduce implementations but not well explored in previous GPU MapReduce implementations.

Using OpenCL, StreamMR can run on any OpenCL-enabled device. Although OpenCL offers code portability across heterogeneous devices, achieving performance portability remains a challenging problem due to the architectural differences between different devices. To cope with this problem, we propose other optimizations especially in the reduce phase,

for StreamMR to behave efficiently on other devices not suffering from penalties of using atomic operations. As a proof of concept, we evaluate the optimized version of StreamMR on NVIDIA GPUs, which shows StreamMR efficiency compared to state-of-the-art MapReduce implementations. The resulting MapReduce implementation is an efficient and portable framework across heterogeneous devices; such framework can be viewed as the initial groundwork towards achieving our ultimate goal.

The rest of this chapter is organized as follows: In Section 4.2 and 4.3, we present the design and implementation details of StreamMR. We discuss the optimization mechanisms in Section 4.4 and 4.6. In Section 4.5, we provide details about the APIs exposed by our implementation. We present the results and discussions in Section 4.7. Finally, Section 4.8 concludes the chapter.

4.2 Design Overview

In light of the discussion in the previous chapter, our MapReduce implementation should completely avoid the use of atomic operations to ensure efficient memory access, through the FastPath, on AMD GPUs. We also believe that being atomic-free improves the scalability of the framework with the still increasing number of cores per device, and thus, beneficial for NVIDIA GPUs as well.

Specifically, there are two major design issues in a MapReduce runtime framework on GPUs: 1) how to efficiently and correctly write output from the large number of threads to the global memory and 2) how to efficiently group intermediate results generated by the map function according to their keys.

4.2.1 Writing Output with Opportunistic Preprocessing

As mentioned above, using global atomic operations in the MapReduce framework can incur severe performance penalties on AMD GPUs. While Mars implementation does not employ atomic operations, it requires expensive preprocessing kernels to coordinate output from different threads to the global memory. In particular, the computation in the counting kernel is repeated in the actual compute (map or reduce) kernel; this redundant computation results in wasted compute resources.

StreamMR introduces a two-pass atomic-free algorithm that enables different threads to efficiently write their output to the global memory. Specifically, each workgroup maintains a separate output buffer in global memory. In the first pass, these output buffers are preallocated according to a user-defined size. Each workgroup independently writes the output to its own buffer without synchronizing with other workgroups. When the preallocated buffer is full, the compute kernel (map or reduce) switches to a counting procedure that only counts the sizes of different output records (without actually writing them), similar to the Mars design. In the second pass, an overflow buffer is allocated for the workgroups that use up their preallocated buffer in the first pass, using the sizes computed in the counting procedure. A separate kernel is then launched to handle the unwritten output of the first pass.

The StreamMR output design eliminates the need for global atomic operations. It can also greatly save the preprocessing overhead compared to Mars. For applications with output sizes that can be easily estimated, e.g., Matrix Multiplication and KMeans, the counting procedure and the second pass can be skipped altogether, yielding the most efficient execution. That is, the preprocessing only happens opportunistically. For applications with output sizes that are hard to predict, StreamMR saves the counting computation corresponding to preallocated buffers during the first pass, whereas Mars performs the redundant counting computation for all output. In addition, in StreamMR, we record the output size per workgroup as opposed

to recording output size per thread in Mars, thus improving the prefix summing performance (as fewer size records need to be dealt with in the prefix summing).

4.2.2 Grouping Intermediate Results with Atomic-Free Hash Tables

Like MapCG, StreamMR organizes the intermediate output generated by the map phase using hash tables. However, MapCG uses atomic operations on global variables, e.g., compare-and-swap, to implement the hash table, which will incur performance penalty caused by the slow CompletePath on AMD GPUs. To address this issue, StreamMR maintains one hash table per wavefront, thus removing the need of using global atomics to coordinate updates from different workgroups to the hash table. Also, as explained in the next section, StreamMR leverages the lock-step execution of threads in a wavefront as well as atomic operations on local variables (i.e., variables stored in the local memory) to implement safe concurrent updates to the hash table of each wavefront. During the reduce phase, a reduce thread reduces the intermediate output associated with a specific entry in all hash tables, i.e., hash tables of all wavefronts.

4.3 Implementation Details

In StreamMR, each workgroup maintains four global buffers as shown in Figure 4.1. Among these buffers, $Keys_i$ and $Values_i$ store keys and values of intermediate results. HT_i is the hash table of wavefront i. Figure 4.2 depicts the details of the hash table design. Each entry in the hash table contains two pointers to the head and tail of a linked list (hash bucket) stored in $KVList_i$. The head pointer is used to explore the elements stored in a hash bucket, and the tail pointer is used when appending a new element. Each element in

$KVList_i$ associates every key to its value, and it contains a pointer to the next element in the linked list.

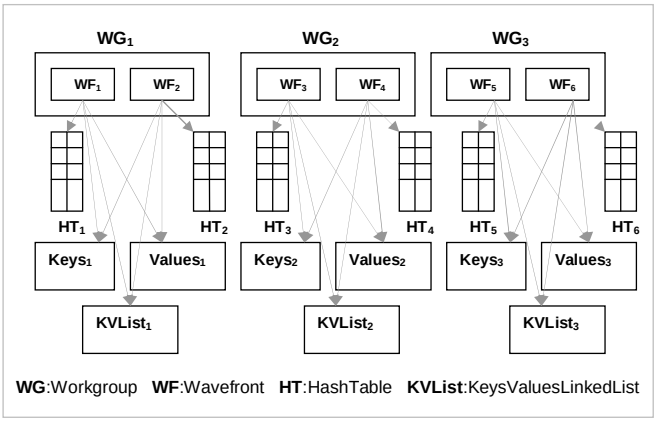

Figure 4.1: Main data structures used in the map phase of StreamMR

4.3.1 Map Phase

Initially, every map thread executes the map function on its assigned input key/value pair. A map thread then collaborates with other threads on the same workgroup i to determine its write location on the global buffers, i.e., $Keys_i$, $Values_i$, and $KVList_i$ without conflicting with other threads in the same workgroup. This can be efficiently done using the system-provided atomic operations on *local* variables, leveraging the fact that atomic operations on local variables does not force memory access to follow the CompletePath.

To safely update the hash table HT_i, a single entry of the hash table should be updated by only one thread in the workgroup, this thread is named *master thread*. Before the *master thread* updates the hash table, all threads in the workgroup should be synchronized. However, since the threads of the workgroup may diverge based on the input characteristics, deadlock can occur during the synchronization. To address this issue, we decide to use one hash table per wavefront, so all threads in a wavefront are synchronized by the lock-step execution.

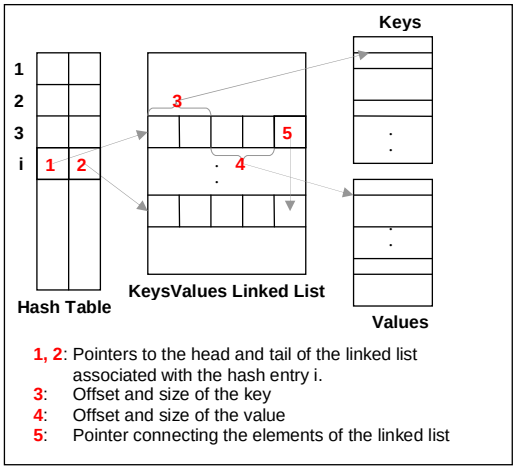

Figure 4.2: Details of the hash table

All threads of a wavefront use three auxiliary arrays stored in shared memory to coordinate concurrent updates to the hash table of this wavefront. The first array is *HashedKeys*. Thread i writes the hash of its key to its corresponding entry *HashedKeys[i]*. The second array is *Slaves*, which is used to identify the *master thread* of each hash entry. The third array *KeyValListId* is used by the master thread to update the links on the linked list associated with the hash entry. In updating the hash table, all threads in the wavefront go through three steps as shown in Figure 4.3. First, all active threads in the wavefront write the hash of their keys to the *HashedKeys* array and the index of the inserted record to $KVList_i$ to the *KeyValListId* array. Second, every thread reads the hash keys of all other threads, and the first thread with a certain hash key is considered as a *master thread*. For example, if thread t_1, t_3 and t_5 all have the same key, then t_1 will be marked as the *master thread*. Finally, the *master thread* t_1 reads the indices of its slave threads, i.e., *KeyValListId[3]*, and *KeyValListId[5]*, and then it updates the tail of the hash entry *HashedKeys[1]* to refer to the slave records, in addition to updating the links of these records to form the extended linked list as shown in Figure 4.2.

Note that, storing the map output into hash tables is only needed for applications with

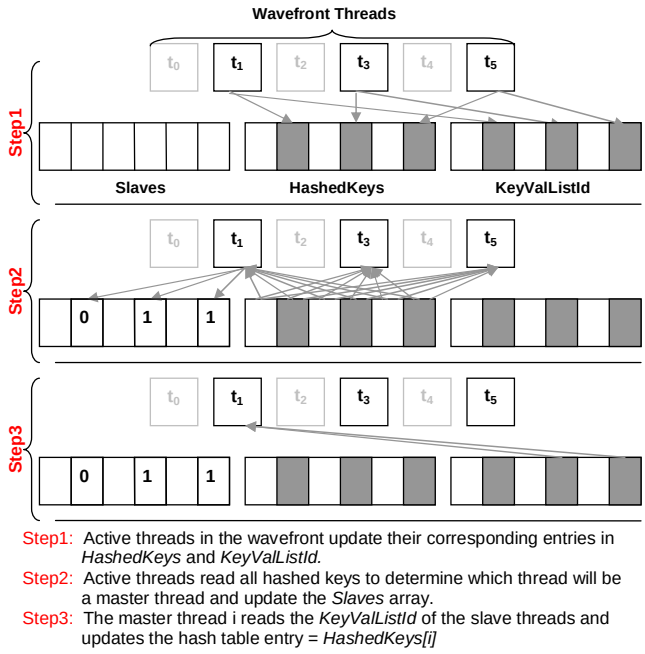

Figure 4.3: Steps for updating the hash table assuming wavefront of 6 threads, and t_1, t_3, and t_5 are the active threads

reduce phase. For applications with map only phase like Matrix Multiplication, the map output is written directly to the shared global buffer i.e., $Keys_i$, $Values_i$, and $KVList_i$. Specifically, threads in a workgroup collaborate using atomic operations to local variable for each thread to write in a separate location without incurring conflicts. This differs from MapCG [17] where writing to the hash table and launching of the reduce phase is required for all applications.

4.3.2 Reduce Phase

Reducing the key/value pairs from different hash tables is not a trivial task. Since a single hash entry may contain different keys for different hash tables as depicted in Figure 4.4(a), care must be taken to insure all keys are handled.

Specifically, a single reduce thread should be assigned to every unique key, to reduce the associated values. To identify this thread, we run a kernel (named *master identification* kernel) with number of threads equals the total number of entries of all hash tables i.e., number of entries per hash table × number of hash tables. Each thread examines a single hash entry in hash table *i*, passes through all associated keys, and compares every key to keys associated with the same hash entry in previous hash tables i.e., hash table *0 to i-1*. If an equivalent key exists in a previous hash table, this key is marked as a slave key, otherwise it is marked as a master key as shown in Figure 4.4(b). The reduce kernel is then launched with the same number of threads as the *master identification* kernel. Every thread handles a single hash entry, passes through all associated keys, only when the key is a master key, the thread reduces the values attached to this key in all subsequent hash tables. Finally, similar to the map phase, threads in the same workgroup collaborate using the system-provided atomic operations on local variables to write their final key/value pairs to the global buffers.

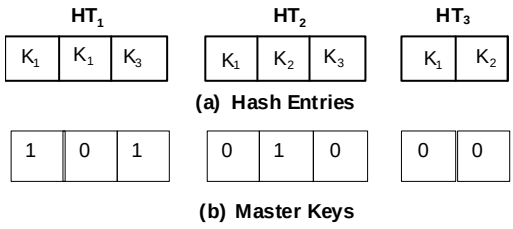

Figure 4.4: (a) Keys associated to a specific hash entry in three hash tables, and (b) the output of the master identification kernel

For applications with perfect hashing functions, the *master identification* kernel can be skipped. Thus the reduce kernel is directly invoked with number of threads equals the number of hash entries in one hash table. And every thread reduces the values of a specific hash entry in all hash tables. StreamMR can switch between the previous implementations according to the specification of the application provided by the programmer.

Experiments show that the above design incurs significant overheads for applications gener-

ating large number of intermediate key/value pairs like wordcount. However, these overheads are amortized by the efficient memory access through the FastPath on AMD GPUs. On other devices not suffering from memory access penalties, these overheads may not be amortized. So, we optimize the reduction algorithm as detailed in Section 4.3.2.1 to scale well with the size of the intermediate output and the number of wavefronts (hash tables) of the map phase, which is expected to increase with the input dataset size.

4.3.2.1 Scalability Improvement

To avoid passing back and forth through the hash tables generated from the map phase, we initially link all hash tables into a single master hash table. Consequently, we efficiently reduce the values attached to every unique key by passing only through the entries of the master hash table.

Basically, the optimized reduce phase executes two kernels. The first kernel (named *joining* kernel) joins all hash tables together into only one hash table (named *master* hash table). And the second kernel (named *reducing* kernel) applies the user-defined reduce function to every unique key attached to the *master* hash table.

In particular, the *joining* kernel is launched with number of threads equals the number of hash entries per hash table. Where every thread i is responsible for linking the key/value pairs attached to hash entry i of hash tables *1 to n* to the same hash entry of hash table *0* (*master* hash table). Note that, the complexity of this kernel is function of the number of hash tables generated from the map phase. To improve the load balancing and scalability of this kernel, we expose more parallelism, by separating it into two kernels. In the first kernel, each thread links the key/value pairs attached to a certain hash entry from only a specific number of hash tables. The second kernel is then launched to link all already linked pairs to hash table *0*. Experiments show that this two-level joining can significantly improve the

reducing performance by more than 6-fold speedup.

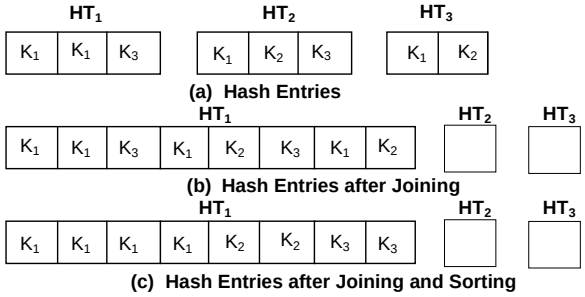

Figure 4.5: (a) Keys associated to a specific hash entry of three hash tables, (b) the output of the joining kernel, and (c) the output of the joining kernel when sorting is applied

For the *reducing* kernel, it is also launched with number of threads equals the number of hash entries per hash table. Every thread i reduces the values associated to every unique key within hash entry i of the master hash table. As shown in Figure 4.5(b), this requires passing through the linked list several times, one for every unique key. To mitigate this, we keep the keys sorted within every linked list while inserting them in the map phase and joining them in the reduce phase as shown in Figure 4.5(c) and explained in Section 4.4.3.

Experiments show that the above approach is more efficient than the redundancy-based approach described in our previous work [51], especially for applications generating large number of intermediate key/value pairs. We quantify the benefits from the optimized reduction design compared to the initial design in Section 4.7.

4.4 Optimizations

StreamMR provides several optimizations in addition to the basic design.

4.4.1 Map with Combiner

If the combiner function is available, the map phase can be modified so that instead of writing the map output directly to the global buffer, only one combined value is written per key. Specifically, the *master thread* generates the combined value of the slave threads, and updates the hash table accordingly. Since the map outputs are combined before being written to the global buffer, the number of global memory transactions can be significantly reduced.

In StreamMR, keys and values produced by the slave threads are written to the shared memory to improve the combining performance. For keys/values with variable sizes, the available shared memory may not be sufficient to hold the output from all threads in the memory. Upon such an overflow, the number of active threads per wavefront is reduced from 64 threads (in case of AMD Radeon HD 5870 GPU) to 32 threads. Threads from 0 to 31 continue their processing and threads from 32 to 64 remains idle. When the first half of threads complete their processing, the other half starts. While processing the active threads, the used sizes are compared to the allocated sizes. If the overflow occurs again, the number of active threads is reduced to 16 threads, and so on until the used sizes fit the available shared memory. The overhead of this mechanism will be evaluated in Section 4.7.

4.4.2 Reduce with Combiner

To further improve the reduce performance, the reduction can be applied on-the-fly. That is, instead of generating a list of values for every unique key, the combiner function can be applied directly to reduce every two values, thus avoiding the memory accesses required to generate and read the list of values. So instead of executing two *joining* kernels and one *reducing* kernel, combining is applied while linking the hash entries, thus avoiding the *reducing* kernel. Such a design allows more parallelism to be exploited during the reduction

because reducing a single hash entry is parallelized.

4.4.3 Optimized Hash Buckets

For efficient traversal of the linked list attached to each hash entry, within the *reducing* kernel, we maintain the list sorted while inserting the elements in the map phase and linking them in the reduce phase. This can significantly reduces the number of memory accesses especially for applications generating large number of intermediate outputs. Experiments have shown that this optimization can improve the reduce performance by more than 2.5-fold speedup.

To further improve the traversal performance; we maintain another pointer in the linked list to connect sublists of different keys. That is, instead of traversing every element in the linked list, only the first elements of these sublists are traversed.

4.4.4 Efficient Storing of Key/Value Sizes

We have realized that the storage requirement of the key/value sizes, shown in Figure 4.2, is linear with the number of key/value pairs. For applications with fixed key/value size, it is enough to store only one key/value size for all pairs. In StreamMR, we enable this feature according to the specification of the application, thus reducing the time required to store and retrieve the output pairs.

4.4.5 Image Memory Input

This optimization aims at improving memory access performance. When the input dataset is bound to the texture memory, the L1 and L2 texture caches can help reduce access to the

global memory. When the input dataset is heavily reused by the kernel, we have found that this optimization can significantly improve performance on AMD GPUs.

4.5 StreamMR APIs

StreamMR exposes few APIs, shown in Table 4.1, that are similar to the APIs offered by existing MapReduce implementations. Some of them are user-defined that need to be implemented by the programmer and two of them are provided by the framework to coordinate the writing of the output to the global buffer i.e., *emitIntermediate* and *emit*. Only when the combiner feature is enabled, *combine* and *combineSize* functions should be implemented.

4.6 Discussion

One limitation of using a separate buffer for each wavefront is that it can cause inefficient memory utilization, especially when the size of the initial buffer is too large. This limitation can be alleviated for applications with relatively predictable output sizes. The multi-buffer design may also cause inefficiency when the final output is copied back to the host memory. Assuming the allocated output buffers for all workgroups are stored in contiguous memory locations in the global memory, there are two options for transferring the final output back to the host memory. The first option is to copy only the used buffer from each workgroup. This requires multiple transfers i.e., one per workgroup. The second option is to copy all allocated buffers using only one transfer. In this case other unneeded buffers will be copied as well. Experiments have shown that the second option is more efficient, since it requires communicating with the host only once. However, the second option is still less perfect.

To cope with this limitation, we keep track of the size of the output generated by each workgroup, and allocate one contiguous buffer for all workgroups. A final kernel is then

User-defined functions
void map(__global void inputDataset, __global void* key, __global void* value)* `Applies the map function to an input key/value pair`
*void reduce(__global void * key, valueListItem * ValueList, __global char* interKeys, __global char* interValues)* `Reduces a list of values`
uint hash(void key, uint keySize)* `Returns the hashing of a given key`
*uint KeyEqual(void * key1, uint key1Size, void * key2, uint key2Size)* `Compares two keys. Returns 1 if they are equal, 2 if key1 > key2, and 3 if key2 > key1`
*void combine(void * value1, void* value2, int stage)* `Combines two values. Combines value2 into value1 if stage is 0, Initializes value1 by value2 if stage is 1, and applies postprocessing to value1 if stage is 2`
int combineSize() `Returns the expected size of the combined value`
System-provided functions
*void emitIntermediate(void * key, uint keySize, void * value,) uint valueSize)* `Emits an intermediate key/value pair`
*void emit(void * value, uint valueSize)* `Emits a final value`

Table 4.1: StreamMR APIs

launched to copy the output from the separate buffers into the contiguous buffers. Thus, only the useful output is copied back to the host. Experiments have shown that this approach provides the best performance.

4.7 Evaluation

In this section, we evaluate the performance of StreamMR against Mars and MapCG using four sets of experiments. In the first and second set, four representative applications are used to show the speedup over Mars and MapCG respectively. In the third set, the overheads of the overflow handling mechanisms i.e., global and local overflow are quantified. The effectiveness of using the Image memory is studied in the fourth experiment. Finally, we quantified the benefits of the scalable reduce design presented in Section 4.3.2.1.

4.7.1 Experimental Platform

All experiments were run on two servers - one equipped with AMD GPU and another one equipped with NVIDIA GPU. The first server is a 64-bit server with an Intel Xeon E5405 x2 CPU (2.00GHz) and 3GB of RAM. The equipped GPU is ATI Radeon HD 5870 (Cypress) with 1024MB of device memory. The server is running the GNU/Linux operating system with kernel version 2.6.28-19 and fglrx 8.84.5 GPU driver. The second server is a 64-bit server with an Intel Celeron E3300 x2 CPU (2.50GHz) and 2GB of RAM. The equipped GPU is NVIDIA Tesla C2050 (Fermi) with 3071MB of device memory. The server is running the GNU/Linux operating system with kernel version 2.6.32-5 and GPU driver version 285.05.33. All frameworks and testing applications are implemented with OpenCL 1.1 and compiled with AMD APP SDK v2.5.

4.7.2 Workloads

We use four test applications that are commonly used in other MapReduce studies such as Mars and MapCG. These applications involve both variable and fixed sized output, in addition two of them execute only the map phase and the others executes both map and reduce phases. These applications include:

- **Matrix Multiplication (MM)**. MM accepts two matrices X and Y as input and outputs matrix Z. Each element $z_{i,j}$ in Z is produced by multiplying every element in row i of X with the corresponding element in column j of Y and summing these products. The MapReduce implementation of MM includes only the map phase, where each map task is responsible for calculating one element of the output matrix.

- **String Match (SM)** SM searches an input keyword in a given document and outputs all matching locations. The MapReduce implementation of SM includes only the map phases. Each map task reads a chunk of the input document, character by character, and outputs the locations of any found matching words.

- **KMeans (KM)**: KM is an iterative clustering algorithm. Each iteration takes a set of input points and a set of clusters, assigns each point to a closest cluster based on the distance between the point and the centroid of the cluster, and recalculates the clusters after. The iteration is repeated until clustering results converge (In our results we run only one iteration). The MapReduce implementation of KM include both map and reduce phases. The map function attaches the assigned points to their closest clusters, and the reduce function calculates the new coordinates of a cluster based on the attached points. Note that the combiner function is enabled for both map and reduce phases in StreamMR in our experiments.

- **WordCount (WC)**: WC is commonly used to study the performance of MapReduce

implementation. It accepts an input file and outputs the number of occurrences of each word in this file. The MapReduce implementation of WC includes both map and reduce phases. The map function reads the assigned portion of the input file, and outputs one as the number of occurrences of every emitted word. The reduce function accepts the values of a specific word and outputs only one value representing the number of occurrences of this word in the whole file. Note that the combiner function is enabled for both map and reduce phases in StreamMR in our experiments. Also to have same hash collisions for the hash table of StreamMR and MapCG, we allocate the same number of hash entries.

For each one of the testing applications, we use three input datasets, i.e., Small (S), Medium (M) and Large (L) whose sizes are given in Table 4.2. The main performance metric is the total execution time, measured from the transformation of the input from host to device to copying the output back to the main memory. The speedup of X over Y is defined as the total execution time of Y divided by the total execution time of X. We repeat each run five times and report the average speedup when the variance of the runs is negligible, otherwise we report the confidence intervals in addition to the average speedup and repeat each run at least ten times. For each MapReduce framework, we try all possible workgroup sizes, and report the best results only. We also assume the size of the hash table is large enough to retain the characteristics of the hashing functions.

Applications	**Dataset Size**
$Wordcount(WC)$	S: 10MB, M: 40MB, L: 80MB
$MatrixMultiplication(MM)$	S: 256, M: 512, L:1024
$KMeans(KM)$	S: 8192 points, M: 32768, L: 131072
$StringMatch(SM)$	S: 16MB, M: 64MB, L: 100MB

Table 4.2: Dataset sizes per application

4.7.3 Comparison to Mars

We first evaluate the performance of StreamMR against Mars with four test applications. In order to execute the same implementation of Mars, which is originally implemented in CUDA, on AMD and NVIDIA GPUs, we have reimplemented Mars [1] with OpenCL. The bitonic sort and scan algorithms available in the AMD APP SDK are used to implement the sorting and scanning phases of Mars.

As shown in Figure 4.6 and Figure 4.8, StreamMR outperforms Mars for almost all testing applications with speedups between 0.9 to 3.5 for AMD GPU and between 1.1 to 10.0 for NVIDIA GPU. For applications with the map phase only, i.e. MM and SM, the advantage of StreamMR comes from the reduced preprocessing overhead (counting and prefix summing phases as detailed in Chapter 2). To better understand the performance gain of StreamMR over Mars, we break down the execution time of the large input dataset into five phases, i.e., preprocessing, map, group, reduce, and copy result (from GPU to CPU), as shown in Figure 4.7 and Figure 4.9. To get normalized times, the execution times of each phase is divided by the total execution time of the corresponding Mars run. For MM, the Mars preprocessing overhead is 5.7% and 4.9% of the total execution time in Mars for AMD GPU and NVIDIA GPU respectively. Since the output size is fixed, the preprocessing time of MM is negligible in StreamMR. As a consequence, StreamMR outperforms Mars by 1.02 and 1.14 times on the average for AMD GPU and NVIDIA GPU respectively. On the other side, in SM, since the size of the output is variable, Mars preprocessing phases, especially the counting phase consumes significant portion of the total execution time. Specifically, the counting phase passes through the whole file and searches for matches to accurately determine the size of the output of each map task. These preprocessing phases represent 49.5% and 40.1% of the total execution time of Mars on the average for AMD GPU and NVIDIA GPU respectively. So our framework better improves the performance by 1.86-fold

[1] Mars version 2 released on 10^{th} November 2009

and 1.58-fold speedup on the average for AMD GPU and NVIDIA GPU respectively.

For KM, as shown in Figure 4.7 and 4.9, although the overhead of Mars preprocessing kernels is small i.e., 5.8% of the total time, the speedup of our framework over Mars is high i.e., 2.53-fold and 2.66-fold speedup on the average for AMD and NVIDIA GPUs respectively. Particularly, the 95-confidence interval is [1.9,2.3], [2,2.2] and [2.9, 3.9] for small, medium and large datasets respectively for AMD GPU. This performance can be attributed to two reasons; first, the efficiency of the hashing-based grouping over sorting-based one which results in reducing the number of accesses to the global memory. Second, the larger number of threads contributing in the reduce phase through the joining kernel and the combiner function which results in improving the reduce time.

For WC, StreamMR achieves 3.41-fold and 8.66-fold speedup on the average compared to Mars for AMD and NVIDIA GPUs respectively as shown in Figure 4.6 and Figure 4.8. This comes mainly from avoiding the time-consuming sorting phase of Mars, that consumes more than 50% and 90% of the total execution time for AMD and NVIDIA GPUs respectively as shown in Figure 4.7 and Figure 4.9. In addition, StreamMR reduces the size of intermediate output due to the use of the combiner function, thus significantly reducing the number of accesses to the global buffer especially because WC generates large number of intermediate output.

4.7.4 Comparison to MapCG

As we discussed earlier, state-of-the-art MapReduce frameworks in CUDA use atomic operations to coordinate the output from different threads. To fairly evaluate atomic-based MapReduce designs on AMD as well as NVIDIA GPU, we implemented MapCG using OpenCL. Specifically, in the map phase, all threads collaborate using global atomic operations i.e., atomic-add and atomic-compare-and-swap to write the intermediate output into

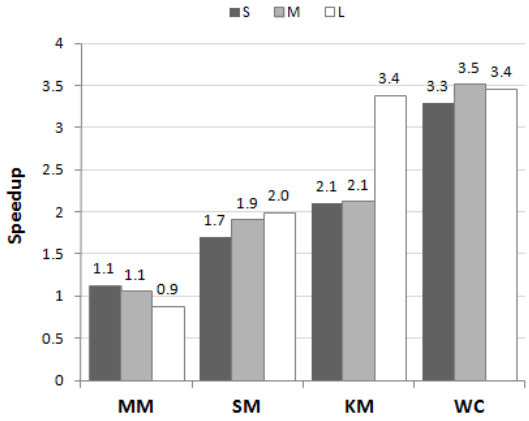

Figure 4.6: Speedup of StreamMR over Mars using small, medium, and large datasets for AMD Radeon HD 5870

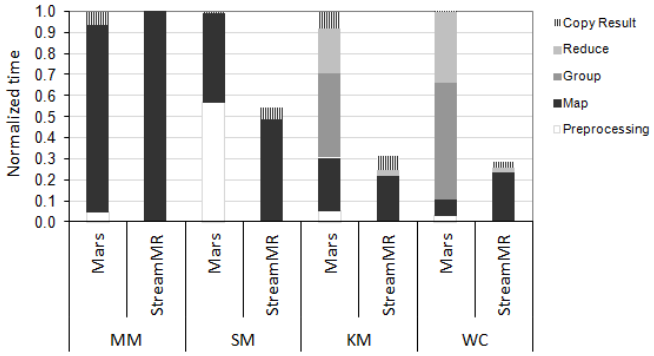

Figure 4.7: Execution time breakdown of Mars and StreamMR using large dataset for AMD Radeon HD 5870

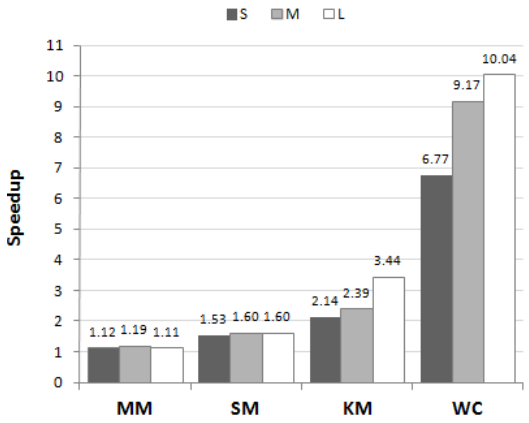

Figure 4.8: Speedup of StreamMR over Mars using small, medium, and large datasets for NVIDIA Fermi

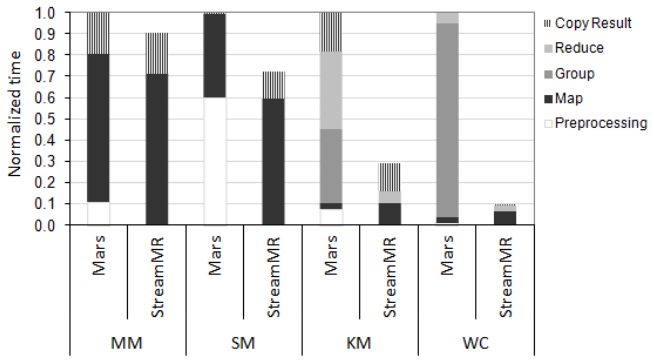

Figure 4.9: Execution time breakdown of Mars and StreamMR using large dataset for NVIDIA Fermi

global buffers and group this output using a single hash table. In our implementation, we set the size of the hash table to the aggregate sizes of the hash tables used in StreamMR. Instead of calling global atomic operation each time a thread writes to a global buffer, MapCG uses memory allocator. Only a single thread per wavefront executes global atomic-add operation to reserve certain bytes from the global buffer, then all threads in a wavefront collaborate using atomic-add operation to local memory to consume this reserved buffer. In our implementation, we reserve the maximum possible global buffer to avoid possible overflow in the allocated global buffers. Note that, In MapCG, grouping the output using the hash table is necessary even for applications without reduce phase like Matrix Multiplication. Finally, in the reduce phase, we assign each hash entry to a single thread to generate the final output.

As we discussed in Chapter 2, atomic operations on AMD GPUs can force all memory accesses to use a slow CompletePath instead of the normal FastPath, thus can result in severe performance degradation for memory-bound applications. StreamMR addresses this issue with an atomic-free design. As shown in Figure 4.10 and Figure 4.11, for MM, StreamMR significantly outperforms MapCG, i.e., with an average speedup of 28.7-fold. It turns out that the ALU:Fetch ratio (measured by AMD APP Kernel Analyzer v1.8) of MM is 0.19. Such a low ALU:Fetch ratio suggests that MM is indeed a memory-bound application. On the other hand, the ALU:Fetch ratio of SM is very high, i.e. 4.94, suggesting that SM is

more compute-bounded. Consequently, StreamMR improves the performance over MapCG by 1.8-fold speedup on the average for SM.

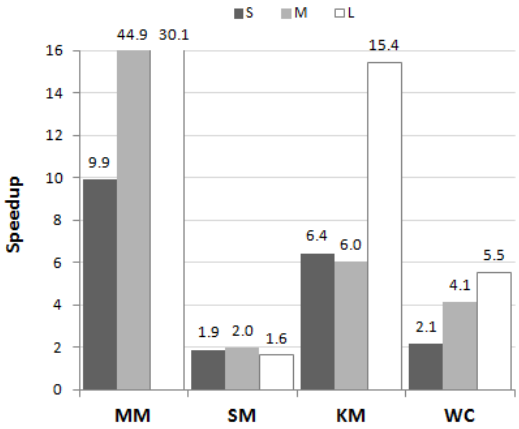

Figure 4.10: Speedup of StreamMR over MapCG using small, medium, and large datasets for AMD Radeon HD 5870

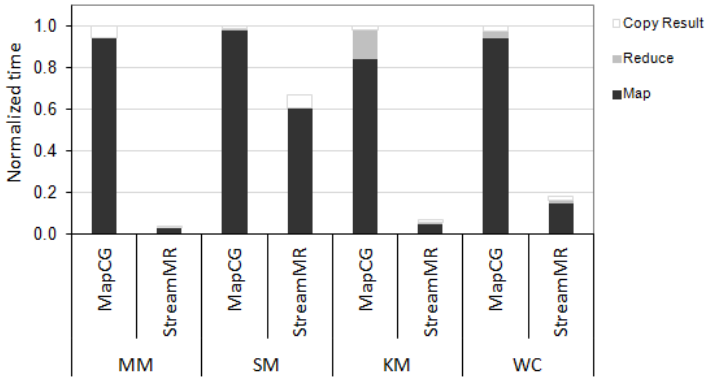

Figure 4.11: Execution time breakdown of MapCG and StreamMR using large dataset for AMD Radeon HD 5870

Although NVIDIA GPUs do not incur severe penalty from using atomic operations, the results shown in Figure 4.12 and Figure 4.13 suggests that StreamMR behaves better than MapCG by 1.1-fold speedup on the average for both MM and SM on NVIDIA GPU. This returns to the fact that StreamMR executes only map phase, however MapCG groups the intermediate output into a hash table and executes reduce phase to trace the size of the generated output in addition to the normal map phase.

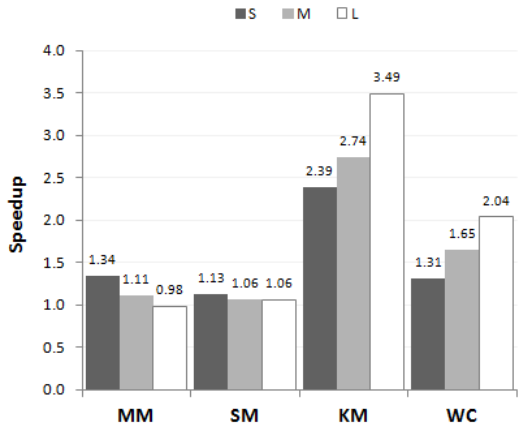

Figure 4.12: Speedup of StreamMR over MapCG using small, medium, and large datasets for NVIDIA Fermi

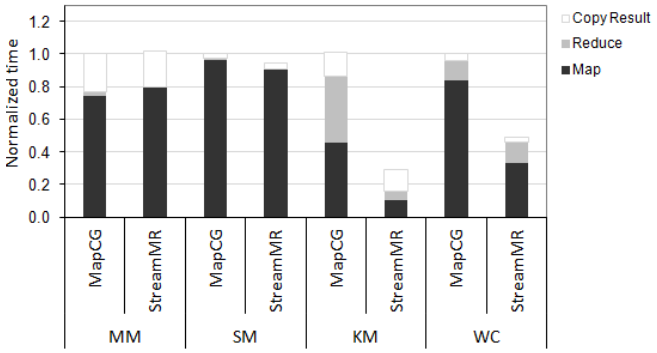

Figure 4.13: Execution time breakdown of MapCG and StreamMR using large dataset for NVIDIA Fermi

For KM, the average speedup of StreamMR over MapCG is 9.3-fold on AMD GPU. Particularly, the 95-confidence interval is [5.7,7.1], [5.7, 6.3] and [13.1, 17.7] for small, medium and large datasets respectively. Again, one of the reasons is that KM is also memory-bounded, as indicated by an ALU:Fetch ratio of 0.42 for its map kernel. In addition, the map phase of KM contributes to more than 80% of the total execution time as shown in Figure 4.11. For NVIDIA GPU, StreamMR behaves better than MapCG by 2.9-fold speedup on the average. This performance is attributed to the use of combiner within the map phase, which significantly reduces the number of accesses to the global memory, and thus improves the execution time of the map phase as shown Figure 4.13. In addition to exploiting more parallelism in the reduce phase through the use of the two-level joining kernel, thus improving the reduction time.

For WC, most of the execution time is spent in the map phase as shown in Figure 4.11 and Figure 4.13. Specifically, the map phase of WC contributes to 94% and 84% of the total execution time for AMD and NVIDIA GPU respectively. With the use of combiner in the StreamMR, the number of intermediate key/value pairs is reduced significantly, thus improving WC performance over MapCG by 3.9-fold and 1.7-fold speedup for AMD GPU and NVIDIA GPU respectively. The better performance of StreamMR on AMD GPU is attributed to the memory-boundness of the map kernel of WC with ALU:Fetch ratio of 0.67, in addition to the large number of intermediate output in WC which further exacerbates the overhead of using the CompletePath within MapCG.

4.7.5 Overflow Handling Overhead

In this experiment, we aim at quantifying the overhead of the overflow handling mechanisms i.e., global and local buffers overflow. For MM, since the size of the output is deterministic, then the overflow can be avoided. For SM and WC, there is a high probability for the global overflow to occur since the size of the output is nondeterministic and depends on the input

file (and the keyword for SM). For KM, if the local buffer is not set appropriately, a local overflow may be encountered.

We run SM using large-size dataset and varied the global buffer size to study the effect of global overflow on the performance. We reduce the size of the preallocated output buffer, so overflow occurs, and another map kernel is executed. The overflow percentage is the ratio between the number of matches emitted by the second map kernel and the total number of matches. As shown in Figure 4.14, for AMD GPU, the speedup of StreamMR over Mars decreases from 1.99 to 1.53 when the percentage of overflow reaches 53%. As the overflow percentage increases to 93%, the speedup drops further to 1.18. The same behavior is noticed for NVIDIA GPUs as shown in Figure 4.15. This is because StreamMR will incur more and more counting overhead as the overflow percentage increases. However, the above performance results also suggest the overhead of global overflow is tolerable.

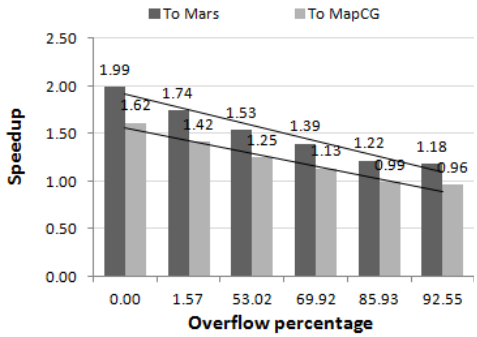

Figure 4.14: Effect of global overflow on the speedup over Mars and MapCG using string-match for AMD Radeon HD 5870

For WC, using large-size dataset, we varied the allocated global buffer to force different percentages of overflow. Upon an overflow, another map kernel is launched to handle the overflowed records. Executing another map kernel, generates more hash tables that are handled by the reduce phase, thus increasing the execution time of the reduce phase. As shown in Figure 4.16 and Figure 4.17, the speedup of StreamMR over Mars decreases from 3.45 to 3.04 and from 10.04 to 8.71 for AMD and NVIDIA GPUs respectively when the

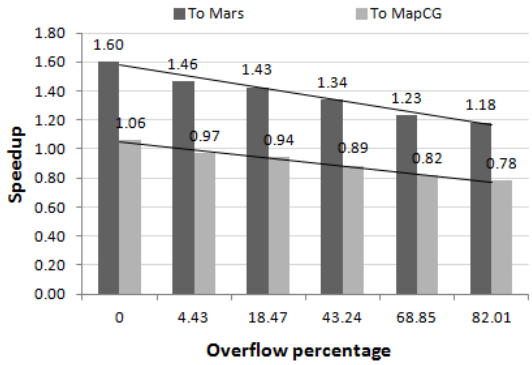

Figure 4.15: Effect of global overflow on the speedup over Mars and MapCG using string-match for NVIDIA Fermi

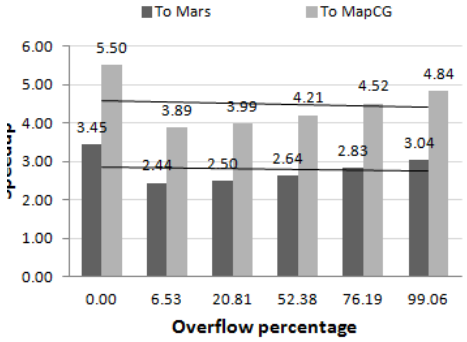

Figure 4.16: Effect of global overflow on the speedup over Mars and MapCG using wordcount for AMD Radeon HD 5870

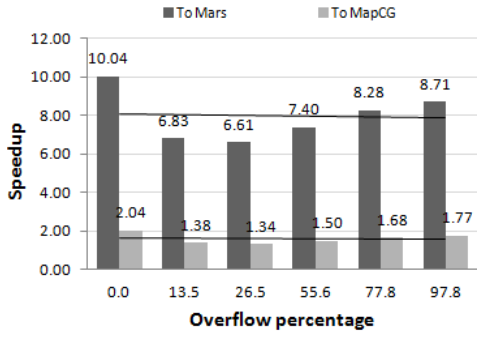

Figure 4.17: Effect of global overflow on the speedup over Mars and MapCG using wordcount for NVIDIA Fermi

percentage of overflow reaches almost 100%.

For KM, we varied the allocated local buffer, so instead of running all threads per wavefront concurrently, they run on two and four consecutive iterations. As a result, the map kernel execution time increases as shown in Figure 4.18. Specifically, the speedup compared to overflow-free case is 0.91 and 0.76 for two and four consecutive iterations respectively.

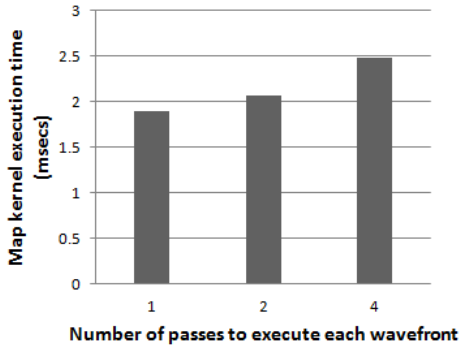

Figure 4.18: Effect of local overflow on the Map kernel execution time of KMeans

4.7.6 Impact of Using Image Memory

In this experiment, we evaluate the effect of using texture memory instead of global memory to store the input dataset. Since the data retrieved from the texture memory are cached, we expect applications with data locality to benefit from this feature. MM is an example of such applications since a single row/column is accessed by several map threads. For SM, KM and WC, since each thread works in a different piece of input data, texture caching may not be beneficial.

For MM, we have found that using texture memory to store the input matrices, improves the performance of the map kernel significantly. More specifically, the speedup of the map kernel over non-texture map kernel is 4.89 and 3.56 for 256 X 256 and 512 X 512 matrices respectively on AMD GPU. Although the use of the image memory is not listed within the major optimizations for new versions of NVIDIA GPUs [54], we have found that it can

improve the performance of the map kernel of matrix multiplication by 2.9-fold speedup for 256 X 256 matrices on NVIDIA Fermi GPUs.

4.7.7 Quantifying the Impact of the Scalability Optimization

In this experiment, we quantify the benefits of the scalable reduce design presented in Section 4.3.2.1 compared to the initial design. As discussed before, this optimization targets applications producing large number of intermediate records and hash tables like wordcount. So in this experiment, we run wordcount using three version of the reduce phase. the first one is the initial design that requires passing through the hash tables multiple times. The second one is the optimized reduce phase involving one joining kernel and one reduction kernel. The third one is also optimized reduce phase but with two-level joining. Since combiner function exists for wordcount, we directly apply combining during joining, thus avoiding the need for a separate reduction kernel.

As shown in Figure 4.19 and 4.20, using the basic-optimized reduce phase slightly reduces the reduction time. However, it allows exposing more parallelism through the use of the two-level joining, thus significantly improving the reduction time by 10.08 and 12.89-fold speedup for AMD GPU and NVIDIA GPU, respectively.

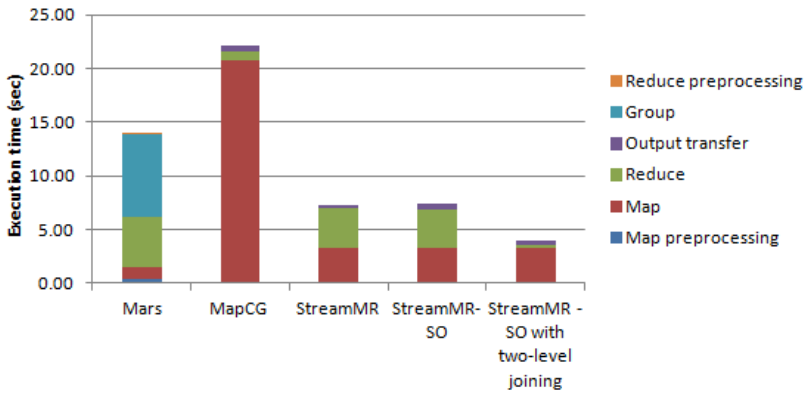

Figure 4.19: Effect of scalability optimization (SO) of the reduce phase using wordcount on AMD GPU

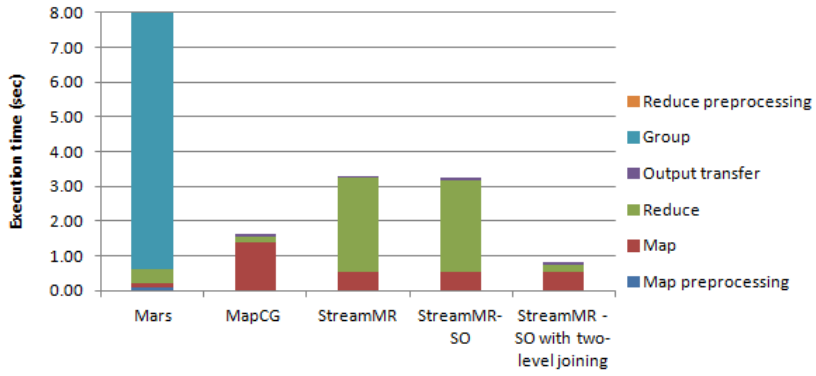

Figure 4.20: Effect of scalability optimization (SO) of the reduce phase using wordcount on NVIDIA GPU

4.8 Chapter Summary

In this chapter, we designed StreamMR, an OpenCL atomic-free implementation of MapReduce optimized for heterogeneous devices. Through atomic-free mechanisms for output writing and shuffling, StreamMR significantly outperforms MapCG on AMD GPUs. Specifically, StreamMR behaves better than MapCG by 10.9-fold speedup on the average (up to 44.5-fold speedup). By avoiding the time consuming preprocessing phases and sorting when grouping intermediate results, StreamMR outperforms Mars on AMD GPUs by 2.21-fold on the average (up to 3.5-fold speedup).

Through highly scalable and optimized reduce phase, StreamMR can outperforms MapCG and Mars on NVIDIA GPUs as well. Particularly, StreamMR behaves better than MapCG and Mars by 1.7-fold speedup (up to 3.5-fold speedup) and 3.85-fold speedup (up to 10.04-fold speedup), respectively.

This page intentionally left blank.

Chapter 5

Optimized MapReduce Workflow

5.1 Overview

Having an efficient and portable MapReduce implementation, the next step towards achieving our overarching goal is to explore how to efficiently co-schedule the map and reduce tasks among different resources within node and across nodes. The traditional approach is to enforce a barrier synchronization between the map phase and the reduce phase, i.e., the reduce phase can only start when all map tasks are completed. For heterogeneous resources, it is highly expected that the faster compute resources will finish their assigned map tasks earlier, but these resources cannot proceed to the reduce processing until all the map tasks are finished, thus resulting in waste of resources.

In this chapter, we propose and compare two asynchronous data-processing techniques to enhance resource utilization and performance of MapReduce for a specific class of MapReduce jobs, called *recursively reducible* MapReduce jobs. For this type of MapReduce jobs, a portion of the map results can be reduced independently, and the partial reduced results can be recursively aggregated to produce global reduce results. More details about recursively reducible MapReduce jobs will be discussed in Section 5.2. Our first approach, hierarchical reduction (HR), overlaps map and reduce processing at the inter-task level. This approach

starts a reduce task as soon as a certain number of map tasks complete and aggregates partial reduced results using a tree hierarchy. The second approach, incremental reduction (IR), exploits the potential of overlapping data processing and communication within each reduce task. It starts a designated number of reduce tasks from the beginning and incrementally applies reduce function to the intermediate results accumulated from map tasks.

Implementing the incremental reduction approach requires running the map and reduce tasks concurrently on each resource. Additionally, the intermediate output need to be regularly pipelined to the resource during the execution of the reduce task. With the current specification and implementation of OpenCL [1], concurrently running more than one kernel on GPUs is not supported. Additionally, it is not possible to transfer data from the host memory to the device memory while executing a kernel. So as a proof-of-concept and to be able to study the scalability of the proposed approaches with the number of resources, we have evaluated our approaches against the traditional one using Hadoop [1], an open-source MapReduce implementation.

The rest of this chapter is organized as follows: Section 5.2 discusses background information about Hadoop and recursively reducible MapReduce jobs. Section 5.3 and 5.4 describe the design of the proposed approaches including the hierarchical and incremental reduction. Section 5.5 evaluates the performance of the proposed approaches using an analytical model. The experimental results are discussed in Section 5.6. We conclude in Section 5.7.

5.2 Background

5.2.1 Hadoop

Hadoop is an open-source Java implementation of the MapReduce framework. It can be logically segregated into two subsystems, i.e., a distributed file system called HDFS and a

[1]OpenCL 1.1, implemented through AMD APP SDK v2.5

MapReduce runtime. The MapReduce runtime follows a master-slave design. The master node is responsible for managing submitted jobs, assigning the map and reduce tasks of every job to the available workers. By default each worker can run two map tasks and two reduce tasks simultaneously.

At the beginning of a job execution, the input data is split and assigned to individual map tasks. When a worker finishes executing a map task, it stores the map results as intermediate key/value pairs locally. The intermediate results of each map task will be partitioned and assigned to the reduce tasks according to their keys. A reduce task begins by retrieving its corresponding intermediate results from all map outputs (called the *shuffle* phase). The reduce task then sorts the collected intermediate results and applies the reduce function to the sorted results. To improve performance, Hadoop overlaps the copy and sort of finished map outputs with the execution of newly scheduled map tasks.

5.2.2 Recursively Reducible Jobs

Word counting is a simple example of recursively reducible jobs. The occurrences of a word can be counted first on different splits of an input file, and those partial counts can then be aggregated to produce the number of word occurrences in the entire file. Other recursively reducible MapReduce applications include association rule mining, outlier detection, commutative and associative statistical functions etc. In contrast, the square root of sum of values is an example of reduce function that is not recursively reducible, because $(a+b)^2 + (c+d)^2$ does not equal $(a+b+c+d)^2$. However, there are some mathematical approaches that can transform such functions to benefit from our solution.

It is worth mentioning that there is a *combiner function* provided in typical MapReduce implementations including Hadoop. The combiner function is used to reduce key/value pairs generated by a *single* map task. The partially reduced results, instead of the raw map output,

are delivered to the reduce tasks for further reducing. Our proposed asynchronous data processing techniques are applicable to all applications that can benefit from the combiner function. The fundamental difference between our techniques and the combiner function is that our techniques optimize the reducing of key/value pairs from *multiple* map tasks.

5.3 Hierarchical Reduction (HR)

5.3.1 Design and Implementation

Hierarchical reduction seeks to overlap the map and reduce processing by dynamically issuing reduce tasks to aggregate partially reduced results along a tree-like hierarchy. As shown in Figure 5.1, as soon as a certain number (i.e., defined by the aggregation level σ_H) of map tasks are successfully completed, a new reduce task will be created and assigned to one of the available workers. This reduce task is responsible for reducing the output of the σ_H map tasks that are just finished. When all map tasks are successfully completed and assigned to reduce tasks, another stage of the reduce phase is started. In this stage, as soon as a certain σ_H reduce tasks are successfully completed, a new reduce task will be created to reduce the output of the σ_H reduce tasks. This process repeats until there is only one remaining reduce task, i.e., when all intermediate results are reduced.

Although conceptually the reduce tasks are organized as a balanced tree, in our implementation a reduce task at a given level does not have to wait for all of the tasks at the previous level to finish. In other words, as soon as a sufficient number of tasks (i.e., σ_H) from the previous level becomes available, a reduce task from the subsequent level can begin. Such a design can reduce the associated scheduling overhead of HR.

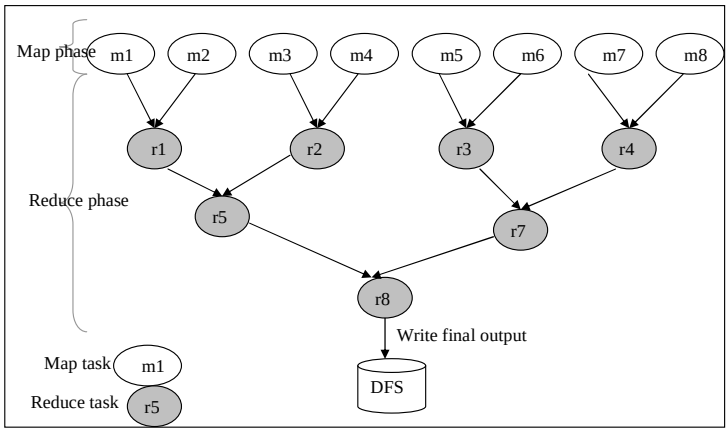

Figure 5.1: Hierarchical reduction with aggregation level equals 2

5.3.2 Discussion

One advantage of HR is that it can parallelize the reduction of a single reducing key across multiple workers, whereas in the original MapReduce framework (MR), the reduction of a key is always handled by one worker. Therefore, this approach is suitable for applications with significant reduce computation per key. However, HR incurs extra communication overhead in transferring the intermediate key/value pairs to reduce tasks at different levels of the tree hierarchy, which can adversely impact the performance as the depth of the tree hierarchy increases. Other overheads include the scheduling cost of reduce tasks generated on the fly.

For the fault tolerance scheme, it should be modified to recover the failure of reduce tasks. In particular, the *JobTracker* should keep track of all created reduce tasks, in addition to the tasks assigned to be reduced by these reduce tasks. Whenever, a reduce task fails, another copy of this task should be created and the appropriate tasks should be assigned again for reduction. Each reduce task materializes its output locally, so it will post this output again to the new task.

5.4 Incremental Reduction (IR)

5.4.1 Design and Implementation

Incremental reduction aims at starting the reduce phase as early as possible within a reduce task. Specifically, the number of reduce tasks are defined at the beginning of the job similar to the original MapReduce framework. Within a reduce task, as soon as a certain amount of map outputs are received, the reduction of these outputs starts and the results are stored locally. The same process repeats until all map outputs are retrieved.

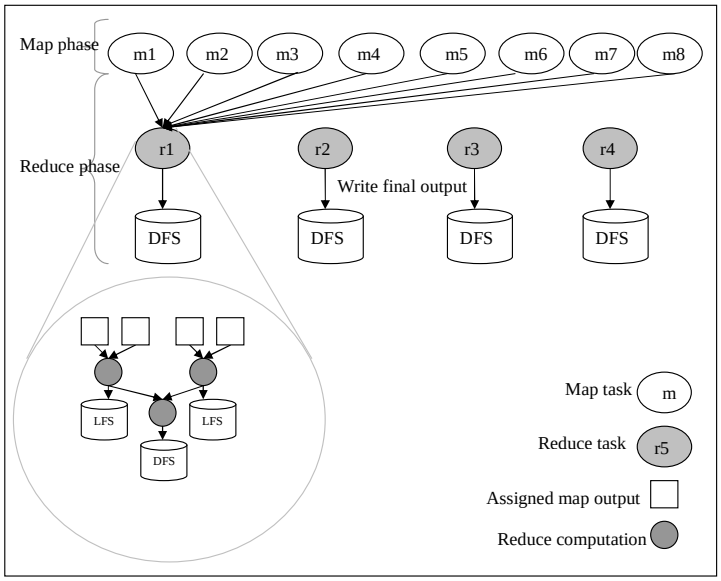

Figure 5.2: Incremental reduction with reduce granularity equals 2

In Hadoop, a reduce task consists of three stages. The first stage, named shuffling, copies the task's own portion of intermediate results from the output of all map tasks. The second stage, named sorting, sorts and merges the retrieved intermediate results according to their keys. Finally, the third stage applies the reduce function to the values associated with each key. To enhance the performance of the reduce phase, the shuffling stage is overlapped with the sorting stage. More specifically, when the number of in-memory map outputs reaches a certain threshold, *mapred.inmem.merge.threshold*, these outputs are merged and the results

are stored on-disk. When the number of on-disk files reaches another threshold, *io.sort.factor*, another on-disk merge is performed. After all map outputs are retrieved, all on-disk and in-memory files are merged, and then the reduction stage begins.

In our IR implementation, we make use of *io.sort.factor* and *mapred.inmem.merge.threshold*. When the number of in-memory outputs reaches to the *mapred.inmem.merge.threshold* threshold, they are merged and the merging results are stored on the disk. When the number of on-disk outputs reaches to the *io.sort.factor* threshold, the incremental reduction of these outputs begins and the reducing results are stored instead of the merging results. When all map outputs are retrieved, the in-memory map outputs are reduced along with the stored reducing results. The final output data is written to the distributed file system. The entire process is depicted in Figure 5.2.

5.4.2 Discussion

IR incurs less overheads than HR for two reasons. First, the intermediate key/value pairs are transmitted once from the map to reduce tasks instead of several times along the hierarchy. Second, all of the reduce tasks are created at the start of the job and hence the scheduling overhead is reduced.

In addition to the communication cost, the number of writes to local and distributed file system are the same (assuming the same number of reduce tasks) for both MR and IR. Therefore, IR can outperforms MR when there is sufficient overlap between the map and reduce processing.

The main challenge of IR is to choose the right threshold that triggers an incremental reduce operation. Too low a threshold will result in unnecessarily frequent I/O operations, while too high a threshold will not be able to deliver noticeable performance improvements. Interestingly, a similar decision, i.e., the merging threshold, has to be made in the original

Hadoop implementation as well. Currently we provide a runtime option for users to control the incremental reduction threshold. In the future, we plan to investigate self-tuning of this threshold for long running MapReduce jobs.

It is worth noting that since the map and reduce tasks in this approach are created by the same manner as in Hadoop, then the fault tolerance scheme of Hadoop works well.

5.5 Analytical Models

In this section, we derive analytical models to compare the performance of the original MapReduce (MR) implementation of Hadoop, and the augmented implementations with hierarchical reduction (HR) and incremental reduction (IR) enhancements. Table 5.1 presents all the parameters used in the models. We group our discussion according to the relationship between the number of map tasks m and the number of available execution slots in the cluster $2n$ (recall that there are two execution slots per node by default).

Without loss of generality, our modeling assumes the number of reduce tasks r is smaller than the number of execution slots $2n$. In fact, the Hadoop documentation recommends that 95% of the execution slots is a good number for the number of reduce tasks for typical applications. However, our analysis can be easily generalized to model the cases where there are more reduce tasks than the number of execution slots.

5.5.1 Case 1: Map Tasks $\leq$ 2 $\times$ Nodes Number

When the number of map tasks is less than the number of execution slots, all map tasks are executed in parallel and completed simultaneously. Assuming the execution time of each map task is the same, the map phase will finish after t_m. And since m $\leq$ 2n means working with small dataset, we can assume that the communication cost is small, so MR and IR cannot overlap the copying with other reduce computations. Particularly, The execution time of the

Parameters	Meaning
m	Number of map tasks
n	Number of nodes
k	Total number of intermediate key/ value pairs
r	Number of reduce tasks of the MR framework
t_m	Average map task execution time
t_{rk}	Average execution time of reducing values of a single key
σ_H	Aggregation level used in HR
C	Communication cost per key/value pair
C_{MR}	Communication cost from m map tasks to r reduce tasks in MR
C_{HR}	Communication cost from the assigned σ_H map tasks to a reduce task in HR

Table 5.1: Parameters used in the performance model

reduce phase is the same for MR and IR, however HR has different reduce computations.

MR / IR For the original Hadoop implementation, and incremental reduction, the reduce phase finishes after $C_{MR} + \lceil \frac{k}{r} \rceil log \lceil \frac{k}{r} \rceil + t_{rk} \times \lceil \frac{k}{r} \rceil$. Where the first term is the communication cost, the second term is the merging time, and the third term is the reducing time. Hence the total execution time:

$$T_{MR} = t_m + C_{MR} + \lceil \frac{k}{r} \rceil log \lceil \frac{k}{r} \rceil + t_{rk} \times \lceil \frac{k}{r} \rceil \tag{5.1}$$

HR For hierarchical reduction, after all map tasks are finished, the reduce phase begins by initiating $\frac{m}{\sigma_H}$ reduce tasks, where σ_H is the used aggregation level. When these reduce tasks are finished, the outputs from every σ_H reduce tasks are assigned to another reduce task and so on until all outputs are reduced. So, we need $log_{\sigma_H}(m)$ stages to finish the reduce phase, where every stage executes in $C_{HR} + \frac{\sigma_H k}{m} log(\frac{\sigma_H k}{m}) + t_r \times \frac{\sigma_H k}{m}$, where the first term is the communication cost, and the second term is the merging time . Moreover, for simplicity, we assume a linear reduce function, i.e., if it takes t to reduce the values of a single key from m map tasks, then it takes $x \times \frac{t}{m}$ to reduce the values from x map tasks. So the total execution

time:

$$T_{HR} = t_m + (C_{HR} + \frac{\sigma_H k}{m} log(\frac{\sigma_H k}{m}) + t_r \times \frac{\sigma_H k}{m}) \times log_{\sigma_H}(m) \tag{5.2}$$

By comparing equations 5.1 and 5.2, we can conclude that when $m \leq 2n$, there is always a configuration for MR to behave better than HR. Specifically, by neglecting the communication cost, and setting r to 2n, T_{HR} becomes longer than T_{MR}, This is expected because all map tasks are finished at the same time and there is no way to overlap map and reduce phases.

5.5.2 Case 2: Map Tasks > 2 × Nodes Number

When there are more map tasks than the execution slots, the map tasks are executed in several stages. In addition, the execution of the reduce tasks can be overlapped with the map computations based on the CPU utilization of the map tasks which we call the *overlapping degree*. We consider two different cases based on the overlapping degree. The first case corresponds to high overlapping degree i.e., copy and merge (merge and reduce in IR) of almost all retrieved intermediate key/value pairs can be overlapped with the map computations. The second case corresponds to low overlapping degree i.e., merging (merging and reducing in IR) of only a portion of the intermediate results can be overlapped. It is worth noting that as the number of map tasks increases, the corresponding number of idle slots between the execution of these map tasks (i.e., scheduling overhead) increases, which in turn increases the overlapping degree.

Throughout this section the total execution time is based on the following equation:

$$T = Maptime + Mergetime + Reducetime \tag{5.3}$$

MR For the original Hadoop implementation, when the overlapping degree is high, the merging phase of MR in Figure 5.3 can be eliminated. So the total time becomes $Maptime + Reducetime$, assuming the final stage merging is neglected.

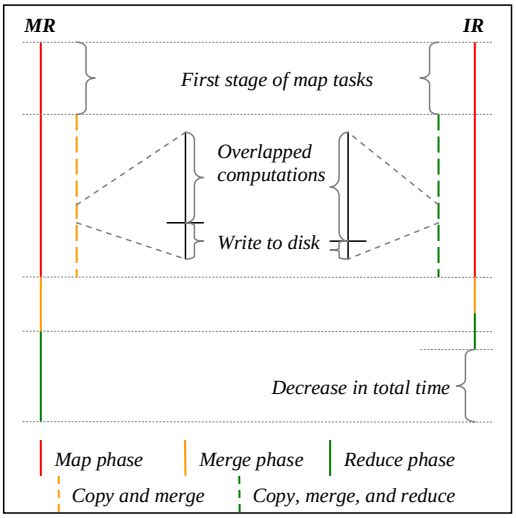

Figure 5.3: Execution of MR and IR

For low overlapping degree, if the reduce tasks occupy all nodes i.e., the number of reduce tasks (r) is larger than or equals n, then merging of only a portion of the intermediate results can be overlapped with the map computations, and the total time can be expressed by equation 5.4. Where o represents the reduction in the merging time.

$$T_{MR} = Maptime + (Mergetime - o) + Reducetime \tag{5.4}$$

However, when the reduce tasks do not occupy all nodes, more merging can be overlapped with the map computations due to the load balancing effect i.e., the nodes executing reduce tasks execute smaller number of map tasks compared to the other nodes. As a result, the Map time is increased and the merging time is decreased as shown in equation 5.5, where (l) represents the load balancing effect, o' represents the overlapping effect, and $o' > o$. As r increases, l and o' keeps decreasing, until reaching 0 and o respectively when r = n (equation 5.4).

$$T_{MR} = (Maptime + l) + (Mergetime - o') + Reducetime \tag{5.5}$$

HR For hierarchical reduction, map and reduce processing at different stages can be overlapped as shown in Figure 5.4. To compare HR's performance with MR, we consider more

detailed modeling like the previous section.

For HR, When all map tasks are finished, the remaining computations is to reduce the un-reduced map tasks. In addition to combining the results of this reducing stage with other partial reduced results. Specifically, the total execution time of MR, and HR can be represented by the following equations, where s is the remaining number of stages of HR's hierachy:

$$T_{MR} = Maptime + C_{MR} + \lceil \frac{k}{r} \rceil log(\lceil \frac{k}{r} \rceil) + t_{rk} \times \lceil \frac{k}{r} \rceil \tag{5.6}$$

$$T_{HR} = Maptime + (C_{HR} + \frac{\sigma_H k}{m} log(\frac{\sigma_H k}{m}) + t_{rk} \times \frac{\sigma_H k}{m}) \times s \tag{5.7}$$

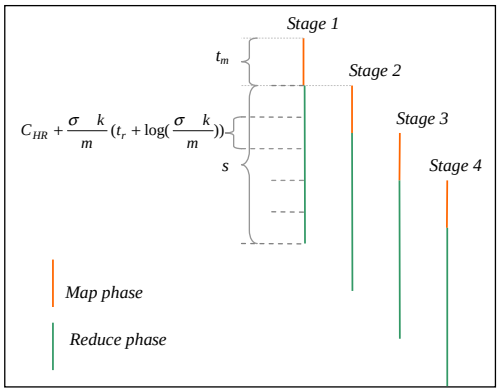

Figure 5.4: Execution of HR framework when $m = 8n$

Assuming every map task produces a value for each given key, then C_{MR} is $\frac{k}{r} \times C$, then C_{HR} is $\frac{\sigma_H k}{m} \times C$, and C_{HR} equals $\frac{\sigma_H r}{m} \times C_{MR}$, where C is Communication cost per key/value pair. By substituting C_{HR} in equation 5.7 by the previous value the equation becomes:

$$T_{HR} = Maptime + \frac{\sigma_H}{m} \times (rC_{MR} + klog(\frac{\sigma_H k}{m}) + kt_{rk}) \times s \tag{5.8}$$

When the overlapping degree is high, s can be replaced by $(log_{\sigma_H}(2n) + 1)$ in equation 5.8, which represents the reduction of the map tasks of the final stage. Moreover, the merging

time can be eliminated from equation 5.6. So for significantly large m, the communication part, and the reducing part of the equation is smaller than that of equation 5.6. If these terms occupy significant portion of the total time of MR, then HR will behave better than MR as we will see in the experimental evaluation. However, when the overlapping degree is low, s can be very deep, and the performance of HR can be worse than MR.

IR For incremental reduction, when the overlapping degree is high, the merging, and reducing phase of IR in Figure 5.3 can be eliminated. So the total time becomes $Maptime$, assuming the final stage merging and reducing is neglected. Definitely, IR can behave better than MR, and HR in this case, especially when the reducing time of MR is significant.

For low overlapping degree, if r is larger than n, then merging and reducing of only a portion of the intermediate results can be overlapped with the map computations, and the total time can be expressed by equation 5.9. Where o_m and o_r represents the reduction in the merging and reducing time respectively.

$$T_{IR} = Maptime + (Mergetime - o_m) + (Reducetime - o_r) \tag{5.9}$$

To compare this with equation 5.5, we consider the details of the overlapping computations in MR and IR. The main difference is that IR performs reducing after merging and writes the results of reduce rather than the results of merge to disk. Assuming the reduce function changes the size of the input by a factor of x and the reduce function is linear, then the overlapped computations of MR and IR can be represented by the following equations, where I is the size of intermediate key/value pairs to be merged and reduced during the map phase, (IlogI) is the average number of compare operations executed during the merge, P_s is the processor speed, d_s is the disk write speed, and p_s is very smaller than d_s:

$$O_{MR} = \frac{I_{MR} log I_{MR}}{P_s} + \frac{I_{MR}}{d_s} \tag{5.10}$$

$$O_{IR} = \frac{I_{IR} log I_{IR} + I_{IR}}{P_s} + \frac{I_{IR} \times x}{d_s} \tag{5.11}$$

Given the same overlapping degree, if $x < 1$, which is valid for several applications like wordcount, grep, linear regression,... etc, then IR is able to conduct more merging in addition to reducing overlapped with map computations. So, the merging and reducing terms in equation 5.9 is less than the same terms in equation 5.4. So, IR can behave better than MR given the reduce computations is significant as illustrated by Figure 5.3. Note that, the cores's speed of the recent multicore machines advances in a higher rate compared to the disk speed. In addition, this speed is further reduced by the contention among the cores on the disk I/O, so IR may behave better in these emerging architectures. On the other side, if $x \geq 1$, then the performance of IR highly depends on the complexity of the reduce function compared to the merging.

By applying the previous analysis to the case where $r < n$, we can conclude that IR can behave better than MR in this case also given the reducing time occupies a significant portion of the total execution time.

5.6 Evaluation

In this section, we present performance evaluations of our proposed techniques. Our experiments are executed on System X at Virginia Tech, comprised of Apple Xserve G5 compute nodes with dual 2.3GHz PowerPC 970FX processors, 4GB of RAM, 80 GByte hard drives. The compute nodes are connected with a Gigabit Ethernet interconnection. Each node is running the GNU/Linux operating system with kernel version 2.6.21.1. The proposed approaches are developed based on Hadoop 0.17.0.

5.6.1 Overview

We have conducted four sets of experiments in order to study the performance of our approaches from different perspective. All of the experiments are conducted using wordcount

and grep applications. Wordcount is an application that parses a document or a number of documents, and produces for every word the number of its occurrence. Grep accepts a document or a number of documents and an expression; it matches this expression along the whole documents and produces for every match the number of its occurrence. In the first experiment, we aim at studying the scalability of the different reducing approaches with the dataset size. In the second and third experiment, we deeply analyzed the performance of wordcount and grep. Finally, another experiment has been conducted to study the robustness of the three approaches to the heterogeneity of the target environment.

The major performance metric in all experiments is the total execution time in seconds. Moreover, For a fair comparison, we follow the guidance given in the Hadoop documentation regarding the number of map and reduce slots per node, the io-sortfactor, and the merge-threshold. In addition, the aggregation level of the hierarchical reduction approach is set to 4 which also produces the best results for 32 nodes cluster. Furthermore, the granularity of intermediate-result merges in MR and the incremental reductions in IR is the same in all experiments. Finally, we flushed the cache before running any job to ensure the accuracy of our results.

5.6.2 Scalability with the Dataset Size

In this experiment, we aim at studying the scalability of the three reducing approaches with the size of the input dataset. We run wordcount and grep using 16GB and 64GB. For wordcount, the number of reduce tasks was set to 4, and 8, a broader range were used in experiment 5.6.3. For grep, we used an average query that produces results of moderate size, the performance of different queries was investigated in experiment 5.6.4.

As shown in Figure 5.5, generally, as the size of the input dataset increases, the performance improvement of IR over MR increases. Specifically, for wordcount, as the size of the input

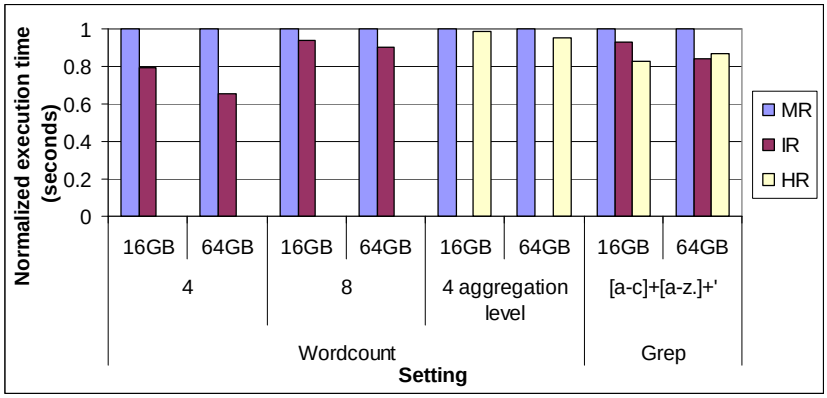

Figure 5.5: Scalability with dataset size using wordcount and grep

increases to 64GB, IR gets better than MR by 34.5% and 9.48% instead of 21% and 5.97% in case of 16GB for 4 and 8 reduce tasks respectively. In addition, for grep, increasing the dataset size, improves IR's performance over MR i.e., IR is better than MR by 16.2 % instead of 7.1% for 16GB. The scalability of our approaches attributes to two reasons; first, as the dataset size increases, the map phase has to perform more IO. Second, for 64 GB, the number of map tasks increases from 256 to 1024, thus increasing the scheduling overheads. Both reasons results in providing more room for overlapping map with reduce computations.

Although the performance improvement of HR over MR with 16 reduce tasks increases by 9.96% for wordcount as the input dataset size increases, it decreases by 4.13% for grep. This is because of the extra communication cost involved with the increase of the dataset size. Which can be compensated by the long map phase time of wordcount i.e., 2801 seconds (average of the three cases shown in Figure 5.5) compared to only 1177 seconds for grep.

Since the normal Mapreduce jobs, processes huge amount of input data, all of the subsequent experiments are performed using input dataset of size 64 GB.

5.6.3 Wordcount Performance

In a cluster of 32 nodes, we run wordcount on a dataset of size 64 GB, the number of map tasks was set to 1024, and the number of reduce task was varied from 1 to 64. As shown in Figure 5.6, as the number of reduce tasks increases, IR's improvement decreases. Specifically, for one reduce task, IR behaves better than MR by 35.33%. When the number of reduce tasks is increased to 4, IR behaves better by 34.49%. As the number of reduce tasks increases, the processing time of a reduce task decreases, thus providing little room for overlapping the map and reduce processing. Specifically, when the number of reduce tasks is 32, a reduce task only consume a mere 6.83% in the total execution time as shown in table 5.2. In addition, during the map phase, most of the resources of any node are utilized by the map tasks i.e., 32 map tasks compared to only 21 in case of 1 reduce task As shown in table 5.3, so IR cannot perform any incremental merges.

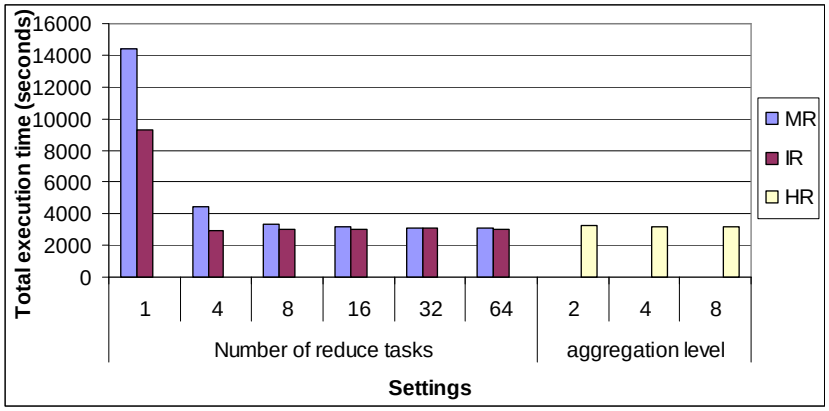

Figure 5.6: Performance of MR vs. IR using wordcount

Furthermore, IR achieves its best performance at 4 reduce tasks because this provides the best compromise between level of parallelism controlled by the number of reduce tasks and overlapping map with reduce. Specifically, in this case, IR conducts 55 incremental merges overlapped with the map computations compared to 0 in case of 32 reduce tasks as shown in table 5.2. As a result, the nodes executing a reduce task executes 26 map tasks instead of

Reduce Tasks	Incremental merges	Reduce Time	Map Time
1	46	11736	2690
4	55	1699.5	2732
8	13	524	2787
16	2	315	2845
32	0	210	2865

Table 5.2: MR, and IR performance measures

Reduce Tasks	MR	IR
1	21	21
4	22	25 + 1S[1]
8	26 + 2S	28
16	30 + 2S	28 + 2S
32	31	32

Table 5.3: Number of map tasks executed with every reduce task

32 map tasks in case of 32 reduce tasks; recall the *Load balancing effect* discussion in section 5.5.

The best performance is achieved at 32 and 4 reduce tasks for MR, and IR respectively. With the best performance for both MR and IR, IR is better by 5.86%.

On the other side, using an aggregation level of 4, HR behaves better than MR with 8 reduce tasks by 5.13%. We changed the aggregation level from 2 to 8 as shown in Figure 5.6, however the best performance is achieved at 4, because it provides the best compromise between the depth of the hierarchy and the waiting time. For example, with aggregation level of 2, the waiting time until triggering a reduce task is minimum, however, the overhead of the hierarchy in terms of its depth and the communication cost is high.

To better understand the benefits of the incremental reduction approach, we measured the CPU utilization throughout the job execution, and the number of disk transfers per second during the map phase for both MR and IR. As shown in Figure 5.7 and 5.8, the CPU utilization of IR is greater than MR by 5% on the average. In addition, the average disk

[1]Speculative Map Task

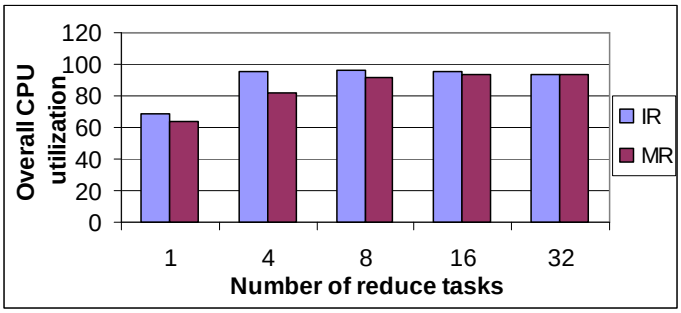

Figure 5.7: CPU utilization throughout the whole job using wordcount

transfers of IR is less than MR by 2.95 transfers per second. This returns to the smaller amount of data written to disk by IR, since it reduces the intermediate data before writing it back to disk. This in turn reduces the size of data read from disk at the final merging and reducing stage.

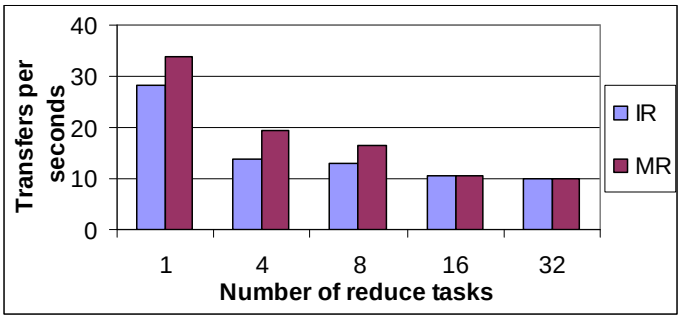

Figure 5.8: Number of disk transfers per second through the map phase using wordcount

To conclude, for any number of reduce tasks, IR achieves either better or same performance as MR. And the best performance for IR is achieved using only 4 reduce tasks, This means that IR is more efficient in utilizing the available resources. So, we expect IR to achieve better performance when several jobs are running at the same time, or with larger amounts of reduce processing. Particularly, when running three concurrent jobs of wordcount, the best configuration of IR behaves better than the best configuration of MR by 8.01% instead of 5.86% as shown in table 5.4.

Concurrent jobs	MR Execution Time (seconds)	IR Execution Time (seconds)
1	3107.5	2925.5
2	6064	5687
3	9025.5	8303

Table 5.4: MR, and IR performance with concurrent jobs

5.6.4 Grep Performance

In a cluster of 32 nodes, we run grep on a dataset of size 64 GB, the number of map tasks was set to 1024, and the number of reduce tasks was set to the default i.e., one. Grep runs two consecutive jobs; one returns for each match the number of its occurrence, and the other is a short job that inverts the output of the previous job so that the final output will be sorted based on occurrence of the matches instead of alphabetically. In this experiment, we focus on the first longer job. We used five different queries each produces different number of matches and hence intermediate and final key/value pairs.

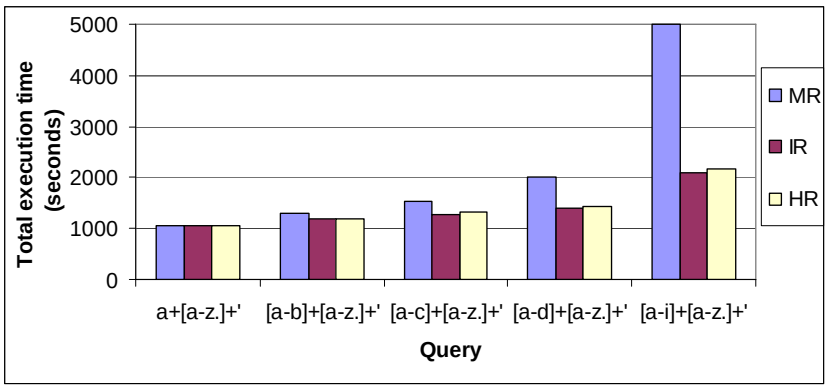

Figure 5.9: Performance of MR, IR, and HR using grep

As shown in Figure 5.9, IR's and HR's performance are almost the same. In addition, for the first query, all reducing approaches have the same performance. For subsequent queries, the performance of HR and IR gets better. Specifically, for the second query, IR behaves better than MR by 12.2%. IR's performance keeps increasing until reaching 30.2 % and

Query	Reduce Time (seconds)	Intermediate Data Size (records)
a+[a-z.]+	135	37,285,680
[a-b]+[a-z.]+	250	55,897,216
[a-c]+[a-z.]+	351	78,196,736
[a-d]+[a-z.]+	742	113,039,360
[a-i]+[a-z.]+	1569	306,921,472

Table 5.5: Characteristics of different queries

57.98% for the fourth and fifth query, respectively. This returns to two reasons; first the increased number of matches as shown by the number of intermediate key/value pairs in table 5.5. Which in turn results in increasing the sorting and reducing computations of MR from 135 seconds for query1 to 1569 seconds for query5 as shown also in table 5.5. And hence overlapping these computations with the map phase as in HR and IR has higher effect on the overall performance.

Furthermore, the performance of HR gets worse than IR for higher queries i.e., fourth and fifth query. The main reason is that HR generates large number of reduce tasks following tree structure. In addition, the output from any reduce task needs to be sent to the subsequent reduce tasks, so as the size of intermediate key/value pairs increases, the communication overhead increases as well. So for applications producing large number of intermediate results, IR will behave better than HR.

5.6.5 Heterogeneous Environment Performance

Nowadays data centers are becoming incrementally heterogeneous; either due to the use of virtualization technology or machines from different generations. In this experiment, we aim at studying the robustness of MR, HR, and IR to the heterogeneity of the target cluster. In a cluster of 32 nodes, we manually slowed down several nodes i.e., 10 nodes to mimic a heterogeneous clusters having nodes with different generations. We continuously run *dd command* to convert and write a large file (e.g. 5.7 GB) to disk in order to slow down a

given node. This approach was used by Zahria et al. in [52].

We expect in these environments, the map phase time gets longer due to the effects of the slow nodes. So, if the reduce tasks are appropriately assigned to the fast nodes, then utilizing the extra map time in reduce computations could improve the performance of the proposed approaches. Using wordcount, we run MR, and IR with the best configuration achieved in experiment 5.6.3 i.e., 32 reduce tasks for MR and 4 reduce tasks for IR. As shown in Figure 5.10, when the reduce tasks are assigned to the fast nodes, IR becomes better than MR by 10.33% instead of 5.86%. However, when they are randomly assigned, IR becomes better than MR by only 2.32%. This is expected since the IO and computing resources available for reduce tasks in this case become limited, so IR cannot efficiently overlap map with reduce computations. We argue that if the heterogeneity originates from different hardware generations, or from virtualization, it is easy to identify the fast nodes and assign more reduce slots to these nodes, so we can guarantee the improved performance.

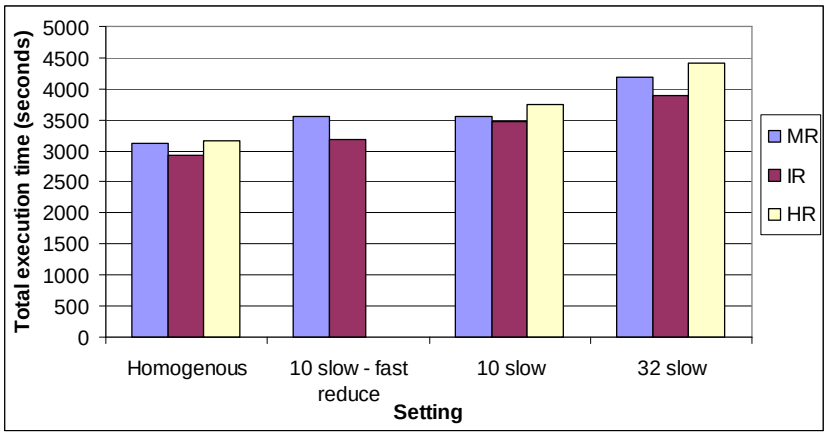

Figure 5.10: Performance in heterogeneous and cloud computing environments using wordcount

Moreover, HR's performance drops significantly when running in heterogeneous environment, this returns to the large number of generated reduce tasks. In addition, it is undeterministic where these tasks will be run, so it is not possible to avoid the effect of the slow nodes.

In a typical cloud computing environments, the computing and io performance of the nodes is lower than what we have in System x. So, we slowed down the 32 nodes to mimic a cloud computing environment. Specifically, using wordcount, IR's performance slightly improves i.e., it is better than MR by 7.14% instead of 5.86% for the homogeneous setting. However, it is less than the performance of heterogeneous setting i.e., 10.33%. The reason is that the reduce tasks get affected by the slow nodes in addition to the map tasks.

5.7 Chapter Summary

In this chapter, we designed and implemented two approaches to reduce the overhead of the barrier synchronization between the map and reduce phases of typical MapReduce implementations. In addition, we evaluated the performance of these approaches using analytical model and experiments on a 32-node cluster. The first proposed approach is the hierarchical reduction, which overlaps map and reduce processing at the inter-task level. It starts a reduce task as soon as a certain number of map tasks complete and aggregates partial reduced results following a tree hierarchy. This approach can be effective when there is enough overlap between map and reduce processing. However, this approach has some limitations due to the overheads of creating reduce tasks on the fly, in addition to the extra communication cost of transferring the intermediate results along the tree hierarchy. To cope with these overheads, we proposed the incremental reduction approach, where all reduce tasks are created at the start of the job, and every reduce task incrementally reduces the received map outputs. The experimental results demonstrate that both approaches can effectively improve the MapReduce execution time; with the incremental reduction approach consistently outperforming hierarchical reduction and the traditional synchronous approach. In particular, incremental reduction can outperform the synchronous implementation by 35.33% for the wordcount application and 57.98% for the grep application

This page intentionally left blank.

Chapter 6

Conclusions

Heterogeneity becomes the trend in designing today's systems; starting from a single chip [6, 37], passing through a single node [79, 78], and ending with large-scale clusters and clouds [85, 42]. Harnessing the computing power of all available heterogeneous resources in the system is a challenging task. In this dissertation, we aim at efficiently facilitating this task within and across resources. Specifically, we leverage the power of MapReduce programming model and OpenCL programming language to achieve our goals. We start with investigating the efficiency of existing MapReduce designs on AMD GPUs, an architecture that is not studied before using previous GPU-based MapReduce implementations. Based on our investigations, we propose an architecture-aware MapReduce implementation targeting AMD GPUs. Due to existing limitations in the OpenCL specifications, our implementation enforces constraint on the number of running threads, thus limiting the utilization of the device. In the second part of this dissertation, we design and implement an efficient MapReduce implementation not only targeting AMD GPUs but also other heterogeneous devices like NVIDIA GPUs. To the best of our knowledge, this is the first portable MapReduce implementation that outperforms state-of-the-art MapReduce implementations on heterogeneous devices. In the third part, we move one step further towards achieving our goal and explore how to efficiently distribute the map/reduce tasks across multiple resources. The major contributions and conclusions

of the three parts are summarized below.

Our investigations reveal that state-of-the-art MapReduce designs are not appropriate for AMD GPUs. These MapReduce designs depend on executing atomic-add operations to coordinate output writing from thousands of concurrently running threads. We realize that atomic operations incur significant overheads i.e., up to 69.4-fold slowdown on AMD GPUs, since it enforces all memory transactions in the kernel to follow a slow CompletePath rather than a fast FastPath. Consequently, we design and implement a software-based atomic operation that does not impact the used memory path. Using this software atomic operation, we implement a MapReduce framework that behaves efficiently and outperforms state-of-the-art MapReduce implementations on AMD GPUs. Specifically, we evaluate this MapReduce framework using three applications that follow different divergence and memory access patterns. The experimental results show that for memory-bound kernels, our software-based atomic add can deliver an application kernel speedup of 67-fold compared to one with a system-provided atomic add. The main shortcoming of the proposed software-based atomic is that it supports limited number of threads.

To address the limitations of the proposed software-atomic operation, we propose and implement an atomic-free design for MapReduce, StreamMR, that can efficiently handle applications running any number of threads. We introduce several techniques to completely avoid the use of atomic operations. Particularly, the design and mapping of StreamMR provides efficient atomic-free algorithms for coordinating output from different threads as well as storing and retrieving intermediate results via distributed hash tables. StreamMR also includes efficient support of combiner functions, a feature widely used in cluster MapReduce implementations but not well explored in previous GPU MapReduce implementations. StreamMR significantly outperforms the state-of-the-art implementation of MapReduce i.e., MapCG by a speedup of between 1.4 to 45. We further optimize StreamMR to work efficiently on other heterogeneous devices not suffering from the penalties associated with the

use of atomic operations like NVIDIA GPUs. Specifically, we propose a mechanism for improving the scalability of the reduce phase with the size of the intermediate output. With the highly scalable reduce phase, StreamMR outperforms MapCG on NVIDIA GPU by up to 3.5-fold speedup.

This dissertation also explores how to efficiently distribute the map/reduce tasks among several resources. The traditional approach is to enforce a barrier synchronization between the map phase and the reduce phase, i.e., the reduce phase can only start when all map tasks are completed. For heterogeneous resources, it is highly expected that the faster compute resources will finish their assigned map tasks earlier, but these resources cannot proceed to the reduce processing until all the map tasks are finished, thus resulting in waste of resources. We propose two approaches to cope with such heterogeneity; the first proposed approach is the hierarchical reduction, which overlaps map and reduce processing at the inter-task level. It starts a reduce task as soon as a certain number of map tasks complete and aggregates partial reduced results following a tree hierarchy. This approach can be effective when there is enough overlap between map and reduce processing. However, it has some limitations due to the overheads of creating reduce tasks on the fly, in addition to the extra communication cost of transferring the intermediate results along the tree hierarchy. To cope with these overheads, we proposed the incremental reduction approach, where all reduce tasks are created at the start of the job, and every reduce task incrementally reduces the received map outputs. Both approaches can effectively improve the MapReduce execution time; with the incremental reduction approach consistently outperforming hierarchical reduction and the traditional synchronous approach. Specifically, incremental reduction can outperform the synchronous implementation by up to 57.98%. As a part of this investigation, we derive a rigorous performance model that estimate the speedup achieved from each approach.

This page intentionally left blank.

Chapter 7

Future Work

In the previous chapters, we presented our efforts towards implementing a MapReduce framework that efficiently exploit all resources existed in today's fat nodes. Specifically, we discussed how to implement an efficient MapReduce framework that is portable across heterogeneous devices. We also explored how to efficiently distribute map/reduce tasks across several resources concurrently. We view the work done in this thesis as groundwork for other potential projects. We discuss them in this chapter.

7.1 CPU/GPU Co-scheduling

Nowadays, most servers and even desktops have at least one GPU in addition to a multi-core CPU. Concurrently exploiting the computing power of these devices is a challenging task. Programmer has to write two versions of an application - one for CPU and another one for GPU. To achieve efficient performance, the code should be optimized to match the architecture of each target device. Appropriately partitioning and scheduling the computation across these devices present another challenge. Fortunately, MapReduce frameworks can help hiding all of these complexities. Most existing efforts [12, 89, 17] that implement MapReduce frameworks to concurrently utilize the CPU and the GPU only report marginal

speedup compared to using only one resource. This performance can be mainly attributed to the overheads of transferring input/output between the CPU and the GPU. With the emergence of the fused architectures [6, 37], the CPU and GPU now share a common global memory. One promising research direction is to investigate the potential of this architecture. Specifically, we need to extend StreamMR framework discussed in Chapter 4 to concurrently make use of the CPU and the GPU. Grouping the map output via distributed hash tables, in addition to the highly scalable reduce phase, makes StreamMR good candidate for the co-scheduling.

7.2 Automatic Compute and Data-Aware Scheduling on Fat Nodes

Current servers do not equipped with only one CPU and one GPU, the trend now is the fat nodes [78, 79] that have more than one multi-core CPU and GPU whether fused or discrete. One crucial question that needs to be answered is which resources are suitable for a given application and how to efficiently distribute the computation among these resources. The answer to this question depends on the running application and the available resources. Specifically, the system should dynamically analyze the characteristics of the running application and the capabilities of the heterogeneous resources. This information should be plugged into a performance model to determine the set of heterogeneous resources to leverage and the ratio of computation assigned to each resource.

A few research efforts proposed the dynamic use of performance models to assign the computations to the heterogeneous resources [30, 14, 15, 59]. They generally base on running the application (either through pre-calibration run or through actual execution) and collecting several profiling information like the performance on different resources using tasks of different sizes. None of them consider the overhead of data transfer among the resources on

their models, despite the fact that the memory transfer alone can slow down the application execution by 2 to 50x [16]. Since all of them target legacy or general code, it is not clear how much input and output produced by each kernel, and how these data will be used afterward.

On the other side, we can leverage the power of MapReduce programming model to quantify the overhead of the data transfer. With MapReduce programming model, we can easily predict the size of the intermediate/final output and the flow of this output along the heterogeneous resources. Specifically, at the start of the application, our framework can execute a pre-calibration run. In this run, few map tasks and reduce tasks are assigned to the heterogeneous computing resources. Based on the execution time of each task, the transfer time of the input/output, the size of the input data set, and the size of the intermediate/final output, the performance model makes a decision regarding the best computing resources to run the target application and the granularity of the tasks assigned to each resource.

7.3 Energy Efficiency of GPU-based MapReduce Implementations

Although the energy efficiency of MapReduce implementations for CPU and clusters of CPUs has been studied by many researchers [84, 91, 90, 56, 92], to the best of our knowledge there is no equivalent studies for MapReduce implementations on GPUs. For GPU-based MapReduce implementation to be candidate for harnessing the computational power of the GPU devices in large-scale clusters and clouds, the energy efficiency of these implementations should be well studied. Most of the current MapReduce implementations for GPUs focus on achieving maximum speedup. It is as important to investigate for different GPU architectures, which implementation is the most energy-efficient.

7.4 Extending Software Atomic Add Operation

In Chapter 3 we proposed a software implementation for atomic-add operations that significantly improves the memory access performance of memory-bound applications on AMD GPUs. For compute-bound applications, it is unclear when our software-based atomic operation can be beneficial. We need to investigate a set of guidelines to decide when to use our software-based atomic operation. Also to address the limitations of supporting a small number of workgroups, we need to study other approaches for implementing the atomic operations that support any number of workgroups. One potential solution is making use of the CPU rather than dedicating the first workgroup to be the coordinator.

This page intentionally left blank.

Bibliography

[1] Hadoop. http://hadoop.apache.org/core/.

[2] Mars Source Code. http://www.cse.ust.hk/gpuqp/Mars.html, Nov 16, 2009.

[3] A. Matsunaga, M. Tsugawa and J. Fortes. CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics. Microsoft eScience Workshop, 2008.

[4] Amazon.com. Amazon Elastic Compute Cloud. http://www.amazon.com/gp/browse.html?node=201590011.

[5] AMD. The Industry-Changing Impact of Accelerated Computing. AMD White Paper, 2008.

[6] AMD. The AMD Fusion Family of APUs. http://www.amd.com/us/products/technologies/fusion/Pages/fusion.aspx, 2011.

[7] AMD. Stream Computing User Guide. http://www.ele.uri.edu/courses/ele408/StreamGPU.pdf, December 2008.

[8] AMD. OpenCL Programming Guide rev1.03. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf, June 2010.

[9] Anastasios Papagiannis and Dimitrios S. Nikolopoulos. Rearchitecting mapreduce for heterogeneous multicore processors with explicitly managed memories. In *39th International Conference on Parallel Processing*, pages 121 –130, sept. 2010.

[10] Advanced Research Computing at Virginia Tech. HokieSpeed (Seneca CPU-GPU). `http://www.arc.vt.edu/resources/hpc/hokiespeed.php`.

[11] D. D. Redell B. N. Bershad and J. R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *5th international conference on Architectural support for programming languages and operating systems*, pages 223–233. ACM, 1992.

[12] Bingsheng He, Wenbin Fang, Naga K. Govindaraju, Qiong Luo, and Tuyong Wang. Mars: a MapReduce Framework on Graphics Processors. In *17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269. ACM, 2008.

[13] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In *Workshop on Software Tools for MultiCore Systems*, 2008.

[14] Cdric Augonnet, Samuel Thibault, Raymond Namyst and Pierre-Andr Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par Parallel Processing*, 2009.

[15] Chi-Keung Luk, Sunpyo Hong and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45 –55, dec. 2009.

[16] Chris Gregg and Kim Hazelwood. Where is the Data? Why you Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proceedings of the IEEE International*

Symposium on Performance Analysis of Systems and Software, ISPASS '11, pages 134–144, Washington, DC, USA, 2011. IEEE Computer Society.

[17] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng and Haibo Lin. MapCG: Writing Parallel Program Portable Between CPU and GPU. In *19th International Conference on Parallel Architectures and Compilation Techniques*, pages 217–226. ACM, 2010.

[18] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[19] Intel Corporation. Intel OpenCL SDK. `http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/`, 2011.

[20] NVIDIA Corporation. NVIDIA GPU Computing Developer Home Page. *http://developer.nvidia.com/object/gpucomputing.html*.

[21] NVIDIA CUDA. CUDA Occupancy Calculator. *http://news.developer.nvidia.com/2007/03/cuda_occupancy_.html*, 2007.

[22] D. P. Playne, K. A. Hawick and A. Leist. Mixing multi-core CPUs and GPUs for scientific simulation software. Technical Report Technical Report CSTN-091, Computational Science Technical Note, 2009.

[23] E. W. Dijkstra. Solutions of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, 1965.

[24] Douglas Thain, Todd Tannenbaum and Miron Livny. Distributed Computing in Practice: the Condor Experience: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, 2005.

[25] Feng Ji and Xiaosong Ma. Using Shared Memory to Accelerate MapReduce on Graphics Processing Units. In *IEEE 25th International Parallel and Distributed Processing Symposium*, 2011.

[26] George Teodoro, Tahsin M. Kurc, Tony Pan, Lee Cooper, Jun Kong, Patrick Widener and Joel H. Saltz. Accelerating Large Scale Image Analyses on Parallel CPU-GPU Equipped Systems. *Center for Comprehensive Informatics, Emory University, Technical Report CCI-TR-2011-4*, 2011.

[27] Christopher Joseph Goddard. Analysis and Abstraction of Parallel Sequence Search. Master's thesis, Virginia Polytechnic Institute and State University, 2007.

[28] gpgpu.org. GPGPU Developer Resources. *http://gpgpu.org/developer*.

[29] Grant Mackey, Saba Sehrish, John Bent, Julio Lopez, Salman Habib and Jun Wang. Introducing MapReduce to High End Computing. In *Petascale Data Storage Workshop Held in conjunction with SC08*, 2008.

[30] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an Execution Model and Runtime for Heterogeneous Many Core Systems. In *17th international symposium on High performance distributed computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.

[31] Khronos Group. The Khronos Group Releases OpenCL 1.0 Specification. *http://www.khronos.org/news/press/releases*, 2008.

[32] Henry Wong, Anne Bracy, Ethan Schuchman, Tor Aamodt, Jamison Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang and Hong Wang. Pangaea: a Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor. In *17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 52–61, New York, NY, USA, 2008. ACM.

[33] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao and D. Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *ACM SIGMOD International Conference on Management of Data*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[34] IBM. IBM OpenCL Development Kit for Linux on Power v0.3. `https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=80367538-d04a-47cb-9463-428643140bf1`, 2011.

[35] Intel. Intel Many Integrated Core Architecture. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`.

[36] Intel. Single-Chip Cloud Computer . `http://techresearch.intel.com/ProjectDetails.aspx?Id=1`, 2009.

[37] Intel. Intel Sandy Bridge. `http://software.intel.com/en-us/articles/sandy-bridge/`, 2011.

[38] Intel. Intel Xeon Phi. `http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html`, 2011.

[39] Jean-pierre Goux, Sanjeev Kulkarni, Jeff Linderoth and Michael Yoder. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Cluster Computing*, pages 43–50. Society Press, 2000.

[40] Jeff A. Stuart and John D. Owens. Multi-GPU MapReduce on GPU Clusters. In *IEEE 25th International Parallel and Distributed Processing Symposium*, 2011.

[41] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems, Design, and Implementation*, 2004.

[42] Jeffrey S. Vetter, Richard Glassbrook, Jack Dongarra, Karsten Schwan, Bruce Loftis, Stephen McNally, Jeremy Meredith, James Rogers, Philip Roth, Kyle Spafford and Sudhakar Yalamanchili. Keeneland: Bringing heterogeneous gpu computing to the computational science community. In *Computing in Science and Engineering*, 2011.

[43] L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[44] Huan Liu and Dan Orban. GridBatch: Cloud Computing for Large-Scale Data-Intensive Batch Applications. In *8th IEEE International Symposium on Cluster Computing and the Grid*, pages 295–305, 2008.

[45] M. Mustafa Rafique, Ali. R. Butt and Dimitrios S. Nikolopoulos. Designing Accelerator-Based Distributed Systems for High Performance. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 165 –174, May 2010.

[46] M. Mustafa Rafique, Ali. R. Butt and Dimitrios S. Nikolopoulos. A capabilities-aware framework for using computational accelerators in data-intensive computing. *Journal of Parallel and Distributed Computing*, 71:185–197, February 2011.

[47] Marc de Kruijf and Karthikeyan Sankaralingam. Mapreduce for the Cell Broadband Engine Architecture. *IBM Journal of Research and Development*, 53(5):10–1, 2009.

[48] Marwa Elteir, Heshan Lin and Wu-chun Feng. StreamMR: An OpenCL MapReduce Framework for Heterogeneous Graphics Processors. *To be submitted to IEEE Transactions on Parallel and Distributed Systems.*

[49] Marwa Elteir, Heshan Lin, and Wu-chun Feng. Enhancing MapReduce via Asynchronous Data Processing. In *IEEE 16th International Conference on Parallel and Distributed Systems*, pages 397–405. IEEE, 2010.

[50] Marwa Elteir, Heshan Lin and Wu-chun Feng. Performance Characterization and Optimization of Atomic Operations on AMD GPUs. In *IEEE Cluster*, 2011.

[51] Marwa Elteir, Heshan Lin, Wu-chun Feng and Tom Scogland. StreamMR: An Optimized MapReduce Framework for AMD GPUs. In *IEEE 17th International Conference on Parallel and Distributed Systems*, 2011.

[52] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[53] Mayank Daga, Ashwin Aji and Wu-chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Symposium on Application Accelerators in High-Performance Computing*, 2011.

[54] Mayank Daga, Tom Scogland and Wu-chun Feng. Architecture-Aware Mapping and Optimization on a 1600-Core GPU. In *17th IEEE International Conference on Parallel and Distributed Systems*, Tainan, Taiwan, 2011.

[55] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *14th annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[56] Michael Cardosa, Aameek Singh, Himabindu Pucha and Abhishek Chandra. Exploiting Spatio-Temporal Tradeoffs for Energy-Aware MapReduce in the Cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 251 –258, july 2011.

[57] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *2nd ACM*

SIGOPS/EuroSys European Conference on Computer Systems, pages 59–72, New York, NY, USA, 2007. ACM.

[58] Michael Linderman, Jamison Collins, Hong Wang and Teresa Meng. Merge: a Programming Model for Heterogeneous Multi-Core Systems. In *13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 287–296, New York, NY, USA, 2008. ACM.

[59] Michela Becchi, Surendra Byna, Srihari Cadambi and Srimat Chakradhar. Data-Aware Scheduling of Legacy Kernels on Heterogeneous Platforms with Distributed Memory. In *22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 82–91, New York, NY, USA, 2010. ACM.

[60] Naga Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin and Dinesh Manocha. Fast Computation of Database Operations using Graphics Processors. In *ACM SIGMOD international conference on Management of data*, 2004.

[61] NVIDIA. NVIDIA Tegra APX Series. `http://www.nvidia.com/object/product_tegra_apx_us.html`.

[62] NVIDIA. NVIDIA CUDA Programming Guide-2.2. `http://developer.download.nvidia.com/compute/cuda/2.2/toolkit/docs/`, 2009.

[63] NVIDIA. NVIDIA OpenCL Implementation. `http://developer.nvidia.com/opencl`, 2009.

[64] David Patterson. The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges. *NVIDIA Whitepaper*, 2009.

[65] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Third Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[66] Wikimedia Foundation project. English-language Wikipedia. *http://download.wikimedia.org/*, 2010.

[67] R. Farivar, A. Verma, E. M. Chan and R. H. Campbell. MITHRA: Multiple Data Independent Tasks on a Heterogeneous Resource Architecture. In *IEEE International Conference on Cluster Computing and Workshops*, pages 1 –10, 31 2009-sept. 4 2009.

[68] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. In *IEEE International Symposium on Workload Characterization*, pages 198–207. IEEE, 2009.

[69] Rob Pike, Sean Dorward, Robert Griesemer and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[70] S. Chen and S. Schlosser. Map-Reduce Meets Wider Varieties of Applications Meets Wider Varieties of Applications. Technical Report IRP-TR-08-05, Intel Research, 2008.

[71] Michael Schatz. Cloudburst: Highly Sensitive Read Mapping with MapReduce. *Bioinformatics*, 25:1363–1369, 2009.

[72] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton and Wen-mei W. Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

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

[74] Shubhabrata Sengupta, Mark Harris, Yao Zhang and John D. Owens. Scan Primitives for GPU Computing. In *22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106. Eurographics Association, 2007.

[75] Shucai Xiao and Wu-chun Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. In *IEEE 24th International Parallel and Distributed Processing Symposium*, pages 1–12. IEEE, 2010.

[76] Steven Y. Ko, Imranul Hoque, Brian Cho and Indranil Gupta. On Availability of Intermediate Data in Cloud Computations. In *12th Workshop on Hot Topics in Operating Systems*, 2009.

[77] J. A. Stuart and J. D. Owens. Message Passing on Data-Parallel Architectures. In *IEEE 23th International Parallel and Distributed Processing Symposium*, pages 1–12. IEEE Computer Society, 2009.

[78] Supermicro. A+ Server 1022GG-TF. `http://www.supermicro.com/Aplus/system/1U/1022/AS-1022GG-TF.cfm`.

[79] Supermicro. Shattering the 1U Server Performance Record. `http://www.supermicro.com/products/nfo/files/GPU/GPU_White_Paper.pdf`, 2009.

[80] Suryakant Patidar and P. J. Narayanan. Scalable Split and Gather Primitives for the GPU. Technical report, Tech. Rep. IIIT/TR/2009/99, 2009.

[81] T. Chen, R. Raghavan, J. N. Dale and E. Iwata. Cell broadband engine architecture and its first implementation - a performance view. *IBM Journal of Research and Development*, 51(5):559 –572, Sept. 2007.

[82] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad and Edward D. Lazowska. *The Interaction of Architecture and Operating System Design*, volume 26. ACM, 1991.

[83] Thomas Scogland, Barry Rountree, Wu-chun Feng, and Bronis de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.

[84] Thomas Wirtz and Rong Ge. Improving MapReduce Energy Efficiency for Computation Intensive Workloads. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –8, july 2011.

[85] top500.org. TOP 10 Sites for November 2011 . `http://www.top500.org/lists/2011/11`, 2011.

[86] Tyson Condie, Neil Conway, Peter Alvaro, Joseph Hellerstein, Khaled Elmeleegy and Russell Sears. MapReduce Online. Technical Report UCB/EECS-2009-136, University of California at Berkeley, 2009.

[87] Steffen Valvag and Dag Johansen. Oivos: Simple and Efficient Distributed Data Processing. In *IEEE 10th International Conference on High Performance Computing and Communications*, pages 113–122, Sept. 2008.

[88] Vasily Volkov and James W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2008.

[89] Wenbin Fang, Bingsheng He, Qiong Luo and Naga K. Govindaraju. Mars: Accelerating MapReduce with Graphics Processors. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):608 –620, April 2011.

[90] Willis Lang and Jignesh M. Patel. Energy Management for MapReduce Clusters. *Proc. VLDB Endow.*, 3(1-2):129–139, September 2010.

[91] Yanpei Chen, Archana Ganapathi and Randy H. Katz. To Compress or not to Compress - Compute vs. IO Tradeoffs for Mapreduce Energy Efficiency. In *Proceedings of the first*

ACM SIGCOMM workshop on Green networking, Green Networking '10, pages 23–28, New York, NY, USA, 2010. ACM.

[92] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur and Randy Katz. Energy Efficiency for Large-Scale MapReduce Workloads with Significant Interactive Analysis. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 43–56, New York, NY, USA, 2012. ACM.