ACCELERATING MAHOUT ON HETEROGENEOUS CLUSTERS USING HADOOPCL

A Thesis Presented

by

Xiangyu Li

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University

Boston, Massachusetts

December 2014

Abstract

MapReduce is a programming model capable of processing massive data in parallel across hundreds of computing nodes in a cluster. It hides many of the complicated details of parallel computing and provides a straightforward interface for programmers to adapt their algorithms to improve productivity. Many MapReduce-based applications have utilized the power of this model, including machine learning. MapReduce can meet the demands of processing massive data generated by user-server interaction in applications including web search, video viewing and online product purchasing. The Mahout recommendation system is one of the most popular open source recommendation systems that employs machine learning techniques based on MapReduce. Mahout provides a parallel computing infrastructure that can be applied to study a range of different types of datasets.

A complimentary trend occurring in cluster computing is the introduction of GPUs which provide higher bandwidth and data-level parallelism. There have been several efforts that combine the simplicity of the MapReduce framework with the power of GPUs. HadoopCL is one framework that generates OpenCL programs automatically from Java to be executed on heterogeneous architectures in a cluster. It pprovides the infrastructure for utilizing GPUs in a cluster environment.

In this work, we present a detailed description of Mahout recommender system

and a profiling of Mahout performance running on multiple nodes in a cluster. We also present a performance evaluation of a Mahout job running on heterogeneous platforms using CPUs, AMD APUs and NVIDIA discrete GPUs with HadoopCL. We choose a time-consuming job in Mahout and manually tune a GPU kernel for it. We also modify the pipeline of HadoopCL from map->reduce to filter->map->reduce that increase the flexibility of HadoopCL in task assignment. Analysis of the performance issues of automatically generated OpenCL GPU program is provided as well as the optimization we make to resolve the issues. We achieve around 1.5 to 2X speedup from using optimized GPU kernel integrated into HadoopCL on a APU cluster and 2X to 4X speedup on a discrete GPU cluster.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The amount of data being generated today is exploding. Analyzing massive data is required by scientific exploration, web search, online advertising, and recommendation systems. The size of a typical dataset is easily over the capacity of a disk on a single machine and the processing time is usually counted in days. Distributed computing frameworks such as MPI [15] emerges as it can efficiently pass data and algorithm as messages for parallel processing. However, this framework requires knowledge of the underlying architectural aspects of a cluster and the programmer is responsible for managing complicated details such as data partitioning and task scheduling.

MapReduce [14] is an approach to ease the burden of the programmer. It provides a simple interface for programmers to express their algorithms while the framework takes care of the complicated details of data distribution and error handling. MapReduce can distribute computation of a large volume of data onto thousands of nodes in a cluster to run in parallel.

Apache Hadoop [2] is an open-source implementation of MapReduce, written in Java, which is widely used in both academia and industry. It provides key infrastructure that includes task scheduling, resource management, fault handling, and additional features to carry out the parallelization of MapReduce computation. Hadoop development community is maintaining Hadoop actively with regular updates and bug fixing, making Hadoop a stable and widely used MapReduce framework.

Many applications have benefited from the massive parallel processing capabilities of MapReduce, including machine learning [18] [13]. The Mahout recommendation system [3] is a prime example of just one of these applications. Mahout utilizes machine learning techniques, processing vast amounts of data to find associations and patterns in the data. One such application of Mahout is in customer rating systems that analyze products a consumer has purchased, and generates a set of recommendations for other products that customers may be interested in purchasing. Built upon Hadoop, Mahout can take a huge dataset as input and output recommendations for millions of customers. However, Mahout is only utilizing the power of CPUs and like many other MapReduce applications, requires a high performance server which costs a lot in hardware and power. We would like to be able to provide Mahout implementations without a large investment in hardware cost and power usage.

GPUs have become the new architecture for general purpose, data-parallel computations. They offer higher bandwidth and computation power, as well as better power-efficiency, for computationally intensive applications than traditional CPUs. Previously, GPUs had been used primarily for graphic applications and image rendering. With advances in GPU programmability, GPUs can accelerate big data processing – we can also consider heterogeneous architectures that utilize the power of both CPU and GPU (APU) to obtain a cost-effective solution.

OpenCL is an open standard for cross-platform parallel programing. It provides an abstract for different hardware architectures to allow executing one program on different platforms including GPU. This significantly increases GPU's programmability which enables the execution of general purpose computation on GPU.

We propose to utilize a heterogeneous architecture that leverages advances in GPU/APU technology, and demonstrate its merit using a Mahout recommendation system. GPUs/APUs are well-suited to the data-parallel characteristics of the processing steps in Mahout with their SIMD architecture. GPU programming and execution under MapReduce has been made considerably easier through the introduction of HadoopCL[19]. We present comparisons of performance of running Mahout tasks on CPU, GPU and APU clusters and provide detailed analysis on how performance is related to the characteristics of the computation and the platform the application runs on. In addition, we propose a modified HadoopCL framework that distributes tasks to different devices based on a programmer's choice, as well as an optimized HadoopCL GPU kernel that achieves speedup versus the original HadoopCL implementation. We are able to achieve 1.5X to 2X speedup from using the optimized HadoopCL GPU kernel on an APU cluster and 2X to 4X speedup on a discrete GPU cluster.

1.2 Contributions

The contributions of this work are:

- We evaluate Mahout running real world applications on multiple nodes, identify the performance and application characteristics.
- We execute Mahout on heterogeneous platforms containing CPU/GPU/APUs

using HadoopCL in a cluster environment, evaluate the performance and identify performance issues present in automatically-generated HadoopCL GPU programs.

- We optimize the execution of HadoopCL to improve resource utilization for Mahout and evaluate the performance potential of running on heterogeneous platforms.

1.3 Organization of the Thesis

The following chapters are organized as follows:

Chapter 2 presents the background of the work. We provide an introduction to GPU/APU architectures, the OpenCL Programming model, the MapReduce Programming model, Apache Hadoop, Apache Mahout, HadoopCL, Aparapi and heterogeneous computation.

Chapter 3 covers related work on leveraging the MapReduce programming model on GPUs.

In chapter 4, we discuss the key algorithms used in Mahout and profile them using real world input data.

In chapter 5, we present Mahout as implemented using HadoopCL. We explore the performance issues with HadoopCL GPU kernels and describe our optimization approach to both GPU kernels and HadoopCL.

In chapter 6, we evaluate the performance of Mahout running on different heterogeneous clusters that include CPUs, discrete GPUs and APUs.

Chapter 7 provide some discussion on the lesson learned and discusses directions for future work.

Chapter 2

Background

This chapter provides background knowledge to help readers understand the material in this thesis. Section 2.1 talks about the role modern GPUs are playing in the area of parallel computing and how it helps improving the performance of data-parallel applications. Section 2.1.3 introduces APU: a fusion of GPU and CPU developed by AMD. Section 2.2 discusses MapReduce framework and its open sourced implementation in Java - Hadoop. We also present an example to illustrate the workflow of a typical Hadoop program. Section 2.3 introduces HadoopCL and Aparapi which HadoopCL uses to generate OpenCL program from JAVA.

2.1 GPU

In this section, we give a brief introduction to the trend of general purpose GPU(GPGPU) computing followed by OpenCL that provides a platform model and a memory model for GPGPU. AMD's APU is also introduced in this section.

2.1.1 GPGPU

The time of improving a processor's performance simply by increasing the operating frequency is gone as it hits the memory wall and the power wall [8]. This motivates the research and developments on multi-core and many-core processors to exploit the data parallelism to improve the overall performance. Single instruction, multiple data(SIMD) is an architecture model that exploits the data parallelism by executing same instruction on multiple items of data in parallel [21]. GPU has emerged as a popular SIMD architecture that utilizes thousands of light weight cores to achieve better performance without consuming much power.

GPU used to be considered as a designated device for rendering 2D and 3D computer graphics. Rendering computer graphics includes independent calculations on every pixel of an image. GPU accelerates the rendering by applying same independent operations on large amounts of pixels for every image achieves good performance to accomplish high frames per second(fps) in real time rendering. Programmers used to write programs to interact with GPUs through standard APIs like OpenGL [6] but the applications on GPUs were restricted to only computer graphics applications by both the programming model and the technique specifications revealed by GPU vendors like AMD [1] and NVIDIA [5].

Modern GPUs are general purpose parallel processors that provides programmers with C-like language models such as OpenCL [4] and CUDA [26]. This allows GPU to process general purpose applications traditionally handled by the CPU and achieve speedups over CPU implementations in orders of magnitude thanks to its high computation power and memory bandwidth. Figure 2.1 shows the theoretical GFLOP/s for NVIDIA GPUs vs Intel CPUs. A typical CPU nowadays has around 4 cores(Intel i7) and up to 16 cores(AMD Opteron 6300), while a normal GPU has hundreds of

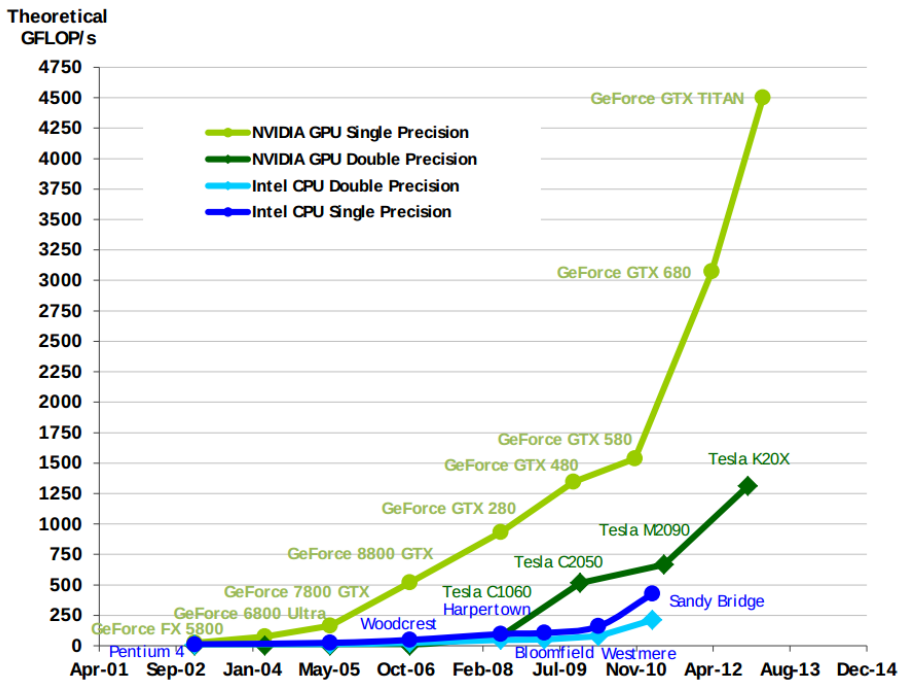

Figure 2.1: Floating-Point Operations per Second for the CPU and GPU

them(AMD Radeon HD 7900 has 1536 streaming processors). GPU devotes most of its resources to the ALU units rather than caches and control flow units. As a result, applications that has massive data parallelism can benefit a lot from using GPU cores concurrently and achieve huge speedup while the ones that runs complicated instructions on a small chunk of data generally work more efficiently on CPUs. Programmers need to learn programming models like OpenCL(will be discussed in following chapters) and CUDA to port their CPU applications to GPUs.

2.1.2 OpenCL

OpenCL is an open standard for cross-platform parallel programming[4] developed and maintained by Khronos consortium. The application programming interface(API)

provided by OpenCL uses C language wrapped by C++ Wrapper API and it provides an abstract hardware model for different architectures. Programmers develop and optimize applications based on the abstract which OpenCL compiles and distributes to different platforms such as CPUs, GPUs and ARM processors. OpenCL defines a platform model describing the relationship between processing units in a heterogeneous system, and a memory model that abstract the memory hierarchy of the platforms which resembles current GPU memory hierarchies [16]. These two models are discussed in following sections.

Platform Model

OpenCL refers to the CPU as the "host" and the GPU as the "device" in a heterogeneous system since the CPU coordinates the execution of a program on one or more GPUs while each GPU acts as an accelerator that runs the data-parallel operations in the program. Figure 2.2 provided by OpenCL specification 2.0 [4] shows the host-device platform model of OpenCL. A GPU contains a certain number of compute units, and each compute unit is composed of a fixed number of processing elements. Take AMD Radeon HD 7970 for an example, there are 32 compute units in the GPU, each of which contains 4 SIMD units. Each SIMD unit contains 16 lanes which corresponds to the processing elements in OpenCL platform model. Thus one compute unit contains 4*16=64 processing elements. As a results, AMD 7970 can schedule up to 32*64 = 2048 instructions at a time. On each compute unit, one instruction is executed by 64 concurrent threads (or work-item as termed in OpenCL standard) at the same time. The group of these 64 work-items are called a "wavefront" as each "wave" are executed together in lock-steps. The set of instructions executed by the work-items on the GPU is called kernel which is written by programmers using

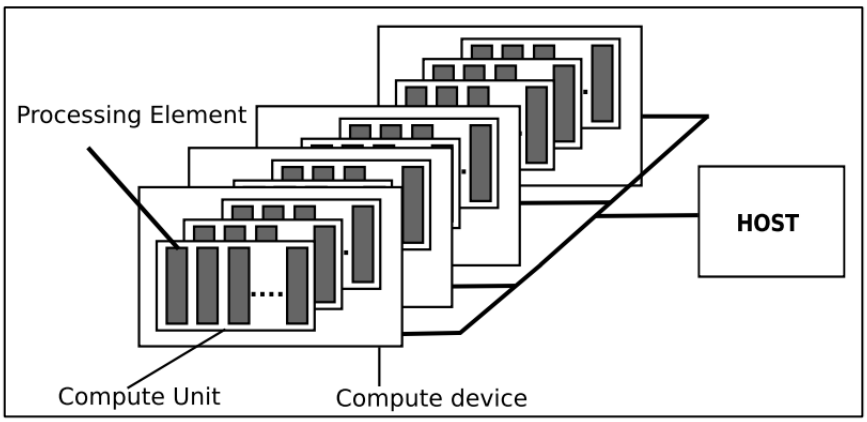

Figure 2.2: OpenCL host-device platform model

OpenCL programming model.

Memory Model

The OpenCL memory model is composed of two parts: the host memory on the CPU side (typically the system's main memory) and the device memory on the GPU side as shown in Figure 2.3. The device memory communicates with host memory through PCIe bus. The device memory contains three levels: global memory (part of which is used as constant memory), local memory and private memory. The global memory is the only place communicating with the host memory. The global memory serves as the first place for the input data transferred from the host memory and the last place for the output data before being sent back to host memory. Thus it is visible by all work-items across the compute units on the device. Global memory of a GPU is usually an off chip memory that has a large size but also introduces transferring latency. Local memory is on chip memory that is small but provides high bandwidth and low latency and only visible to the work-items within the same compute unit. Private memory again is fast and even further limited as it is only visible to individual

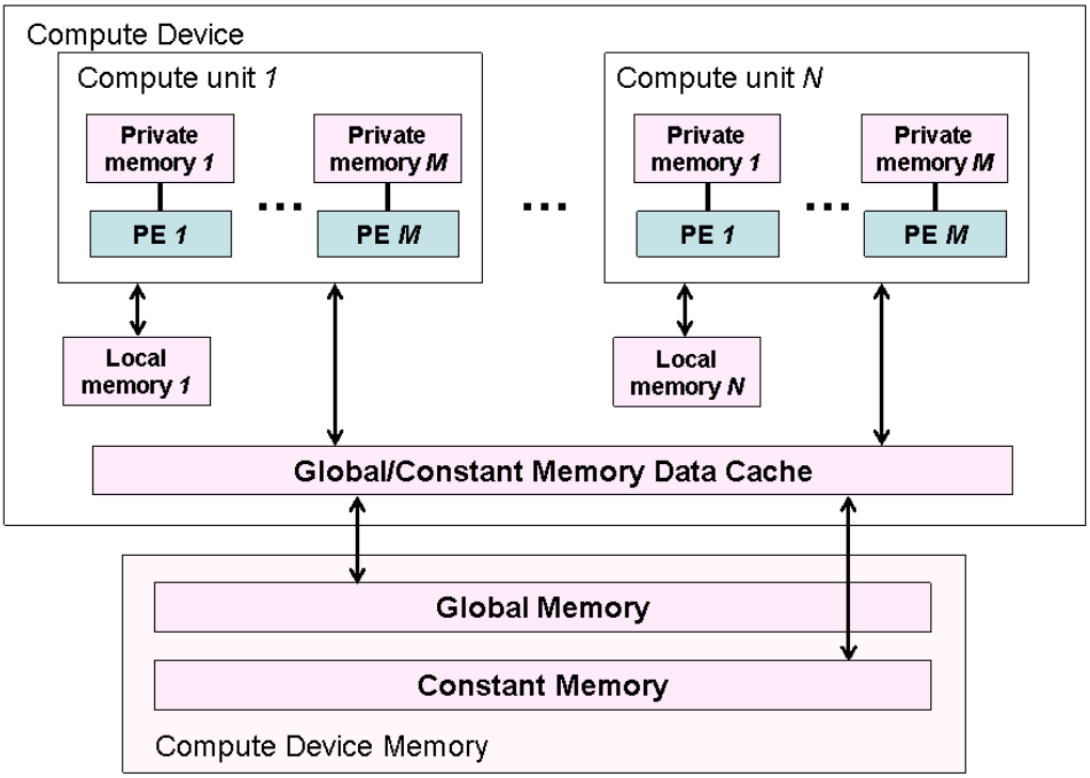

Figure 2.3: OpenCL memory model

work-item. Private memory is always mapped to single registers on a GPU.

A typical GPU program flow starts by transferring the input data to global memory first, then each compute fetches the data from the global memory for processing, or to transfer it to the local memory and accessing it from local memory for to reduce accessing latency. The private memory is used to store small data like function arguments and iteration counters. Utilizing local memory is a common approach when it comes to improving the performance of an application if the data is being used frequently. The output data is then placed on the global memory to be transferred back to the host memory. Efforts are always made on transferring a bulk of data at once between the host memory and the global memory to hide the extra PCIe

transferring latency. AMD's APU is an attempt to eliminate the transferring latency and it will be discussed in the following section.

2.1.3 APU

Traditionally a GPU is separated from CPU and they communicate through PCIe bus while using separate memory addressing space. This involves latency in transferring the data between the CPU memory to GPU memory through the PCIe bus. In order achieve better performance programmers always need to tune the size of data transferred at a time so that the latency can be hidden by the throughput of bulk data transferred and computed.

AMD's APU(Accelerated Processing Unit) integrates a GPU with a CPU on the same die making them sharing same board and resources. GPU and CPU share same physical memory and address space as well. This fusion design saves power as well as reducing latency as the data movement only takes place within the chip.

2.2 MapReduce and Hadoop

2.2.1 MapReduce Overview

The conventional approach of distributed computation on huge data requires a lot of efforts. The user need to manually partition the data to be distributed and send/receive each chunk of data to/from multiple remote machines, carefully apply operations on different partitions in parallel, design a scheduler to balance the workload among machines and recover from failed tasks should any machine not work properly. The time devoted to managing these is always longer compared to how much it actually costs to design the actual algorithm.

MapReduce [14] is a programming model that distributes a large volume of data onto thousands of nodes in a cluster for parallel computation. As shown in Figure 2.4, it provides an interface for programmers to express their algorithm in a functional way. The top part of the Figure shows a typical program flow on a single machine. The input is operated by algorithm that runs in serial and generates the output. In MapReduce, the algorithm can be mapped to a MapReduce process that runs on multiple nodes as shown below. A typical MapReduce process contains three phases: Map, Shuffle and Reduce where the user is required to express the serialized algorithm in the form of two functions map() and reduce(). The MapReduce library can take care of the complicated data partitioning, task scheduling, parallelization and error handling automatically. This model turns out to be very flexible and can be applied on a wide range of applications.

The data used in a MapReduce process contains input data which is fed to the map function, intermediate data generated by map function before being fed to reduce function, and the final output data generated by reduce function. All these data are in the format of <key, value> pairs and a pair is also called a record. In the Map phase, user-defined map function applies operations to each record which generally alters the data and increase the data size. In the shuffle phase, MapReduce framework shuffles the records and clusters the values associated with the same key to generate a new set of <key, list[values]> records where every key is distinct. In the Reduce Phase, each record are sent to reduce function where the values in the list are typically converged to one value so that output data is again in the form of <key, value> records.

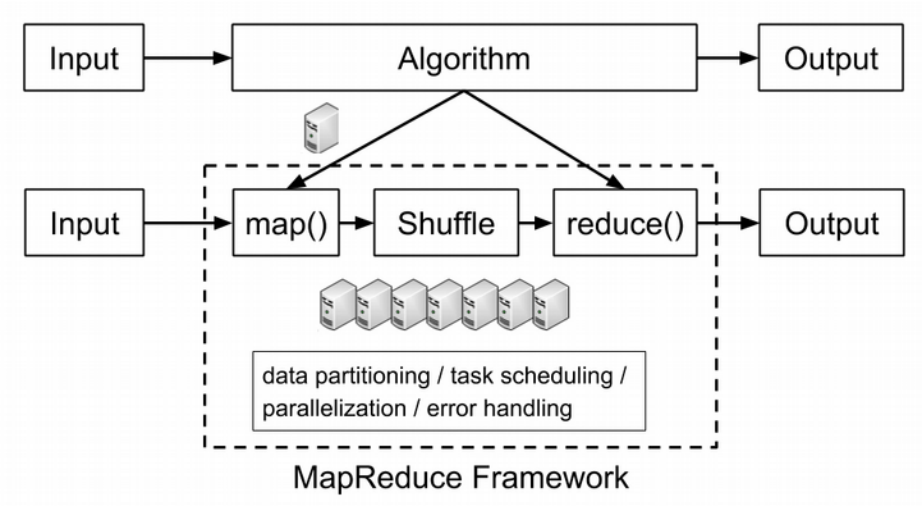

Figure 2.4: MapReduce Overview

2.2.2 MapReduce Framework

The MapReduce framework is based on a JobTracker located in the master node (NameNode) and one task tracker on each slave node (DataNode) as shown Figure 2.5. The MapReduce program is submitted to the JobTracker which initializes the job with the input splits read from the Hadoop Distributed File System(HDFS). The JobTracker then assigns tasks to the data nodes and they maintain a heartbeat communication for assigning new tasks and returning finished tasks. The TaskTracker launches a child JVM which either runs a map task or a reduce task using the data also from the HDFS. The child JVM is released after a task is done, the results are written back to HDFS and the JobTracker is noticed so the next task can be assigned.

2.2.3 A Word Count Example

Figure 2.6 illustrates how MapReduce framework works using a word count example. Given a text file, a word count program counts the number of every distinct word in the file and outputs a list of each word associated with its count. The algorithm is

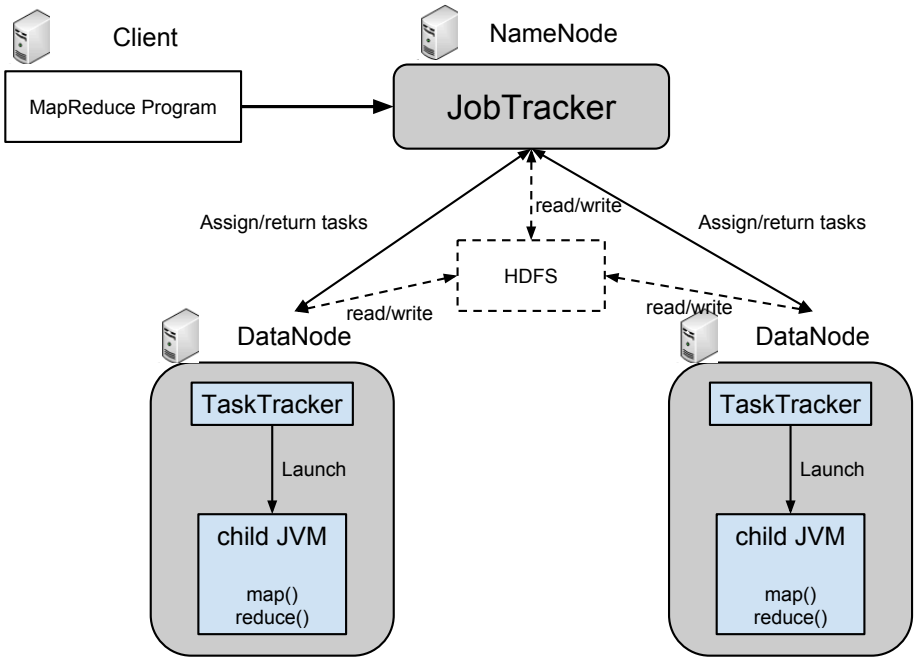

Figure 2.5: MapReduce Framework

fairly simple on a single machine: the text file is read in and for each word, we store it in a hash table as the key while its value is incremented each time the word is seen again later. However, once the text file becomes too big to fit in a single machine's memory or the processing time is too long, we need to distribute this one file over to multiple machines for parallel processing. In this case, more problems are introduced: how to partition the data to maintain a balanced distribution, how to synchronize every hash tables on every computing node, how to merge the partial results back to form a single output file and so forth.

The MapReduce library handles the problems mentioned above implicitly for the programmers as long as they can cast their program into functional styles using map() and reduce(). As shown in Figure 2.6, the input data is a text file containing the words to be counted. Assuming we have two computing nodes, MapReduce framework splits the data into two halves and sends each half to each node to feed map(). The

programmer can define the map() in such a way that it outputs each word as a key with a count of "1" associated as the value. Then these key-value records are shuffled by the framework which generates <key, list[values]> records where keys are sorted and mapped to a list of values that are associated to the same key. Taking "Hello" as an example, before the shuffle step, there are two "Hello"s from two nodes each associated with a count "1", the shuffle phase group those two values together by generating a list of them and associates it with the key "Hello": <Hello, [1,1]>. The intermediate data is split and sent again to every node where user-defined reduce() accumulates the count and generates a new record - for instance - <Hello, 2>. The framework again gathers the output from each node and merge them together and send to a single node where the final output can be retrieved.

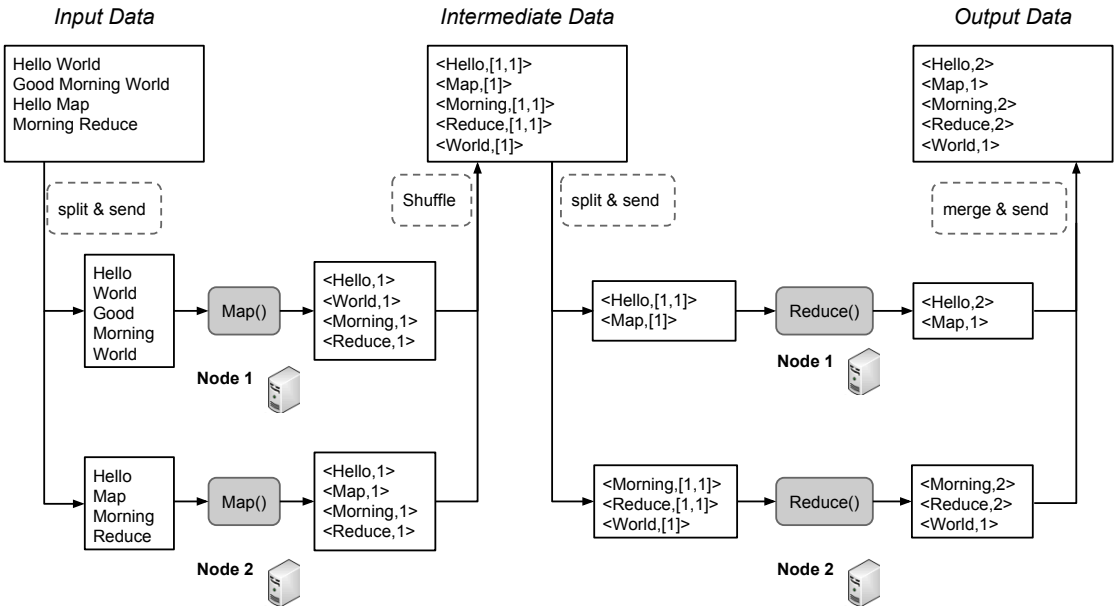

Figure 2.6: MapReduce: Word Count Example

2.2.4 Hadoop

Hadoop [2] is an open source implementation of the MapReduce framework written in Java. The simplicity and productivity of Java programming and MapReduce make Hadoop widely used in both academia and industry. By using Hadoop, a programmer can simple cast their problem to fit the map() and reduce() steps, while Hadoop offers the infrastructure to carry out the parallel computation, task scheduling, and resource management.

The simplicity and robustness of Hadoop offers great facilities to process massive data on large scale clusters. However, Java provides the high productivity at the cost of loss of efficiencies. Hadoop launches JVMs frequently on short-lived tasks, the serialization and deserialization in Hadoop I/O and Java Child process start-ups introduce a lot of overhead. On the other hand, the lack of native language support (C and OpenCL) of Hadoop also presents a barrier to optimize certain types of computation-intensive applications using massive parallel accelerators such as GPUs and emerging heterogeneous architectures such as APUs.

2.3 HadoopCL

HadoopCL [19] is an extension of Hadoop that integrates OpenCL with Hadoop to utilize a wide range of architectures to provide better performance. It automatically translates user-defined Java kernels into OpenCL that runs on both CPUs and GPUs. The translation of Java to OpenCL is done by Aparapi [17], an open source tool that offers JIT compilation to translate Java bytecode to OpenCL. HaoopCL extends Hadoop's Mapper and Reducer classes in order to allow programmers to make minimal modifications to their code written in Hadoop to execute applications on

heterogeneous platforms. HadoopCL uses dedicated threads to handle asynchronous communication between CPUs and GPUs to maximize the utilization of the available bandwidth, and hide data transfer latency. HadoopCL tries to abstract away most of the details of GPU programming, so that an average programmer can write MapReduce-style code and execute it on GPUs without having any knowledge of GPU architecture or OpenCL.

Figure 2.7 is a high-level system diagram of HadoopCL in a single DataNode cited from [19]. HadoopCL introduces a TaskTracker that launches multiple child JVMs to run jobs on both Multi-core CPUs and Many-core GPUs. Each child JVM grabs one HDFS chunk of <key,value> records and execute a map or a reduce task on it. Two Java threads are spawned: the main thread reading input data from HDFS, launching asynchronous OpenCL kernels and retrieving the output, and the child thread writing the output back to HDFS. The asynchronous kernel launching overlaps the kernel execution on the device with the data transferring while two Java threads in the same JVM add another layer of parallelism as the main thread can start next round of kernel launching while the child thread is processing the current batch of output data.

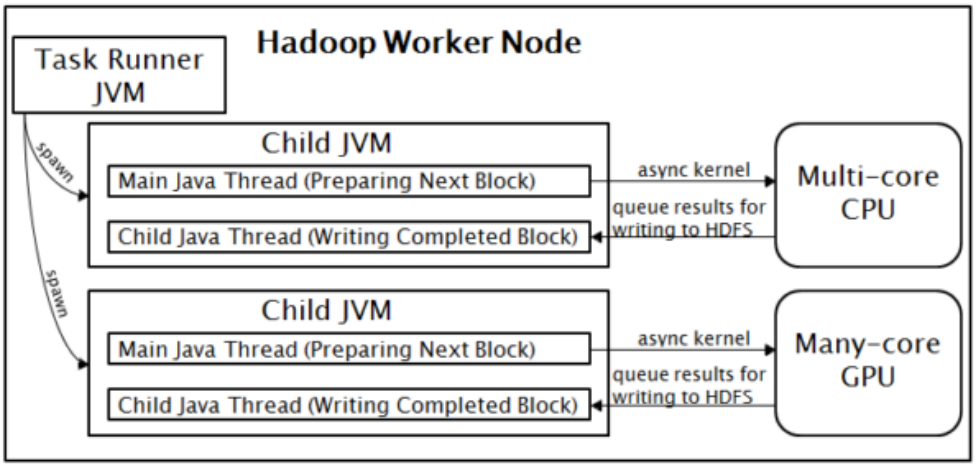

Figure 2.7: System diagram for HadoopCL

2.3.1 Aparapi

Aparapi [17] is an open source library developed by AMD. It provides an interface that dynamic translate Java bytecode into OpenCL kernels, allocates device memory, transfers data forth and back between host and device and handles kernel declaration and invocation that is hidden from the Java programmer. Thus an average Java programmer does not need any specific knowledge related to GPU or OpenCL before executing a Java program that utilizes the power of heterogeneous platform.

Chapter 3

Related Work

A lot of efforts on utilizing heterogeneous architectures for MapReduce jobs have been made. The majority of works focus on utilizing heterogeneous platforms while reducing communication cost. While some frameworks offers high flexibility that require the programmer to understand GPU programming to tune the code in order to achieve high performance, others try to hide the complexity of GPU programming and focus on increasing programmability and productivity.

Mars [20] is the first attempt on using a GPU for in MapReduce framework. It is a standalone MapReduce framework written in C++ and CUDA [26], another GPU programming language that can only be used on NVIDIA devices exclusively. It has the same goal as HadoopCL which is to hide the complexity of GPU programming by offering a MapReduce interface. It only supports single GPU workload, and it executes two additional pre-processing kernels to decide the exact location of output written to GPU buffers. This is a trade off because of the lack of atomic operation supports on GPU by the time the paper is written.

GPMR [29] is another project that utilizes clusters of GPUs for massive processing

of MapReduce workloads. It is also a standalone MapReduce implementation written in C++ and CUDA. It changes some aspects of original MapReduce mechanism, such as adding partial reduction and accumulation to the MapReduce pipeline. The efforts are mainly devoted to improving the efficiency of communications between multiple GPUs. It requires CUDA programming knowledge for optimizing an application and it doesn't demonstrate the work on multi-core CPUs that are idle when GPUs are running.

Chen et al [12] [11] follows a different path to improve MapReduce performance on both CPUs and GPUs. It introduces two different MapReduce schemes for different types of tasks - a map-dividing scheme and a pipelining scheme. The map-dividing scheme treats CPU cores and GPU computing units as equivalent workers and distribute workload evenly onto every worker. The pipelining scheme treats two devices distinctively where one device is assigned map operations while the other is used for reduce operations. It utilizes GPU's low-latency local memory to achieve better bandwidth and it also supports runtime tuning to keep load balance and to reduce scheduling overhead.

Catanzaro et al [10] proposes a standalone MapReduce framework running on NVIDIA GPUs with CUDA. It requires the programmer to provide a list of functions to be used by the framework, as well as a set of parameters specifying variables such as dimensions of work-groups, arrays to be dynamically allocated and arrays to be placed in local memory. It also uses parallel reduction in reduce phase. The framework doesn't hide the complexity of GPU programming from programmers completely, but it provides a scheme that helps achieve good performance with less efforts. They apply the framework on five different machine learning benchmarks and achieve 5x to 32x in speedup in SVM training and 120x to 150x speedup in SVM classification.

HAPI [24] is a similar approach as HadoopCL to utilize heterogeneous platform for MapReduce workload. It also uses APARAPI to generate OpenCL code to map the computational intensive part of a MapReduce job onto a GPU to achieve speedup. It supports Hadoop by providing a new mapper class which requires the programmer to define three functions: preprocess(), gpu() and postprocess(). The preprocess() function converts the Hadoop input objects to its own objects that can be directly fed to a GPU. The gpu() is the function whose operations are translated to OpenCL language by APARAPI and are executed on a GPU. The last postprocess() function then converts the HAPI's own data objects back to Hadoop's native objects. HAPI achieves over 80x speedup for an nbody benchmark. However, HAPI only supports a heterogeneous mapper on a single device while the real world Hadoop applications are always executed on multiple devices across many machines. Thus HAPI doesn't consider any network communication cost which plays an important role in a general Hadoop application. It also only utilize one CPU managing one GPU execution model, leaving other CPU cores idle.

Chapter 4

Mahout Recommendation Engine

In this chapter, we begin in section 4.1 with a brief introduction to Mahout. In section 4.2 we discuss details of the Mahout recommendation engine and present profiling data collected on a cluster of machines.

4.1 Introduction to Mahout

Data is exploding in the Internet era. Analyzing and processing the volumes of rapidly growing data poses a challenge. The computational complexity of new applications is able to easily saturate a single CPU, as well as overwhelm a single machine's disk space. Mahout [3] is an approach to increase the scalability of computation on big data. It is an open source machine learning library based on the MapReduce/Hadoop programming model which provides an underlying infrastructure for distributed computing on thousands of nodes on a cluster. Three machine learning techniques are implemented in Mahout: recommendation engines, clustering and classification. In this work, we focus on the recommendation engines.

4.2 Mahout Recommendation Engine

4.2.1 Collaborative Filtering Recommendation Algorithms

Recommendation algorithms make personalized recommendations for products to customers and are being widely used on e-commerce websites. One major challenge for recommendation systems is the huge amount of information that needs to be processed, typically including hundreds of millions of customers and product items. Collaborative filtering techniques are commonly used in recommendation engines [23].

Conventional collaborative filtering is user based. It collects ratings for items from the users. Later, it can recommend items to a user by looking for the user's *neighbors* (other users with similar taste), and recommends items that neighbors have purchased or positively rated. An example of user-based collaborative filtering is illustrated in the left side in Figure 4.1. One problem of this approach is that the information of all users needs to be processed to find potential neighbors for a user [22]. This approach does not fit the requirements of recommendations in terms of the kind of interactivity typically needed; the other problem is the accuracy: new customers have very limited information and they could have too many neighbors whose preferred items are too scattered to serve as an accurate recommendation.

Alternatively, item-based collaborative filtering techniques [25, 28] have been proposed to address this issue. In this approach, the relationship between items is explored instead of those between users. The time-consuming determination of item preferences/similarities is computed offline. Whenever the system has to recommend new items to a user, instead of searching for all possible potential neighbors through the whole database, only the items which this user likes are used to match in the database to find the similar items. These similar items are then recommended to

the user. This approach is illustrated on the right side of Figure 4.1. Item-based collaborative filtering increases the scalability when working with rapidly growing datasets, without sacrificing accuracy. Mahout uses this filtering technology for its recommender system, which is discussed in this chapter.

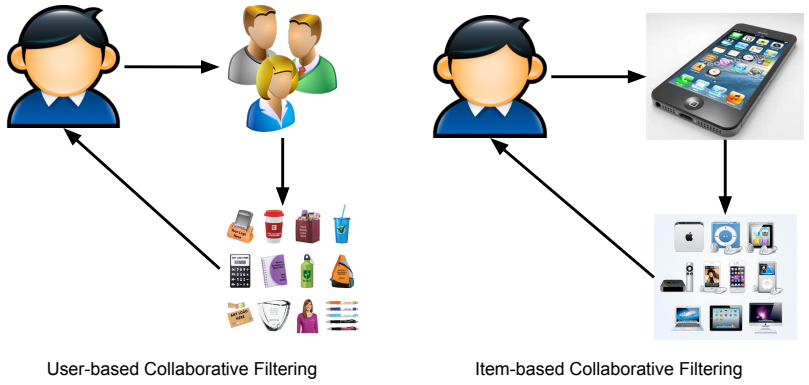

Figure 4.1: User-based versus Item-based Collaborative Filtering Techniques

An item-based collaborative filtering consists of two main steps: a similarity computation step and a recommendation computation step, as illustrated in Figure 4.2. The input is called a rating matrix. A rating matrix is an $m * n$ matrix, where m represents the number of users, and n is the number of items. Row i contains user i's ratings for all the items, where r_{ij} represents the rating on item j from user i. Each rating is within a numerical range, but can be 0, indicating that the user has not rated the item. In the similarity computation step, a similarity value s_{ij} between items i and j is computed by first isolating the users who have liked both items, and then computing the similarity between them. As shown in Figure 4.2, to compute the similarity between items p and q, we only look at the users who have rated both items (u_2, u_s and u_m in this case). Items p and q are called a co-occurred pair given that they were both rated in some user's rating list. The similarity is computed for every pair of co-occurred items i and j. The similarity values can be represented using a

similarity matrix: an $n * n$ symmetric matrix, where the entry (i,j) and entry (j,i) represent the similarity s_{ij} between items i and j. Note that s_{ij} is only calculated if both items i and j are liked by some user, versus computing it for every pair of i and j. The fact that items i and j are both liked by the user indicates some similarity between them, especially if they are both liked by many users. If items i and j never co-occur in any user's rating list, the entry (i,j) in the similarity matrix is equal to zero.

In the recommendation computation step, a recommendation value R_{ui} is computed on an item i for a user u by computing a weighted sum: the sum of ratings on every item j that is similar to item i, weighted by their similarity s_{ij}. As shown in Figure 4.2, user t has ratings on items p, q and n, denoted by shading in the rating vector. To compute the recommendation value of item 1 for user t, the similarities between item 1 with items p, q and n (shaded in the similarity vector) are used to weight user t's ratings. The recommendation value is a weighted sum, computed as:

$$R_{t1} = r_{tp} * s_{1p} + r_{tq} * s_{1q} + r_{tn} * s_{1n}$$

This is the dot product between user t's rating vector and item 1's similarity vector.

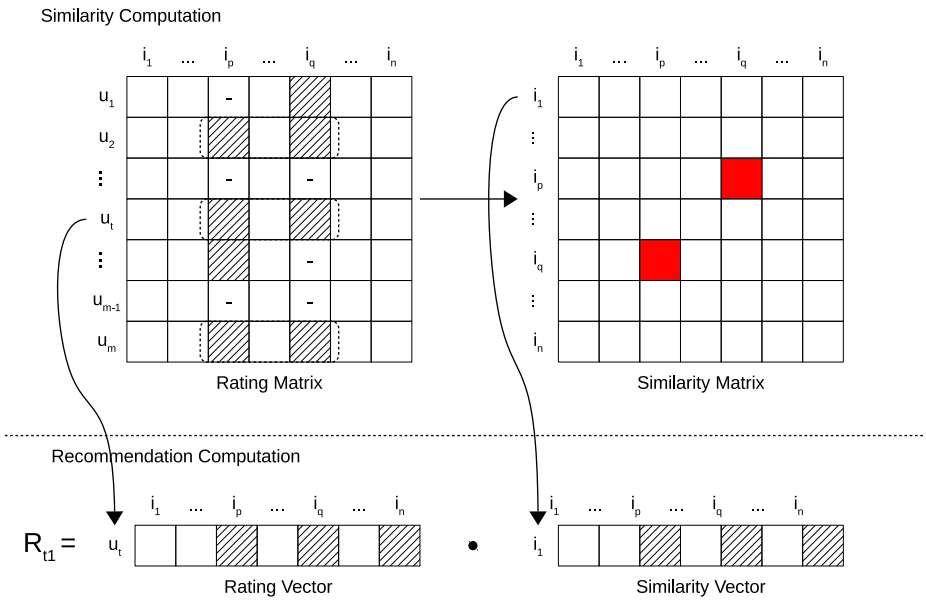

Figure 4.2: Overview of Mahout

4.2.2 Mahout Recommender Overview

The Mahout recommendation[27] engine is an item-based recommendation system. Figure 4.3 gives an example of a Mahout workflow for a movie rating dataset. The input is millions of user-item-preference triplets { *userID: itemID, preference* } and each of which represents a rating of a user to a movie. For example, { *1001: 3, 3.0* } indicates that user 1001 rates movie 3 with a score of 3.0.

Mahout gathers all the itemIDs associated with the same userID to form a "user vector" for that user. Figure 4.3 presents an example of four user ratings for 6 movies. Note that the user vectors are just another name for the rating matrix discussed in 4.2.1, with rows representing userIDs, and columns representing itemIDs. The user vectors matrix is first transposed into an item vector matrix, with rows representing itemIDs, and columns representing userIDs. The item vector matrix is

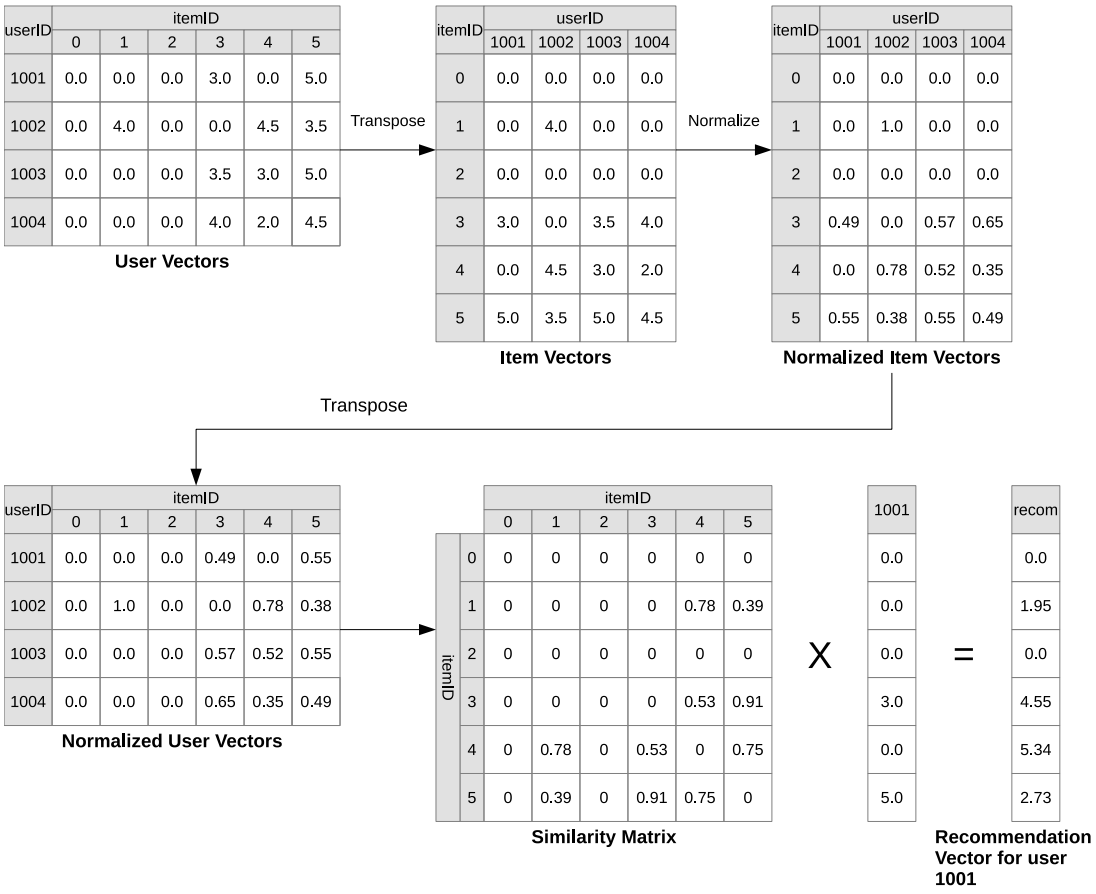

Figure 4.3: An Example of a Mahout Recommendation.

then normalized by assigning $norm_j = v_j / \sqrt{\sum_j {v_j}^2}$ to every j^{th} entry in a row, for every row. The normalized item vector matrix is transposed again to generate a new normalized user vector matrix. In theory, we can generate the normalized user vectors by just applying the formula discussed before. However, in the Hadoop framework, a row of the matrix is treated as <key, list[values]> records that must reside on the same machine, while a column could be scattered on different computing nodes. Thus, to normalize a column of elements, we must transpose the column into a row to ensure the accessibility within a computing node. That is why we need to generate to an item vector matrix before performing normalization.

Mahout then generates the similarity matrix from the normalized user vectors. As discussed in 4.2.1, to compute the similarity between i and j, we only look at the users who have rated both i and j (i and j are labeled as co-occurring). Only co-occurred entries are assigned a similarity value. For example, item 3 and 5 co-occur in user 1001, 1003 and 1004's vectors. The entry for (3,5) and (5,3) are then assigned a value that is calculated, based on the number of times they co-occur in some user's vector, as well as based on the ratings given by that user. In this example, cosine similarity is used.

With the user vectors and the similarity matrix at hand, we can obtain the output of the Mahout recommender. The output is captured in a recommendation vector in the format { *userID:* $item_1$, $score_1$*;* $item_2$, $score_2$*;* , where a user is recommended with multiple items and the score indicates how often each item is recommended. Thus, the higher the score, the stronger the recommendation. This vector is obtained by multiplying the similarity matrix with a transposed user vector. Each element in the vector is a weighted sum of the ratings using the similarities that were discussed in 4.2.1. To recommend an item to user 1001, the similarity matrix is multiplied by user 1001's transposed user vector. As we can see from the user 1001's user vector, he already shows a preference for items 3 and 5, so that the recommender will recommend something he does not know: items 0-2,4. The final results show that item 4 is recommended, as it has the highest recommendation score. The score "5.34" is obtained by computing a dot product of the 4th row of the similarity matrix with the column vector of the user vector 1001. A non-zero entry from the 4th row of the similarity matrix is the similarity between the item and item 4. As discussed before, the higher the value of the entry, the more similar the item is to item 4; on the other hand, a non-zero entry from the user vector indicates how strongly he likes each

item (0 to 5). Again, the higher the score is, the more this user likes the item. The final recommendation score for item 4 is the sum over every product of two entries from each vector. Thus, if two items are similar (i.e., they have a high value in the similarity matrix), and they are also the favorite items of the user (they have high preference value in the user vectors matrix), the item obtains a high recommendation score. In this example, item 4 is similar to both item 3 and 5. In addition, user 1001 likes both items 3 and 5 (with 3.0 and 5.0 in the user vector). As a result, item 4 is the highest recommendation for user 1001.

In the final recommendation vector, we first discard the recommendation for items 3 and 5, as the user 1001 already shows preferences for these two items and we are recommending that the user consider something new. The best recommendation is then item 4, with a score of 5.34, followed by item 1 with a score of 1.95.

In order to handle enormously growing amount of data, Mahout leverages its scalability using the MapReduce framework implemented by Hadoop. This enables the algorithm discussed above to adapt to the framework. We discuss the details of Mahout in the next section.

4.2.3 Mahout Recommender Break Down

The Mahout recommender is based on the MapReduce model. Mahout recommender chains 8 MapReduce jobs together, as shown in Figure 4.4. There are three tasks that contain multiple MapReduce jobs, shown in the figure enclosed in three dotted boxes: PreparePreferenceMatrixJob, RowSimilarityJob and Partial MultiplyJob. They follow a chronological order from left to right. Inside each dotted box, MapReduce jobs are represented by small solid boxes which also follow a chronological order, from top to bottom. An arrow indicates the direction of data flow, from one job to another.

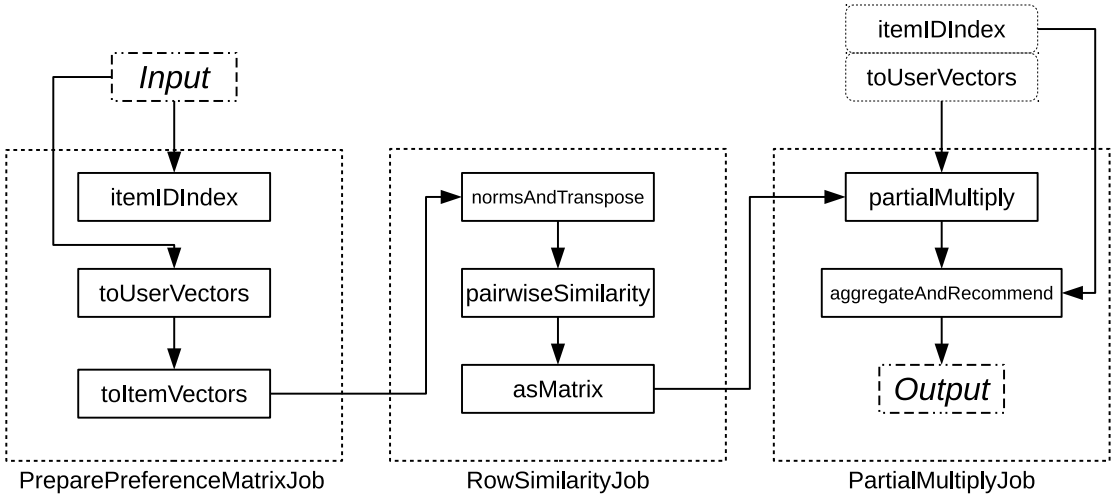

Figure 4.4: Mahout Workflow

Note that the two jobs on the top right corner are not new jobs, but instead from the first task. They are placed there to illustrate that their output data is fed to the jobs in the third task. The three tasks cover Mahout recommender's three main steps: 1) generation of the user vectors, 2) generation of the similarity matrix and generation of the final matrix-vector multiplication. In this section we give a detailed analysis of each job.

The first job is called *itemIDIndex*. This job improves Mahout's robustness as it handles invalid itemIDs that are out of range by applying a conjunction operation between the itemID and 0x7FFFFFFF. The input is the original file that contains { *userID: itemID, preference* }. The itemID from the file is extracted and a mapping of this old itemID together with a new itemID *itemID: internalItemID*, is generated as the output.

The second job *toUserVectors* generates the user vector matrix. This job takes the lines of { *userID: itemID, prefVal* } from the original input data and gathers itemIDs that are associated with the same userID. For example, take user 1002 in

Figure 4.2: the input triplets of { *1002: 1,4.0* }, { *1002: 4,4.5* } and { *1002: 5,3.5* } represent three items that are associated with the same user 1002. In this job, these three triplets are combined into { *1002: 1,4.0; 4,4.5; 5,3.5* }. This vector is the user vector for user 1002. This procedure is repeated for every user and the output is the user vector matrix as shown in Figure 4.3.

The third job *toItemVectors* transposes the userVectors to *itemVectors*, as shown in Figure 4.3. The itemVectors map every userID to every itemID using preference values. For example, using item 3 as an example, user 1001, 1003 and 1004 show preferences of 3.0, 3.5 and 4.0 for this item. The toItemVectors job then associates those users and their preferences with this item, and generates the vector: { *3: 1001,3.0; 1003,3.5; 1004,4.0* }.

The three jobs described above together form the *PreparePreferenceMatrixJob* task in Mahout. It pre-processes the raw input by eliminating invalid itemIDs, generating the userVectors, and generating the itemVectors. These data are then fed to the following jobs.

The fourth job *normsAndTranspose* takes the output of the last job as input and normalizes the preference values for every associated itemID. The normalized preference values are called weights, calculated as $w_i = v_i / \sqrt{\sum_i^n {v_i}^2}$. One weight value is calculated for every user associated with the same item, where v_i is the ith preference value, and n is the total number of users that have rated this item. For example, the output from last job — { *3: 1001,3.0; 1003,3.5; 1004,4.0* } — is normalized and becomes { *3: 1001,0.49; 1003:0.57; 1004,0.66* }. A higher weight indicates a relatively higher preference for the item by the user versus all other users. Then the mapping of { *itemID: userID, weight* } is transposed to { *userID: itemID, weight* }. These normalized user vectors are used in the next job to generate the

similarity matrix.

One of the major jobs in Mahout is *pairwiseSimilarity* job. It takes in the normalized user vectors and generates the similarity between every pair of items: it first counts the number of times a pair of items i, j have been liked by a user, and applies the user's normalized preference for that pair to generate the similarity $s(i, j)$, which is then assigned to the similarity matrix.

The similarity matrix { $item_1$*:* $item_2$*,* $similarity_{12}$*;* $item_3$*,* $similarity_{13}$ can contain a huge amount of data. However, if two items only co-occur once or twice in the whole user vector, their similarity is relatively low. Maintain information for thee pairs introduces a lot of I/O overhead for Hadoop. The sixth job *asMatrix* reduces this overhead by only keeping K (a value set by the user) items with the highest similarity values for each item in the matrix. This job is very time-consuming because the process of generating the top K similarities involves a lot of sorting operations.

The three jobs described above form the second task, the RowSimilarity job. This task generates the similarity matrix which will be used together with the user vector matrix generated from the first task during the rest of Mahout's process.

The last task *PartialMultiplyJob* includes two jobs: *partialmultiply* and *aggregateAndRecommend*. This task multiplies the similarity matrix with every user vector to generate one recommendation vector for every user. The role of the multiplication is not to compute a conventional dot-product between every column in the similarity matrix with a transposed user vector across every user vector, because this would require a whole column of the matrix being available in a single machine. Given the typical size of this matrix, it is impossible to have all the data on a single node. Instead, the ith column of the similarity matrix is multiplied by the ith element in

the user vector j, generating a partial vector P_{ij}. We can then sum up all the vectors P_{ij}s from every column of the matrix to generate the recommendation vector for user j. The same recommendation vector is generated for every user. In order to do that, we need to gather the column i of the similarity matrix and the ith element in the user vector for every user. The *partialmultiply* job contains two mappers that collect those two kinds of data and one reducer to combine them, thus it does not consume a lot of time. The runtime grows linearly with the size of the dataset. The output of the partialmultiply job is then fed to the last job aggregateAndRecommend, which aggregates the preference values for every item across every user and outputs the recommendation vector: { *userID: item*$_1$*, score*$_1$*; item*$_2$*, score*$_2$*; ...* }. AggregateAndRecommend job is another time consuming job in the whole mahout process.

4.2.4 Performance Evaluation of Mahout

In this section we present a detailed performance evaluation of every job that is part of the Mahout recommendation system. In our experiment we use a Wikipedia dataset as input. The Wikipedia dataset consists of 5 million pages located at: `www.wikipedia.org`. The input is composed of lines of plain text in the form of { *pageID: linkID* } pairs, where the pageID represents a page and the linkID represents a second page that is linked to the original page. The Mahout recommender suggests new links to every page by generating a list of { *pageID: link*$_1$*, score*$_1$*; link*$_2$*, score*$_2$*; ...* } for every pageID, based on the similarities among these links. The score represents the estimated preference (relavance in this case) of a page to a link. This dataset fits Mahout's user-item-preference model. A link on the page indicates that the content in the link is relevant to the content of the page, thus a page can be considered as a

CPU	AMD A8-3850
# CPU Cores	4
Clock Rate	2.9GHz
L1 Cache	128KB
L2 Cache	1MB
System Memory	4GB

Table 4.1: Details of the experimental platform used in this thesis.

user and the links on that page are the items a user likes. A preference is expressed as a boolean value, since there is no measurement or scale measuring how strongly or weakly a link is related to a page. For consistency, we will use the terms *user* and *item*, instead of page and link in the rest of the thesis.

The hardware specification for our experiment is listed in Table 4.1. A cluster of 9 AMD Llano A8-3850 APU nodes is deployed with 8 nodes as DataNodes, used for computation and one node as the NameNode, managing task scheduling, data partitioning and resource management for the DataNodes.

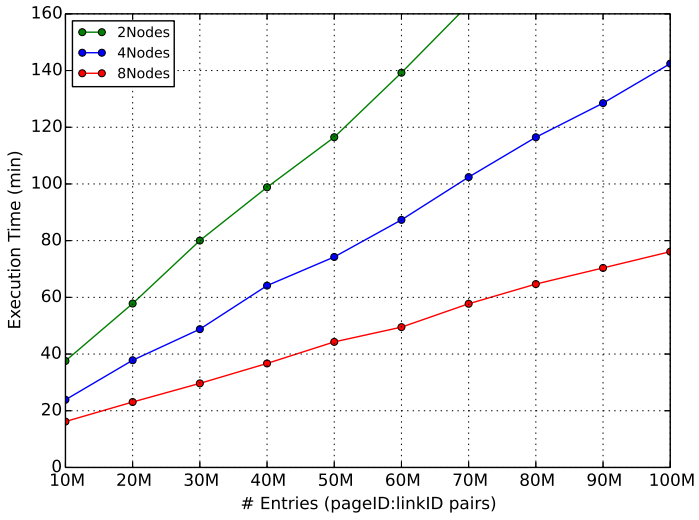

Figure 4.5: Execution time for variable-sized data sets, as run on 2, 4 and 8 nodes.

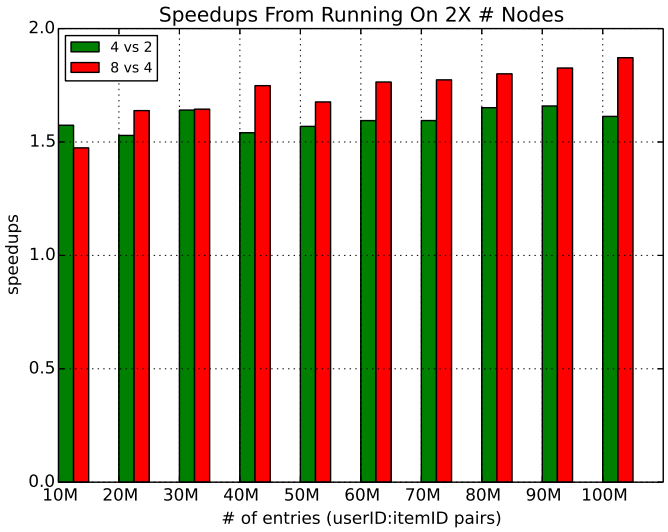

Figure 4.6: Speedups obtained running on 2X the number of nodes.

Figure 4.5 presents the runtime of Mahout for different size input data (the size of the input is measured by the number of userID:itemID pairs) on 2, 4 nodes and 8 DataNodes. We selected the data size for our experiments based on an empirical study that showed that computation can hide Hadoop overhead when the data size exceeds 10 million values. We see very good scalability, with runtime increasing linearly with the number of data elements. We also see favorable scalability as we increase the number of nodes; doubling number of nodes increases performance 1.5-2X, as shown in Figure 4.6.

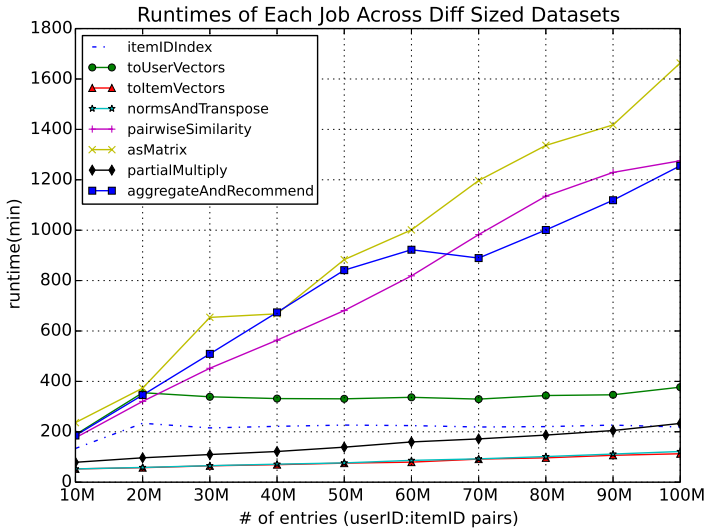

Figure 4.7: Execution time of each job across different dataset sizes.

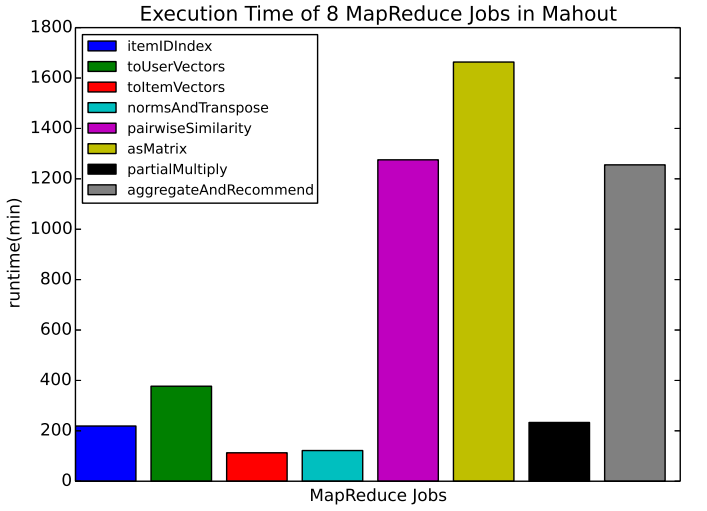

Figure 4.8: Runtime comparison when running on 8 DataNodes with 100 million entries.

Figure 4.7 represents the time spent on each MapReduce job while varying the input datasets running on 8 DataNodes. Five jobs (toUserVectors, toItemVectors, itemIDIndex, partialMultiply and normsAndTranspose) take a relatively small part in the overall time as compared to the remaining three jobs (asMatrix, pairwiseSimilarity and aggregateAndRecommend).

Figure 4.8 presents a comparison of the execution time of each job running on 8 DataNodes, and with a 100 million entry input. The execution time of the five small jobs is constant with a slope $kof \approx 1$ for a runtime $y = kx + b$. The three larger jobs runtimes increase much faster and result in much longer execution times as we scale the size of the data set. As a result, our work on improving Mahout performance using GPUs will focus mainly on optimizing these three jobs. The next chapter of this thesis considers how best to accelerate the pairwiseSimilarity Hadoop job using HadoopCl on a GPU.

Chapter 5

Mahout on GPUs

In this chapter, we present the work on improving Mahout performance by running one of the time-consuming MapReduce job: pairwiseSimilarity on multiple GPUs in a cluster using HadoopCL. We first discuss the algorithm of the pairwiseSimilarity job and its original implementation in Mahout in section 5.1. We then present the original naive GPU implementation generated from HadoopCL in section 5.2 and evaluate its performance. In section 5.3 we present the optimized GPU algorithm we develop and integration of optimized GPU kernel into HadoopCL.

5.1 PairwiseSimilarity Job in Mahout

5.1.1 Overview

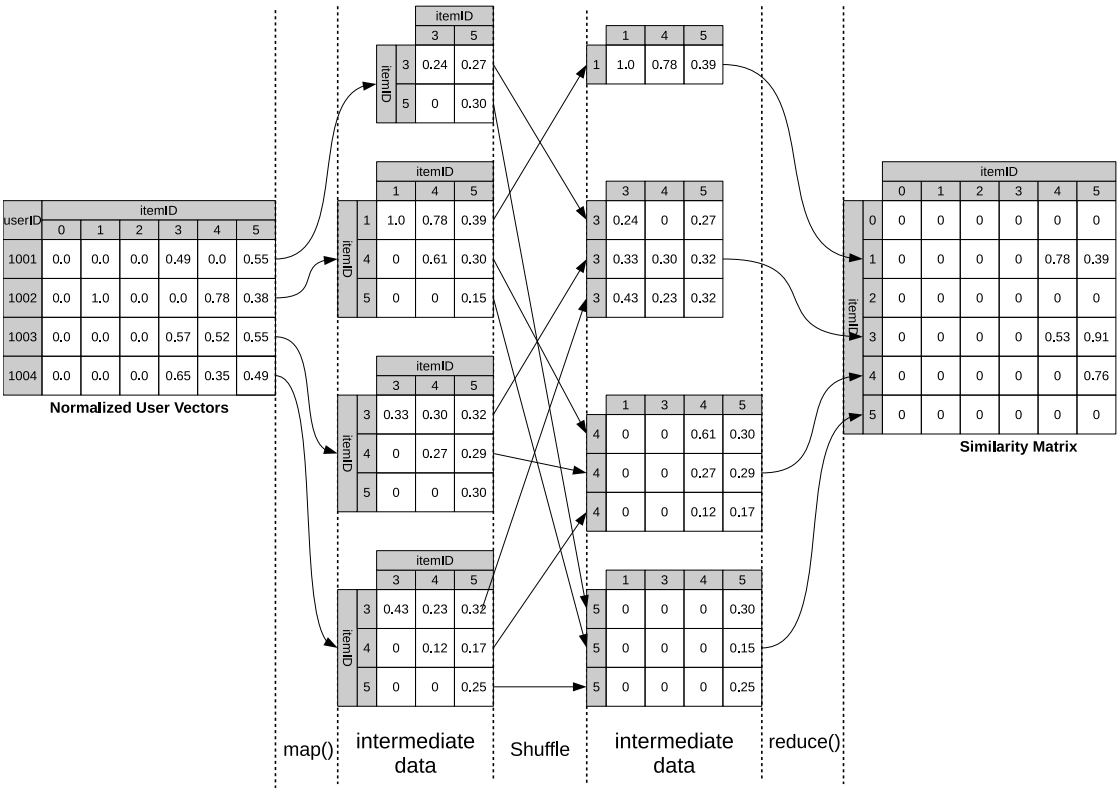

Figure 5.1: PairwiseSimilarity Job Workflow

As the name indicates, the PairwiseSimilarity job is to find the similarity between a pair of items for every pair data set. As discussed in the previous section, the PairwiseSimilarity job converts the "Normalized User Vectors" matrix to a Similarity Matrix by computing the similarity between items i and j, for every i and j that co-occurred in some user's user vector. Figure 5.1 provides a detailed description of this job and the associated data flow.

One map function is applied on exactly one user vector. In this example, we

have four user vectors for users 1001-1004, thus four map functions are launched in parallel. Each job works on one user vector. The output is a partial similarity matrix recording the similarities between every pair of items co-occurring in this user vector. For instance, if user 1002 shows a preference for three items (1, 4 and 5), then the output generated from 1002 is a triangular matrix with similarities of (1,4), (1,5) and (4,5) assigned to the corresponding entries.

To merge the four partial similarity matrices into a single matrix, the MapReduce framework first shuffles the records so the keys are sorted, and then the values associated with the same key value are gathered. The shuffled data is listed from key 1 to key 5.

In the reduce phase, each chunk is again assigned to one reduce function, where the similarity values of the same itemID are summed up, and the similarity between the itemID and itself are discarded. Multiple reduce functions can be run in parallel. After the reduction phase in complete, each record is reduced and sent back to the NameNode, filling in the unified Similarity Matrix.

5.1.2 Mahout Implementation of PairwiseSimilarity

Data Format

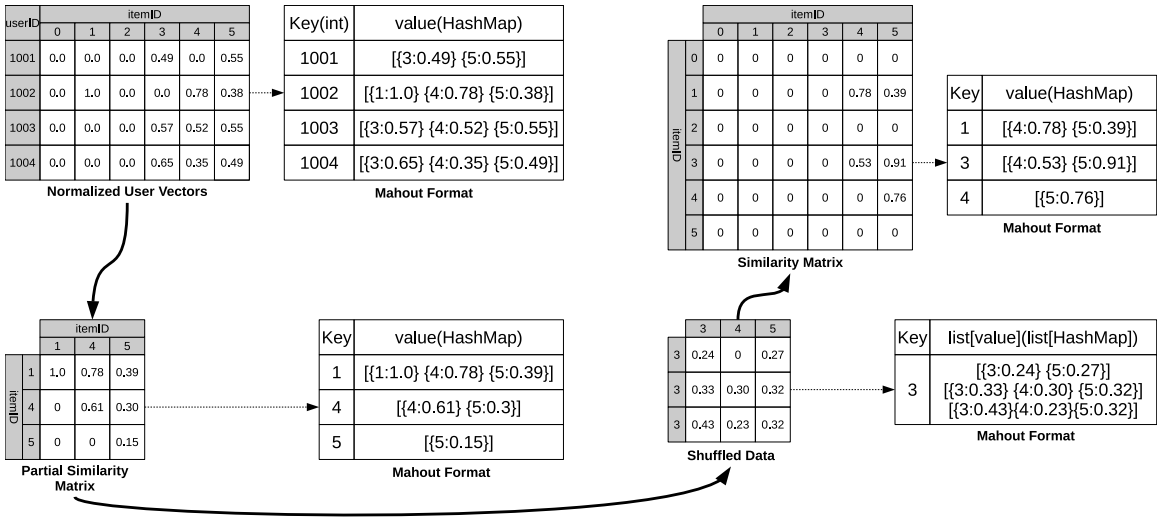

Figure 5.2: A description of the PairwiseSimilarity data format in Mahout.

Figure 5.1 presents a mathematical representation of the data used in Mahout. We cover this format first to help explain the Mahout algorithm. As discussed in section 2.2, MapReduce data uses a format of <key,value> pairs. Since both the user vectors matrix and similarity matrix are very sparse, only non-zero values are kept in the Hadoop Distributed File System. This helps save memory and disk space. Figure 5.2 shows the data format Mahout uses to store the data during each phase corresponding to Figure 5.1. Mahout stores the key values using a Hadoop class named IntWritable, which wraps an Integer-typed data structure together with associated functionality such as serialization (to handle data compression) and data transfer in MapReduce framework. The value is stored in a hash map, which is also wrapped in a Hadoop class named VectorWritable.

The normalized user vectors are presented as a <key, value> pair, where the key

represents the userID. The normalized preference values associated with the user are stored in a hash map, with the itemID as the index, and the preference value as the value. Note that only non-zero values are stored, since a preference value of 0 does not provide any useful information. The map function discards the userIDs and generates <key, value> pairs, where the key represents the itemID, and the value capturing the similarity between the key item and all other co-occurred items. Again, zero similarity values are discarded. The shuffled data is in a slightly different data format: instead of a <key, value> record, it is stored in a <key, list[value1, value2, ...]> record. This record gathers all the values associated with the same key to form a list. The list is output together with the key. Every chunk of shuffled data only has one key, which further saves memory and disk space. The final output similarity matrix uses the same data format as the partial similarity matrix.

Algorithm

The pseudocode for the map() and reduce() functions of the pairwiseSimilarity job shown in Algorithm 1 and Algorithm 2, respectively. The input to the map() function is composed of a userID as the key, a hash map that contains itemIDs as indices, and preference values as values. The hash map is firstly sorted based on the indices. A two-level nested for loop forms the major part of the map function. For every itemID i in the hash map, all the itemIDs j that come after i, including $j = i$ are inserted into a hash map (similarityHashMap) s as indices. The preference value of i is multiplied by that of j, and is inserted as values. Each hash table s is input to the MapReduce framework as the values, with the i values as the keys. The "Partial Similarity Matrix" in Figure 5.2 shows this format. After the Shuffle phase, a list of all the hash tables associated with the same itemID is generated. The"Shuffled Data"

shown in Figure 5.2 includes a list of the three hash maps associated with item 3.

Algorithm 1 The map() function in Mahout.

```
1: function MAP(IntWritable<userID> u, VectorWritable<itemIDPrefHashMap>
   i)
2:     sort(i)
3:     for every itemID in i do
4:         initialize a similarityHashMap s
5:         itemI = itemID
6:         prefI = i.get(itemI)
7:         for every itemID in i that is bigger than itemI do
8:             itemJ = itemID
9:             prefJ = i.get(itemJ)
10:            simiIJ = prefI * prefJ
11:            s.insert(itemJ, simiIJ)
12:        end for
13:        Output <itemI, s>                          ▷ fed to Shuffle phase
14:    end for
15: end function
```

The shuffled data is fed to the reducer. The logic of the reduce() function is very simple. The reduce() receives a key and a list of hash maps as arguments, iterates over every hash map in the list and to produce a new merged hash map, and outputs a record of the <key, value> pair, where the key is the input itemID and the value is the hash map. The "Similarity Matrix" in Figure 5.2 serves as an example.

5.2 PairwiseSimilarity On HadoopCL

Previous work showed that the mapper takes more than 80% of the total execution time of the whole job. Therefore, we implemented the map() function of the pairwiseSimilarity job in HadoopCL to investigate how we can achieve better performance.

Algorithm 2 The reduce() function in Mahout.

```
function REDUCE(IntWritable<itemID> i, VectorWritable<list[similarityHashMap]>
l)
    Initialize a FinalSimilarityHashMap f
    for every similarityHashMap s in the list l do
        for every entry e in s do
            if e.index exists in f then
                f[e].value = f[e].value + e.value
            else
                f.insert(e.index, e.value)
            end if
        end for
    end for
end function
```

5.2.1 Data Format

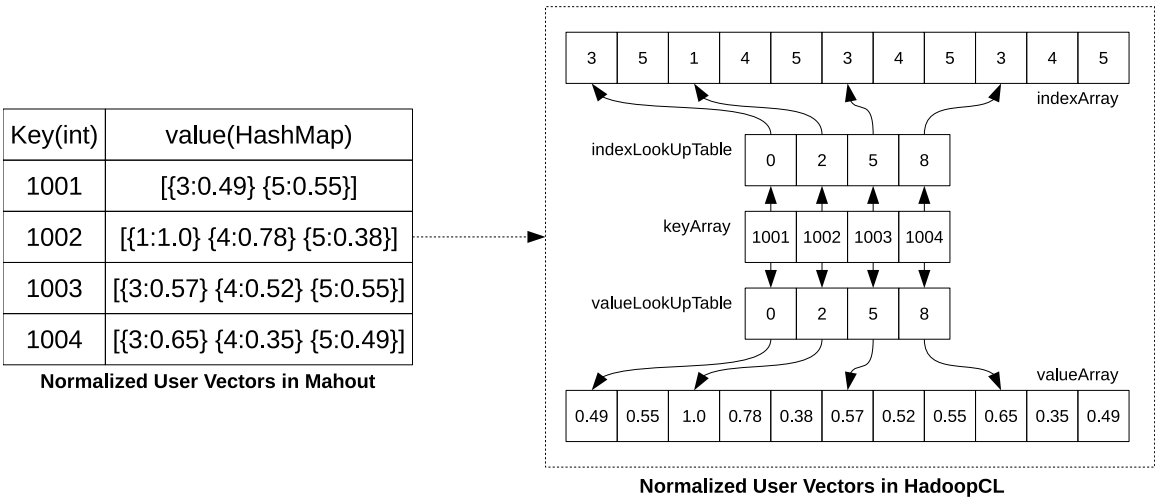

Figure 5.3: The PairwiseSimilarity data format in HadoopCL.

Mahout utilizes hash maps to store the sparse data commonly seen in machine learning datasets. However, implementing a hash map on a GPU requires significant effort to implement and produces limited performance benefits. A parallel implementation of a hash map generation involves a number of atomic operations and search operations

[7]. There present is no library support for hash maps on a GPU in Aparapi [17]. As a result, HadoopCL utilizes a different data structure to replace hash maps.

HadoopCL implements five basic typed arrays, as shown in Figure 5.3 that works together to replace hash maps. The data on the left is the Normalized User Vectors data represented in Mahout. In HadoopCL, we have an integer array named keyArray which stores the keys (userIDs in this case) in a linear array. Instead of using the Java Iterator class in Mahout to store the hash maps, HadoopCL uses two linear arrays - indexArray and valueArray - to store the indices and values of multiple hash maps separately. To find the hash map associated with a specific key, the indexLookUpTable and valueLookUpTable methods are used. These tables are created based on prefix sum algorithm [9] and can point to the beginning and ending locations of the indices and values of a hash map based on the location of the key in the keyArray. For example, to find the hash map associated with key 1002, we use the location of 1002 in the keyArray - which is 1 - to index both tables. The methods indexLookUpTable[1] and valueLookUpTable[1] point to the beginning location of the hash map, while methods indexLookUpTable[2] and valueLookUpTable[2] point to the ending location. Note that the two lookup tables are not always equal. There exists a one-to-many mappings between indices and values in a hash map.

5.2.2 Algorithm

To run the map() function in HadoopCL, a programmer only needs to make small changes in the original Java map() function. Most of the changes are needed in order to adapt to the replacement of hash maps used in Mahout. Algorithm 3 presents the differences between a Mahout map function and a HadoopCL map function. They are:

Algorithm 3 map() function in HadoopCL

```
1: function MAP(int userID, int[] indices, double[] values, int length)
2:     sort(indices, values, length)
3:     int[] indicesBuffer = allocInt(length)                    ▷ temporary buffer
4:     double[] valuesBuffer = allocDouble(length)               ▷ temporary buffer
5:     for every itemID in indices do
6:         itemI = itemID
7:         prefI = preference value associated with itemI
8:         int allocCnt = 0          ▷ the size of output that will be allocated
9:         for every itemID in indices that is bigger than itemI do
10:            itemJ = itemID
11:            prefJ = preference value associated with itemJ
12:            simiIJ = prefI * prefJ
13:            indicesBuffer[allocCnt] = itemJ                   ▷ temporarily stored
14:            valuesBuffer[allocCnt] = simiIJ                   ▷ temporarily stored
15:            allocCnt++                                        ▷ calculate output size
16:        end for
17:        int[] indicesOutput = allocInt(allocCnt)                          ▷ allocate
18:        double[] valuesOutput = allocInt(allocCnt)                        ▷ allocate
19:        for i from 0 to allocCnt - 1 do                    ▷ write output from temp
20:            indicesOutput[i] = indicesBuffer[i]
21:            valuesOutput[i] = valuesBuffer[i]
22:        end for
23:        Output <itemI, indicesOutput, valuesOutput, allocCnt>  ▷ fed to Shuffle
   phase
24:    end for
25: end function
```

1. A different signature at line 1: HadoopCL uses two primitive arrays and the length of the array instead of a hash map wrapped in VectorWritable in Mahout. The use of the arrays is discussed in Figure 5.3.

2. Different output buffers at line 17 and 18: same as the input hash map, HadoopCL uses indicesOutput and valuesOutput to replace the hash map output in Mahout.

3. Temporary buffers at line 3 and 4: since the size of the output is not know beforehand, two temporary buffers - i) indicesBuffer and ii) valuesBuffer - are provided.

The two temporary buffers are allocated with the maximum size that could possibly be needed. The results are temporarily written to the buffers at line 13 and line 14 as a preprocessing step. In this step, the actual allocation size needed for the output is determined. The output buffers are then allocated with the actual allocation size. The last step is to transfer the data from the temp buffers to the output buffers, as shown between 19 and 22. Finally, the buffers are output to feed the shuffle phase, along with the allocated size. HadoopCL wraps these values in a vectorWritable class written to HDFS.

The map() function in Java is translated into OpenCL code by Aparapi. HadoopCL provides "glue code" which changes the memory access pattern and task-to-work-item mapping. HadoopCL aggregates multiple <key, value> records and launches a map() or reduce() function on one work-item in OpenCL. The goal is to fully utilize the massive parallelism available on a GPU. Since each task operates only on a single <key, value> record, thousands of tasks can be executed concurrently. The HadoopCL glue code also changes the access pattern and data layout to accommodate coalesced

memory access.

5.2.3 Performance Evaluation

We perform a preliminary performance evaluation of running the PairwiseSimilarity job on 2 APU nodes, with one as the NameNode and one as a DataNode. The NameNode is the same machine we used for evaluating Mahout. We changed the DataNode to a AMD A10 APU as it supports double precision data types on its GPU natively. The node configuration can be found in Table 5.1. The job runs in two different modes: OpenCL in CPU-only mode and OpenCL on a GPU. The execution time is shown in Figure 5.4. The performance achieved running on GPU is 2X slower than the CPU. This is mostly due to the poor performance of the sorting algorithm, atomic operations and random memory access on the GPU kernel. These operations are not very expensive to run on a CPU. In next section we will present a detailed analysis of the performance issues and provide an optimized OpenCL kernel to be integrated into HadoopCL.

	NameNode	DataNode
CPU	AMD A8-3850	AMD A10-6700
GPU	AMD Radeon HD 6550D	AMD Radeon HD 8670D
# CPU Cores	4	4
# GPU Cores	5	6
CPU Clock Rate	2.9GHz	3.7GHz
GPU Clock Rate	600MHz	844MHz

Table 5.1: The APU cluster hardware used in this work.

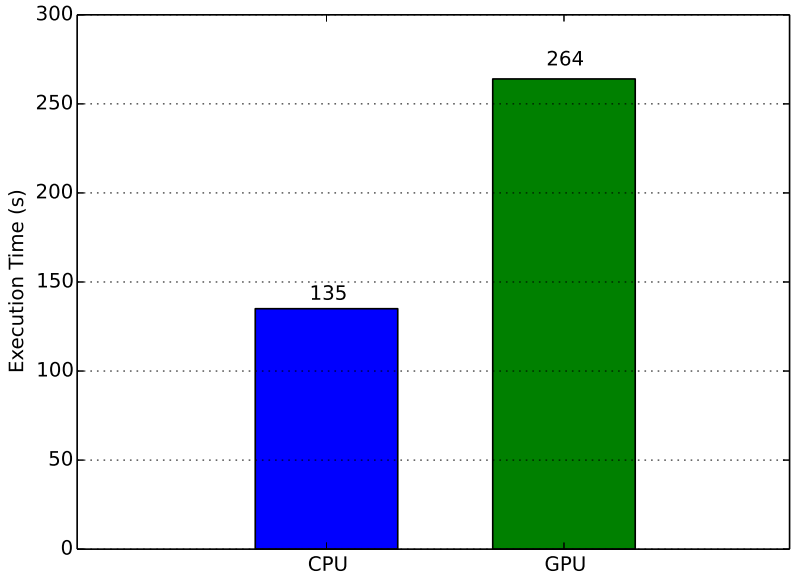

Figure 5.4: Execution time of two HadoopCL modes.

5.3 PairwiseSimilarity Job Optimized GPU kernel

In this section we present an optimized OpenCL kernel and how we integrate it into HadoopCL.

5.3.1 Optimized GPU Kernel

To investigate the poor GPU performance we encountered, we tried fetching only part of input data generated by HadoopCL and then feed it to the GPU kernel code generated by Aparapi. We then profiled the automatically generated OpenCL kernel. The kernel can be split into four major steps: i) sorting, ii) nested-loop, iii) writing output and iv) validation, according to Algorithm 3. The execution time of each step is shown in Figure 5.5. Each time bar represents a step in the algorithm, and the four bars follow a chronological order from left to right. The execution time of each

part, as well as its contribution to the total execution time, is noted on the top of each bar.

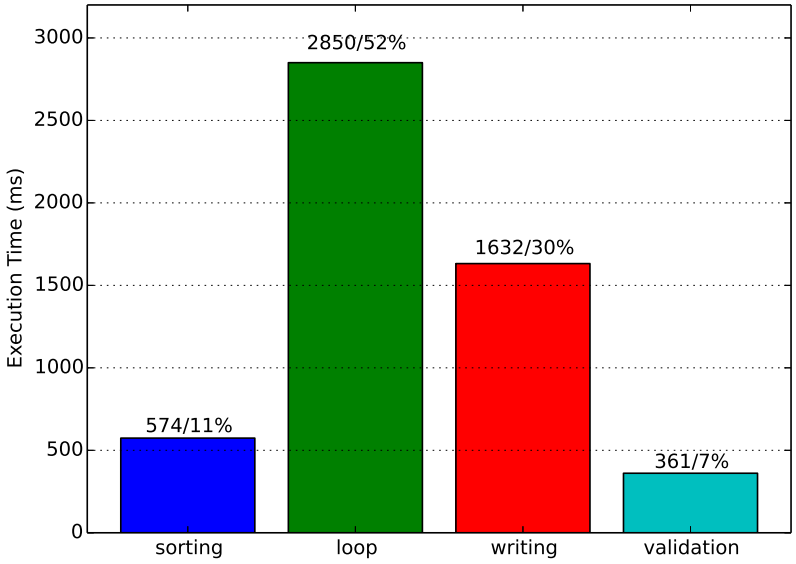

Figure 5.5: The execution time distribution in GPU mode.

Sorting

The first part of the HadoopCL map function is sorting. A key-value sort algorithm is applied to both the indexArray and the valueArray. Every work-item runs a bubble sort on a small chunk of data (approximately 30 elements) in parallel, which results in a large number of random accesses on GPU global memory. On the other hand, the CPU OpenCL kernel benefits from low access latency from using the cache for the same operation. To improve overall performance, we changed the HadoopCL framework by adding a filter function which is executed before the map() function. This changes original map->reduce scheme in HadoopCL to filter->map->reduce. The filter phase serves as a step where operations that do not fit on the GPU are executed. This allows HadoopCL to distribute a task to the computing device that

achieves the best performance.

Nested Loop

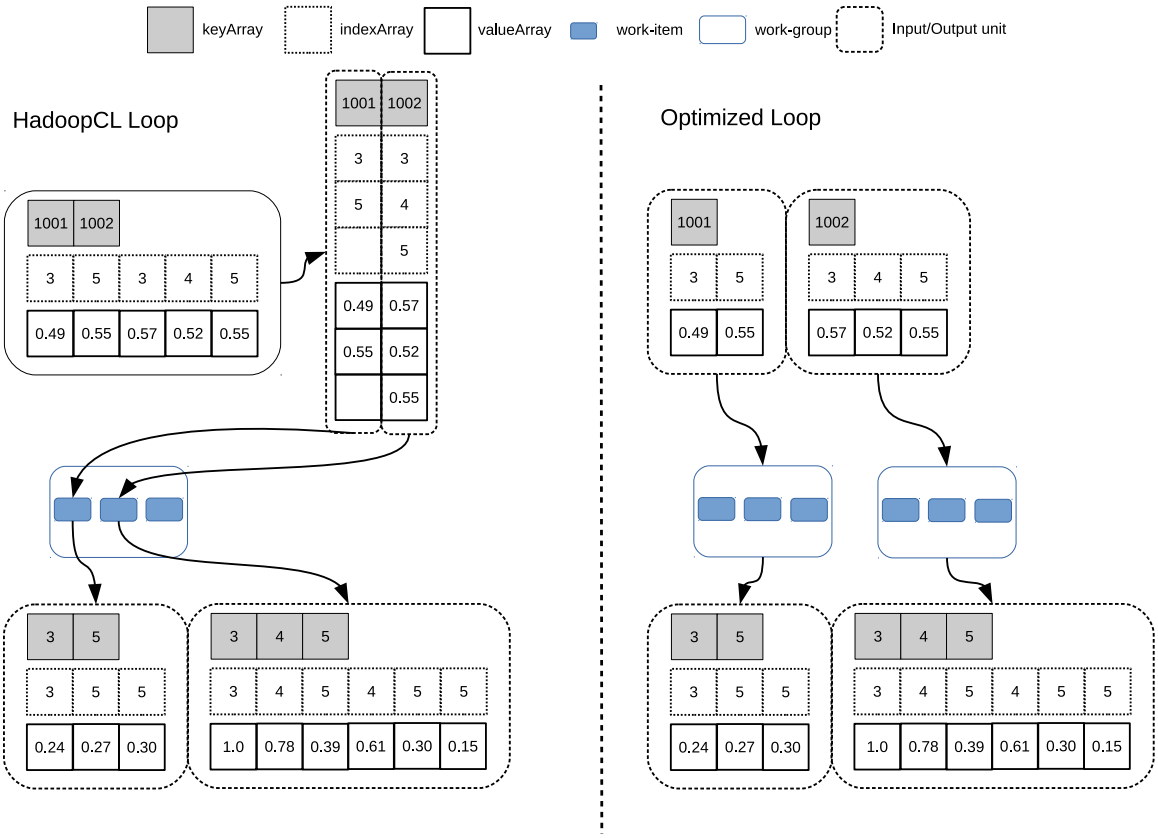

Figure 5.6: Loop Optimization

The major part of the GPU kernel is the two-level nested loop. The nested loop consumes 52% of the entire execution time. The left part of Figure 5.6 shows how HadoopCL executes the operations in the loop. A <key, value> record is fed to a work-item and the nested loop shown in Algorithm 3 is executed by every work-item in parallel. The input data in this example is the user vector of user 1001 and user 1002. The keys are represented by gray boxes, and indexArray containing itemIDs and

valueArray made up of preference values are represented by dotted and solid boxes, respectively. A dotted rounded rectangle box represents a chunk of input/output of the map() function. Since each work-item accesses a different number of indices and values in a contiguous memory space, there is a high possibility that multiple accesses can conflict for the same memory bank resulting in serialized accesses. This situation is called a *bank conflict*. To resolve bank conflicts, contiguous work-items should access contiguous memory locations - this pattern is called *coalesced memory accesses*. HadoopCL transpose the indexArray and valueArray to avoid bank conflicts. The tradeoff is a increased size of input, as every indexArray and valueArray that are not full are padded with zeros. In this example, the indexArray and valueArray of user 1001 is each padded with one element to match the length of user 1002's arrays.

One transposed chunk of data is then fed to one work-item, where n keys are generated for a user vector of length n. Each key i ($i \in [0, n-1]$) is associated with $n-i$ indices and values, thus the total number of indices generated by one work-item is $n*(n+1)/2$. However, this approach could lead to a load imbalance problem. Each work-item reads in n elements from the indexArray and valueArray, and then writes $n*(n+1)/2$ elements to the output indexArray and valueArray. Each work-item also writes n elements to the output keyArray and n elements to the output indexLookUpTable and valueLookUpTable.

Thus, for a user vector of length n, the total number of elements being generated by one work-item is $3n + (n*(n+1))$, which is quadratic with n. This could result in a highly imbalanced workload when working with different work-items in a work-group, when the n values for different users are highly different. Since n is the number of ratings a user provides, this situation is likely to be true. The gap between n active users and an inactive user could be very large. We denote $n_{max}/n - min$ as

the largest/smallest number of elements in a user vector. The work-item with least amount work will need to wait for the busiest work-item, waiting on the completion of an additional $\Theta({n_{max}}^2 - {n_{min}}^2)$ operations. For example, if user 1 rates 100 more items than user 0, work-item 1 is stalled for $\Theta(10000)$ cycles before being assigned any more work. The idle work-items can significantly impact performance.

We developed an improved kernel to solve this issue, which is shown in the right part of Figure 5.6. Instead of using a one-to-one mapping between a work-item and a user vector, the improved kernel maps a whole work-group to process a user vector. For a user vector of size n, n work items in a work-group are launched, where each work-item i ($i \in [0, n-1]$) takes the ith elements of the indexArray and valueArray and writes $n - i$ indices and values to the output. The output location of each work-item i is calculated by $pos = i * (2 * n - i + 1)/2$. Using the arithmetic progression formula, $S_n = \frac{n}{2}[2a_1 + (n-1)d]$. We also developed a further generalized algorithm to handle the case when n is bigger than the work-group size. The number of idle cycles in the improved kernel is significantly reduced from $\Theta({n_{max}}^2 - {n_{min}}^2)$ to $\Theta(n_{max} - n_{min})$. The improved kernel also saves transfer time between host and device memory since the input data is no longer transposed and padded, while still being accessed without introducing bank conflicts.

Writing Results

A second major portion (30%) of execution time of a rating system is spent on writing the results to the output buffer. This is partially due to the time spent on dynamic allocation of the output buffer. HadoopCL assumes a general Hadoop application that utilizes hash maps has a non-deterministic output size. Thus a dynamic allocation mechanism on the GPU memory is used. For every chunk of input data, HadoopCL

first allocates the maximum memory space needed to a temporary output buffer and calculates the actual size of the output buffer during the computation. It then allocates the final output buffer size using the actual counter before writing the results to the output. The dynamic allocation introduces significant overhead. HadoopCL maintains an integer counter and is atomically incremented every time a new buffer is allocated in order to detect if there is enough memory space left on GPU. As every work-item in a work-group needs to allocate an output buffer, the atomic operation invoked by allocation forces work-items to execute serially instead of in parallel. In the PairwiseSimliarty job, however, the output size can be determined before launching the kernel. As discussed in the last section, a user vector containing n non-zero elements corresponds to $3n + (n * (n + 1))$. Since the output size is deterministic, we can pre-allocate GPU device buffers before launching a kernel to avoid using expensive atomic operations on the GPU.

Validation

To recover from the situation when a work-item fails to allocate a new buffer because of lack of GPU memory, HadoopCL introduces a validation step. Hadoop uses a boolean value to notify the host if a work-item failed to be delivered and should be resent. This also incurs a 7% overhead, as shown in Figure 5.6. The improved kernel eliminates the overhead totally since we preallocate GPU memory and we will not encounter overruns.

5.3.2 Optimized GPU Kernel Integration

In this section, we describe how we integrate the optimized GPU kernel with HadoopCL. There are two main changes needed in HadoopCL to support this integration: i) the

use of a kernel file and ii) the addition of bookkeeping variables.

Kernel File

A kernel file is a plain text file that contains the OpenCL program. By default, a user can inform HadoopCL that a device should specify to use HadoopCL to execute the OpenCL program generated by Aparapi on that device. To execute our optimized kernel, we first manually write the kernel program using the same signature as the Aparapi-generated program. Then we added functionality to HadoopCL so that it can use a manually-tuned kernel file instead of the Aparapi program generated at run-time.

Book-keeping Variables

Besides the default arguments passed to the kernel (inputKey and inputValue buffers), HadoopCL adds several other variables to bookkeep run-time information in order to recover from possible problems such as shortage of GPU memory. This information will be passed back to the host to effect a relaunch of the associated kernel that failed. In the optimized kernel however, there will be no memory shortage since it is output-size aware and GPU memory is allocated on the host side. We still maintain the bookkeeping variables and assign them values so that the host side is informed that no failures occurred.

5.3.3 Performance Evaluation of Optimized GPU kernel

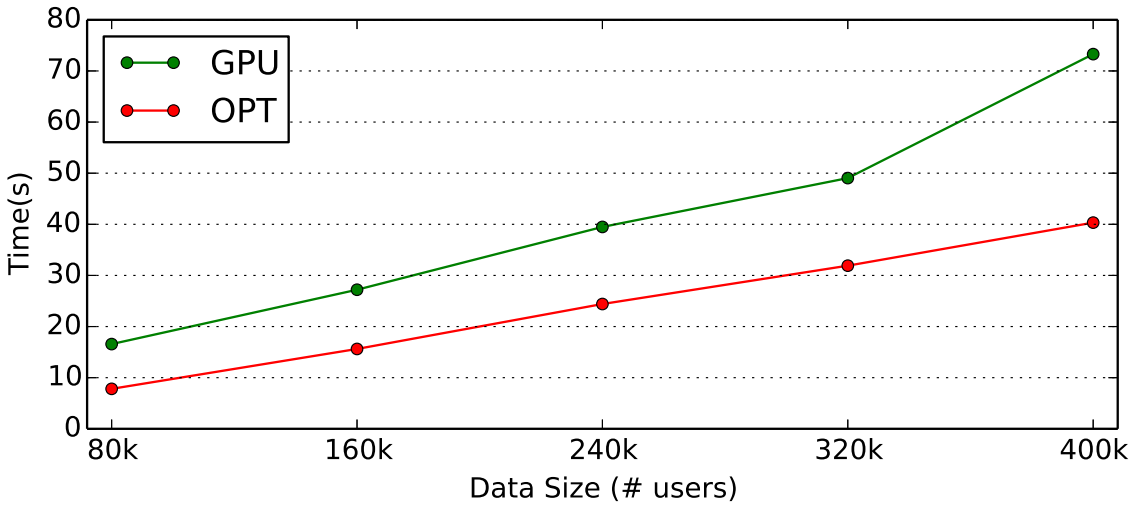

Figure 5.7: Execution time of an optimized vs. an auto-generated GPU kernel.

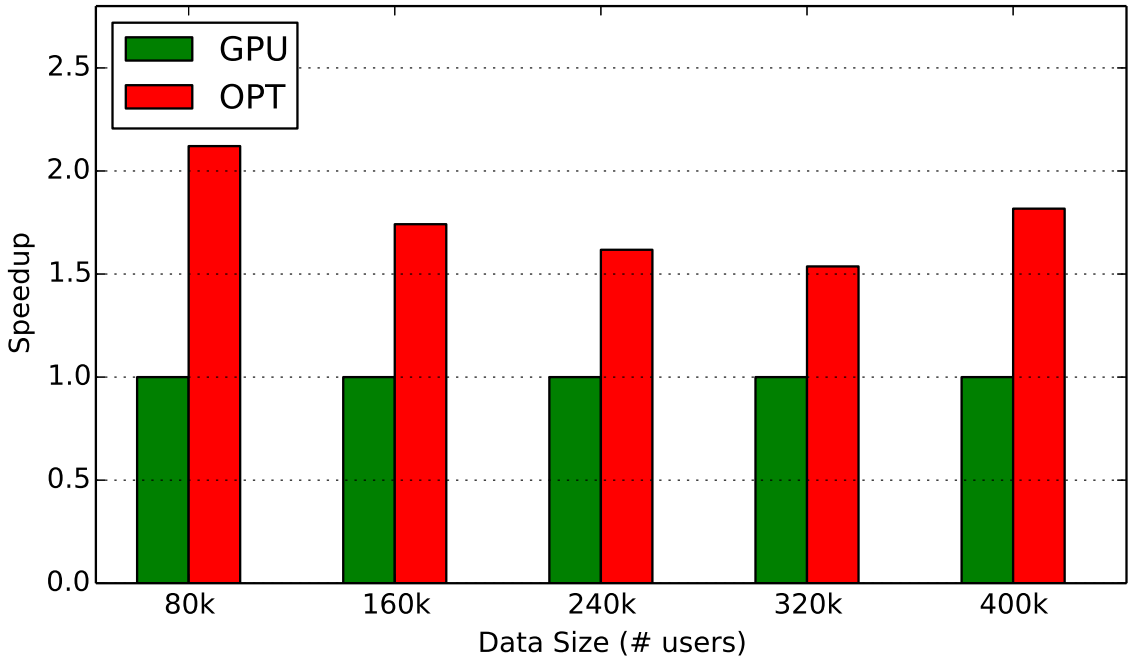

Figure 5.8: Speedup of an optimized vs. an auto-generated GPU kernel.

To evaluate the effect of optimizations of the GPU kernel described in previous section. We select a subset of the input data of the pairwiseSimilarity job and measure the execution time of the mapper running on the GPU using automatically generated and manually tuned OpenCL programs. As shown in Figure 5.7, the input data used is part of the wikipedia dataset. The number of users is increased from 80k to 400k. The execution time of both GPU kernel scales well as we increase the size of the dataset. The optimized GPU kernel achieves around a 1.7X to 2.1X speedup, as shown in Figure5.8. The results highlight the benefits of eliminating expensive operations such as atomic operations and random memory accessing. Another observation made during this evaluation is that once the data size is increased to 10M users, the GPU kernel will hang, since a large number of work-items are trying to access one integer atomically. In the performance evaluation presented in next chapter, we use real world data set sizes. We only present the performance results comparing Java, OpenCL on CPU and optimized OpenCL on GPU.

Chapter 6

Performance Evaluation

In this chapter, a performance evaluation of the Hadoop pairwiseSimilarity job on different platforms is presented. In Section 6.1, we report on the overall performance improvement of the pairwiseSimilarity job as run on our APU cluster. In Section 6.2 we move the job to a cluster with an Nvidia K20m discrete GPU. The Nvidia GPU has a much larger number of GPU cores.

6.1 Performance Evaluation on APU cluster

The hardware configuration in this section is shown int Table 5.1. The NameNode contains an AMD A8-3850. Since the NameNode mainly manages tasks such as data partitioning and task scheduling instead of computational intensive job, the affect on performance from the NameNode is not addressed in this work. The DataNode, on the other hand, dominates the execution performance of HadoopCL. The DataNode in this experiment is run on an AMD A10-6700 APU. The GPU on this APU is an AMD Radeon HD 8670D that uses the Northern Islands Architecture. This APU has 6 compute units and 32K local memory on each compute unit. The CPU part is a

quadcore x86 processor running at 2.9GHz.

To test the scalability of the framework, we experimented using one and two DataNodes.

6.1.1 Execution time on one DataNode.

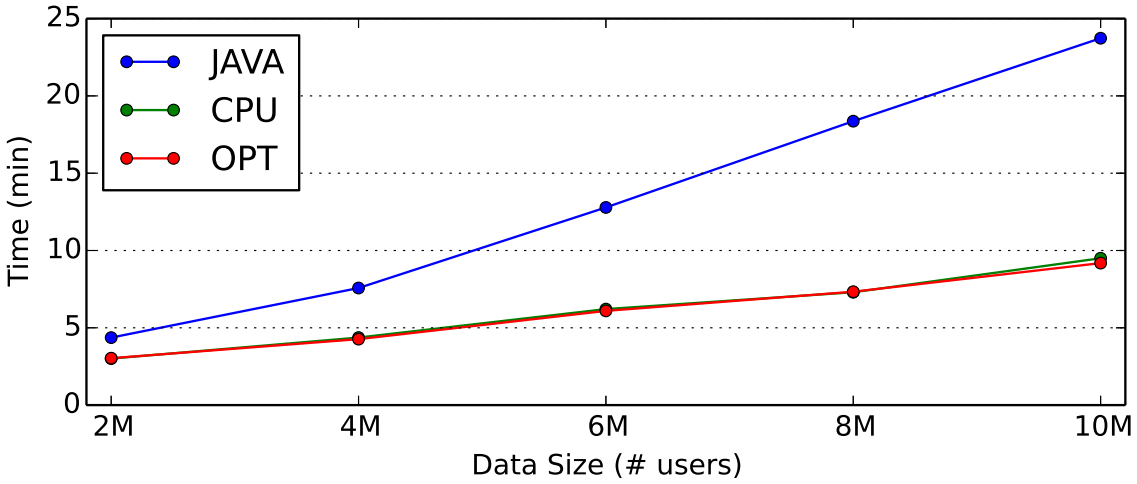

Figure 6.1: Execution time of JAVA, CPU and OPT.

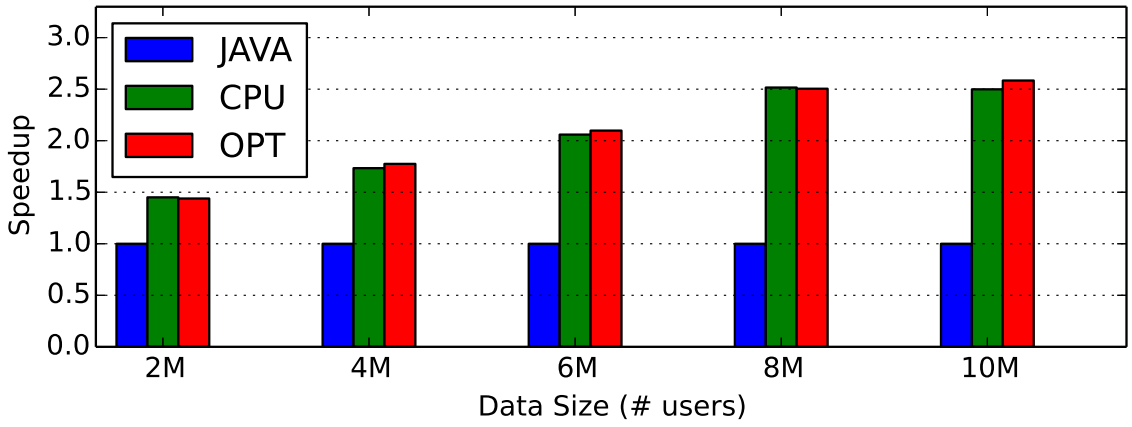

Figure 6.2: Speedup of JAVA, CPU and OPT.

Figure 6.1 shows the execution time of the pairwiseSimilarity job run while increasing input data size. The data size is increased from 2 million to 10 million users. JAVA, CPU and OPT labels indicate our three different execition modes: original Mahout jobs running on Hadoop using Java, HadoopCL generated OpenCL programs running on a CPU and optimized OpenCL programs running on a GPU. All modes demonstrate good scalability with increased input data size. Figure 6.2 presents the speedup of the CPU and OPT modes over a JAVA mode baseline. The speedup range from 1.5x to 2.6x for both CPU and OPT. And the performance of CPU and OPT is very close. This is because the optimization on the GPU kernel eliminates many random memory accesses and atomic operations, which are very expensive operations on the GPU. A multi-core CPU has a relatively large cache, as well as a sophisticated latency-hiding memory interface. Also, OPT is using the GPU portion of an APU, which has similar computational resources to the multi-core CPU.

6.1.2 Execution on Two DataNodes

Figure 6.3 presents the execution time on 2 APU nodes. We find the same trend as seen in Figure 6.1 for a single node. In terms of scalability, when doubling the number, we achieve speeds ranging from 1.4X to 2.8X as shown in Figure 6.4. The OPT mode outperforms the CPU mode, but not significantly.

6.2 Performance Evaluation on Nvidia cluster

To further investigate the performance of our optimized GPU kernel, we experimented with a more powerful discrete GPU cluster which hosts an Nvidia k20m. The hardware configuration is shown in Table 6.1. Our experiments are carried out on two

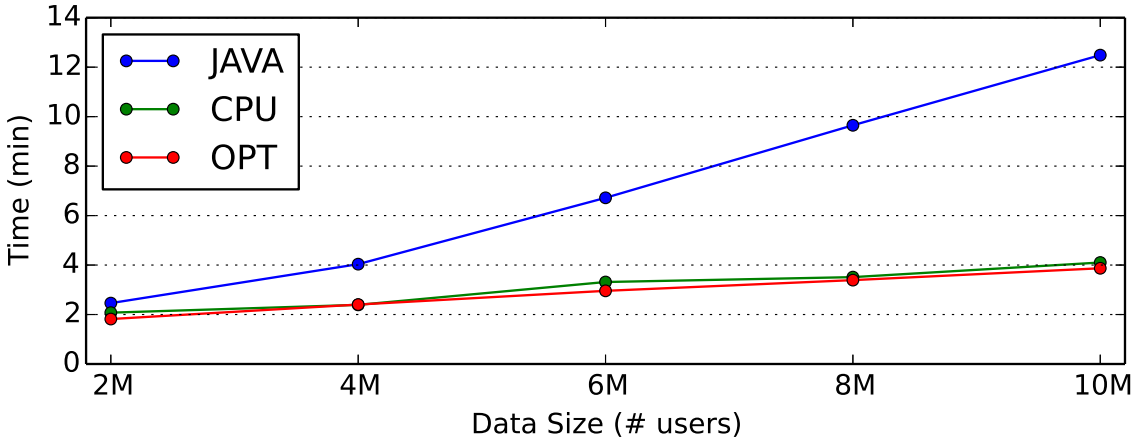

Figure 6.3: Execution time of JAVA, CPU and OPT.

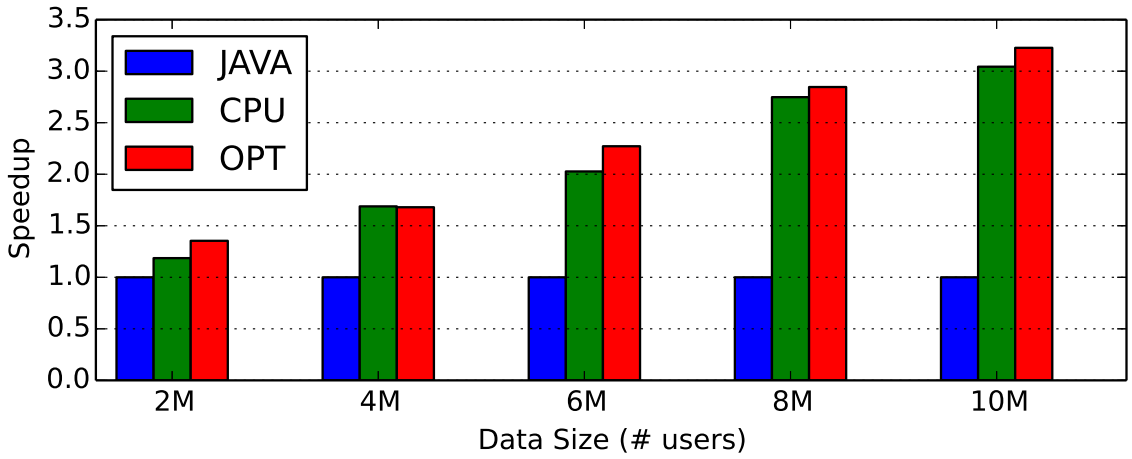

Figure 6.4: Speedup of JAVA, CPU and OPT.

compute nodes, with one acting as the NameNode and the other as a DataNode. We only present execution time of JAVA and OPT in Figure 6.5 since the Nvidia k20m cluster does not support OpenCL running on CPU. Figure 6.6 shows that the discrete GPU yields a higher speedup (1.5X to 4.4X) as compared to the APU cluster.

	NameNode & DataNode
CPU	Intel Xeon E5-2650
GPU	NVIDIA Tesla K20m
# CPU Cores	8
# GPU Cores	13
CPU Clock Rate	2GHz
GPU Clock Rate	706MHz

Table 6.1: Nvidia K20m-based platform specification.

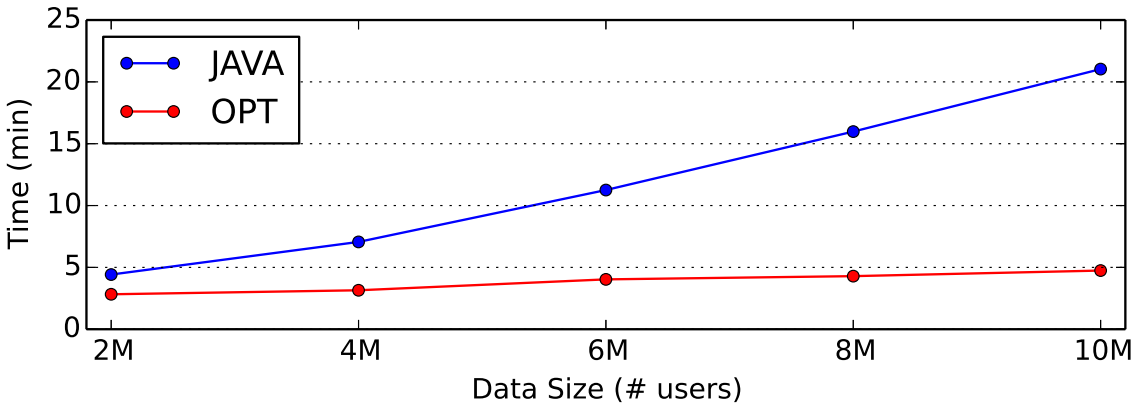

Figure 6.5: Execution time of JAVA and OPT.

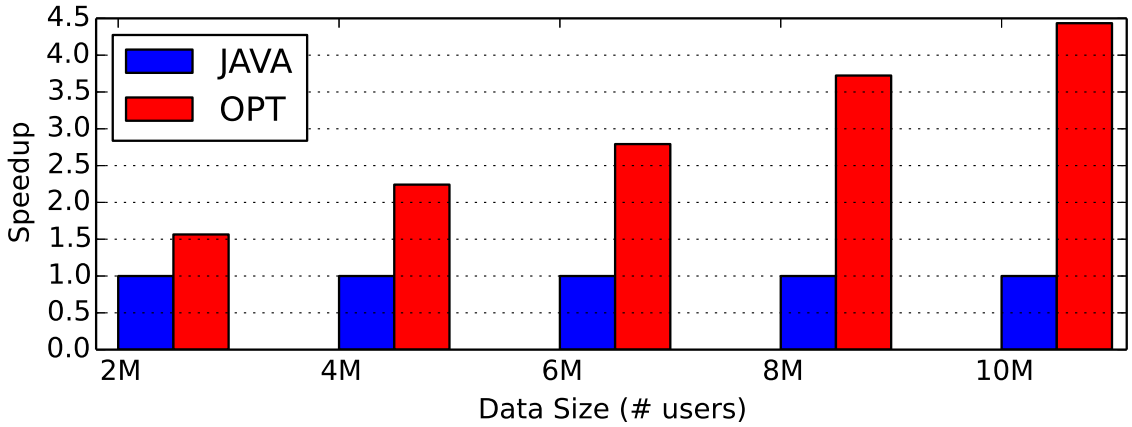

Figure 6.6: Speedup of JAVA and OPT.

Chapter 7

Discussion, Conclusion and Future Work

7.1 Conclusion

In this thesis we began by presenting a detailed description of the Mahout recommendation system. We have profiled Mahout performance while running on multiple nodes in a cluster. We have also presented a performance evaluation of a Mahout job running on heterogeneous platforms using CPUs, AMD APUs and NVIDIA discrete GPUs with HadoopCL. We focused our efforts on a time-consuming job in Mahout and manually tuned this job to run on a GPU. We also modified the HadoopCL pipeline from map->reduce to filter->map->reduce to increase the flexibility of HadoopCL during task assignment.

We consider how to optimize the performance of automatically generated OpenCL GPU We were able to achieve speedups ranging from 1.5X to 2X using optimized GPU kernel integrated into HadoopCL, run on an APU cluster. We were able to get further

speedup utilizing an Nvidia discrete GPU, generating speedups of 2X to 4X.

7.1.1 Contributions of this Thesis

The contributions of this work are:

- We evaluate Mahout running real world applications on multiple nodes, identify the performance and application characteristics.
- We execute a time-consuming job in Mahout on heterogeneous platforms including CPU-only, APU, and discrete GPUs using HadoopCL in a cluster environment. We evaluate the performance and identify performance issues present in automatically-generated HadoopCL GPU programs.
- We optimize the HadoopCL execution by inserting a manually-tuned GPU kernel which reduces the workloads imbalacing problem in the original kernel. We also add a new filter phase to HadoopCL to improve resource utilization.
- We evaluate the performance of optimzied execution in HadoopCL on heterogeneous platforms.

7.1.2 Future Work

There are additional Mahout jobs which could be potentially mapped to heterogeneous platforms. We would like to extend our work to consider those jobs, to build further on the results of this work, and to seek further improvements for the HadoopCL framework. The pairwiseSimilarity job is a typcial application where sparse data causes workload imbalancing problems on the GPU. We would like to study more applications in this catogory and implement other alogrithms to handle sparse data.

We would also like to provide a general interface that provides experienced GPU programmers a easier way to integrate their fun-tuned GPU kernel into HadoopCL.

Bibliography

[1] Amd website. www.amd.com/.

[2] Apache hadoop website. http://hadoop.apache.org/.

[3] Apache mahout website. http://mahout.apache.org/.

[4] Khronos opencl. http://www.khronos.org/opencl/.

[5] Nvidia website. www.nvidia.com/.

[6] Opengl website. http://www.opengl.org/.

[7] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Building an efficient hash table on the GPU. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 4, pages 39–53. Morgan Kaufmann, October 2011.

[8] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Ktbiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[9] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

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

[11] Linchuan Chen, Xin Huo, and G. Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, 2012.

[12] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 25:1–25:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[13] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[15] The MPI Forum. Mpi: A message passing interface, 1993.

[16] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous computing with OpenCL, 2nd Edition.* Morgan Kaufmann, 2011.

[17] G.Frost. Aparapi website. `https://code.google.com/p/aparapi/`.

[18] Dan Gillick, Arlo Faria, and John Denero. Mapreduce: Distributed computing for machine learning, 2006.

[19] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1918–1927, Washington, DC, USA, 2013. IEEE Computer Society.

[20] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[21] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[22] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '99, pages 230–237, New York, NY, USA, 1999. ACM.

[23] Joseph A. Konstan, Bradley N. Miller, David Maltz, Jonathan L. Herlocker, Lee R. Gordon, and John Riedl. Grouplens: Applying collaborative filtering to usenet news. *Commun. ACM*, 40(3):77–87, March 1997.

[24] Y. Lin, S. Okur, and C. Radoi. Hadoop+aparapi: Making heterogenous mapreduce programming easier. 2012.

[25] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, Jan 2003.

[26] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[27] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning Publications Co., Manning Publications Co. 20 Baldwin Road PO Box 261 Shelter Island, NY 11964, first edition, 2011.

[28] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 285–295, New York, NY, USA, 2001. ACM.

[29] Jeff A. Stuart and John D. Owens. Multi-gpu mapreduce on gpu clusters. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1068–1079, Washington, DC, USA, 2011. IEEE Computer Society.