2011

Analysis and acceleration of data mining algorithms on high performance reconfigurable computing platforms

Song Sun
Iowa State University

Analysis and acceleration of data mining algorithms on high performance reconfigurable computing platforms

by

Song Sun

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Joseph Zambreno, Major Professor

Arun K. Somani

Akhilesh Tyagi

Morris Chang

Srikanta Tirthapura

Iowa State University

Ames, Iowa

2011

DEDICATION

In Memory of My Maternal Grandma

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, I want to thank my advisor Dr. Joseph Zambreno for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I always feel warm when recalling those days we worked together and he taught me everything on those projects. The research style and methodology I learned from him will benefit me in the rest of my life.

I am also great thankful to Dr. Akhilesh Tyagi, Dr. Phillip Jones and Vicky Thorland-Oster who gave me a lot of help in my Ph.D work. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Arun Somani, Dr. Morris Chang and Dr. Srikanta Tirthapura. I would additionally like to thank Madhu Monga and Lakshmi Kiran Tondehal for their contribution to the projects.

I also appreciate my beloved parents, younger sister, my nephew, my mother-in-law, especially my wife and my lovely daughter, and all other members of my family and friends who gave me so much help for the years in Ames.

ABSTRACT

With the continued development of computation and communication technologies, we are overwhelmed with electronic data. Ubiquitous data in governments, commercial enterprises, universities and various organizations records our decisions, transactions and thoughts. The data collection rate is undergoing tremendous increase. And there is no end in sight. On one hand, as the volume of data explodes, the gap between the human being's understanding of the data and the knowledge hidden in the data will be enlarged. The algorithms and techniques, collectively known as data mining, are emerged to bridge the gap. The data mining algorithms are usually data-compute intensive. On the other hand, the overall computing system performance is not increasing at an equal rate. Consequently, there is strong requirement to design special computing systems to accelerate data mining applications.

FPGAs based High Performance Reconfigurable Computing(HPRC) system is to design optimized hardware architecture for a given problem. The increased gate count, arithmetic capability, and other features of modern FPGAs now allow researcher to implement highly complicated reconfigurable computational architecture. In contrast with ASICs, FPGAs have the advantages of low power, low non-recurring engineering costs, high design flexibility and the ability to update functionality after shipping. In this thesis, we first design the architectures for data intensive and data-compute intensive applications respectively. Then we present a general HPRC framework for data mining applications:

Frequent Pattern Mining(FPM) is a data-compute intensive application which is to find commonly occurring itemsets in databases. We use systolic tree architecture in FPGA hardware to mimic the internal memory layout of FP-growth algorithm while achieving higher throughput. The experimental results demonstrate that the proposed hardware architecture is faster than the software approach.

Sparse Matrix-Vector Multiplication(SMVM) is a data-intensive application which is an important computing core in many applications. We present a scalable and efficient FPGA-based SMVM architecture which can handle arbitrary matrix sizes without preprocessing or zero padding and

can be dynamically expanded based on the available I/O bandwidth. The experimental results using a commercial FPGA-based acceleration system demonstrate that our reconfigurable SMVM engine is more efficient than existing state-of-the-art, with speedups over a highly optimized software implementation of 2.5X to 6.5X, depending on the sparsity of the input benchmark.

Accelerating Text Classification Using SMVM is performed in Convey HC-1 HPRC platform. The SMVM engines are deployed into multiple FPGA chips. Text documents are represented as large sparse matrices using Vector Space Model(VSM). The k-nearest neighbor algorithm uses SMVM to perform classification simultaneously on multiple FPGAs. Our experiment shows that the classification in Convey HC-1 is several times faster compared with the traditional computing architecture.

MapReduce Reconfigurable Framework for Data Mining Applications is a pipelined and high performance framework for FPGA design based on the MapReduce model. Our goal is to lessen the FPGA programmer burden while minimizing performance degradation. The designer only need focus on the mapper and reducer modules design. We redesigned the SMVM architecture using the MapReduce Framework. The manual VHDL code is only 15 percent of that used in the customized architecture.

CHAPTER 1. INTRODUCTION

1.1 The Problem

1.1.1 Intensive Data

The use of digital technologies has fueled rapid data growth over the past decade. Business is collecting vast amounts of data to make forecasts and intelligent decisions. Millions of Internet users are making available data in the form of photograph, movie and music libraries. Billions of on line transactions in worldwide web-based stores and Internet banking are committed everyday. In scientific research scenarios, remote sensors on a satellite and gene sequence generated by microarrays may produce Tera bytes of data in a short time. It has been estimated that the amount of data stored in the world's databases doubles every 20 months [32]. The largest hybrid databases are now on the orders of Peta bytes [143]. Here are some examples cited from [89, 65]:

- The North American electric power grid operations generate 15 Tera Bytes data per year.
- Social networking sites such as Facebook capture and store Peta Bytes of heterogeneous information.
- Google sorts through 20 Peta Bytes everyday.
- The Large Hadron Collider(LHC) at the Center for European Nuclear Research(CERN) generate raw data at a rate of 2 Peta Bytes per second starting from 2008.
- The Large Synoptic Survey Telescope(LSST; www.lsst.org) will generate several Peta Bytes of new image and catalog data every year. The Square Kilometer Array (SKA; www.skatelescope.org) will generate about 200 Giga Bytes of raw data per second that will require Peta Flops (or possibly Exa Flops) of processing to produce detailed radio maps of the sky.

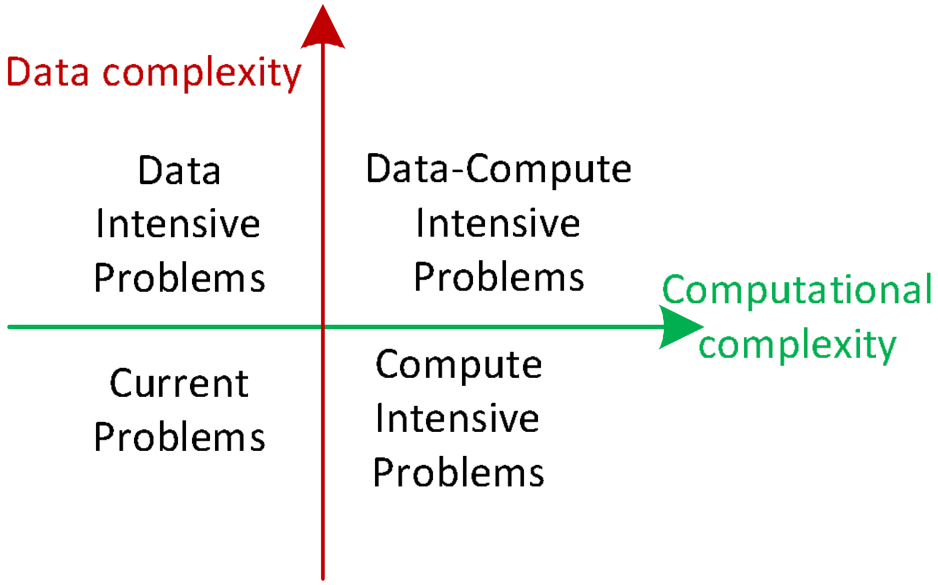


Figure 1.1 Classification of Intensive Computing [65]

- International Data Corporation(IDC) estimates that there are 281 Exa Bytes digital data in 2007. The amount of information will grow by a factor of 10 in five years.

1.1.2 Intensive Computing

Figure 1.1 classifies the computing into four categories. Data-intensive tasks usually involves with Terabytes to Petabytes or even larger size of datasets which may come with different formats from different places. Processing requirements of this kind of data typically scale **near-linearly** with data size. The data correlation is little and parallelization is easy to be achieved. Data-compute intensive tasks need to deal with both large data sets and complicated computation. Processing requirements typically scale **super-linearly** with data size. Specialized hardware accelerators can be used to improve the computation performance.

People have proposed a lot of definitions for data-intensive computing(DLC). One straightforward definition is that DIC is a discipline focusing on massive data processing. However, the application requirements of DIC are usually different from traditional ones, including the way of gathering and processing data, hardware and software algorithms. For example, as stated in [64], ”many data-intensive applications are data-path-oriented, making little use of branch prediction and speculation hardware

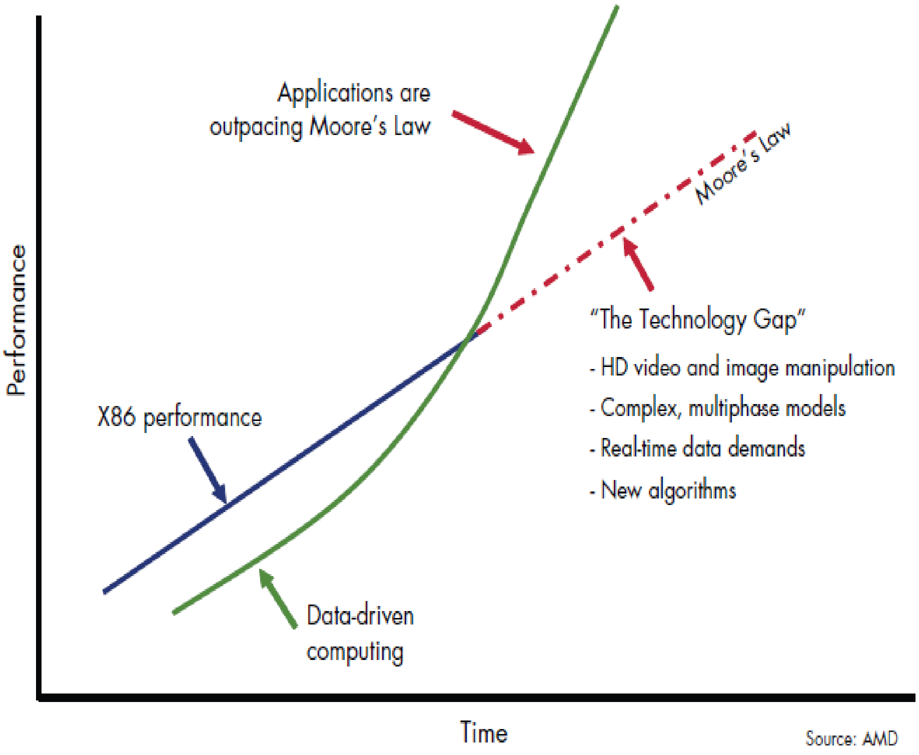


Figure 1.2 Industry landscape: The Technology Gap [13]

in the CPU. These applications are well suited to steaming data access and can't effectively use the sophisticated on-chip cache hierarchy. Their ability to process large data sets is hampered by orders-of-magnitude mismatches between disk, memory, and CPU bandwidths." We use the definitions of compute-intensive and data-intensive computing from [89]. The compute-intensive computation is to process those tasks where the processing power is the rate-limiting factor; the data-intensive computation is qualitatively defined to process those tasks where the data availability is the rate-limiting factor. The ”availability” used here includes such factors as latency and bandwidth in hardware systems that impact the capacity to obtain, process and dispose of data at rates that match the source's capability to provide data. In this thesis, we focus on the upper two quadrants in Figure 1.1 using reconfigurable computing system.

1.1.3 Limitations of Traditional Processor Architecture

For most of the microprocessor's history, application demands have risen in response to processor improvement [13]. However, application demands have outpaced the conventional processor's ability

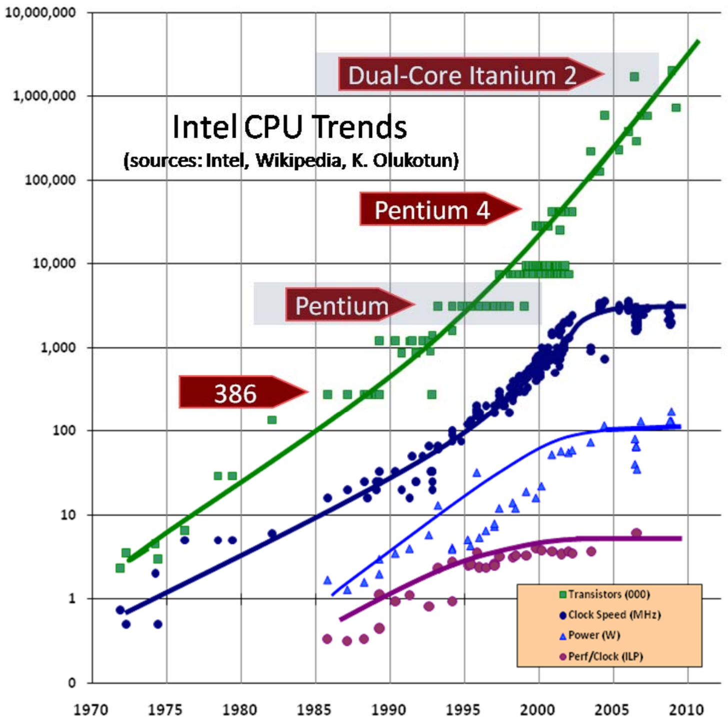


Figure 1.3 Traditional Performance Improvement Flattened [131]

to deliver [13] recently. High performance computing applications are now demanding more than processors alone can handle, creating a technology gap between demand and performance as shown in Figure 1.2.

These enormously complex and heterogeneous data requires not only high computation speed but also a systematically high performance infrastructure. Traditional general-purpose single-core CPUs take the advantage of VLSI technology over the past few decades. The size of transistors is reduced. The frequency closely follows Moore's law [1]. The performance of those applications running in PC is improved without modifying or redesigning the software. However, more recently, the gain in the single-core performance of general-purpose processors has diminished because VLSI system performance hit the memory wall, power wall [69], instruction-level parallelism(ILP) wall. The **memory wall** refers to the increasing gap between processor and memory speeds. This results in an increase in cache sizes required to hide memory access latencies [117]. Eventually the memory bandwidth becomes the bottleneck in performance. The **power wall** refers to power supply limitations and thermal

[1] The Moore's law states that we can integrate twice as many components onto an integrated circuit every 18 months at fixed cost.

dissipation limitations. For the silicon lithography below 90nm, the static power from leakage current surpass dynamic power from circuit switching. Power density has become the dominant constraint in chip design, and limits the clock frequency growth [131]. The performance, cost, and reliability of modern computer systems and data centers are dictated by the management of their limited energy and thermal budgets [29]. The **ILP wall** refers to the rising difficulty in finding enough parallelism in the existing instructions stream of a single process. Increasing cache size or introducing more ILP yields too little performance gain compared to the development cost [25]. Together, these three walls reduce the performance gains expected for single-core general-purpose processors. The VLSI system performance has not shown much gain from continued processor frequency increases since 2003 as was once the case shown in Figure 1.3. Further, newer manufacturing and device constraints are faced with decreasing feature sizes, making future performance increases harder to obtain. Intel summarized these issues in their Platform 2015 documentation:

> *First of all, as chip geometries shrink and clock frequencies rise, the transistor leakage current increases, leading to excess power consumption and heat... Secondly, the advantages of higher clock speeds are in part negated by memory latency, since memory access times have not been able to keep pace with increasing clock frequencies. Third, for certain applications, traditional serial architectures are becoming less efficient as processors get faster (due to the so-called Von Neumann bottleneck), further undercutting any gains that frequency increases might otherwise buy. In addition, partly due to limitations in the means of producing inductance within solid state devices, resistance-capacitance(RC) delays in signal transmission are growing as feature sizes shrink, imposing an additional bottleneck that frequency increases don't address.*

1.2 Architecture Solutions

1.2.1 Traditional Computing Architecture

In order to break down the power and ILP wall while increasing computing performance, the microprocessor industry rapidly shifted to multi-core processors after exhausted all well-understood avenues to extract more performance from a uniprocessor. The dual-core designs running at 85% of the fre-

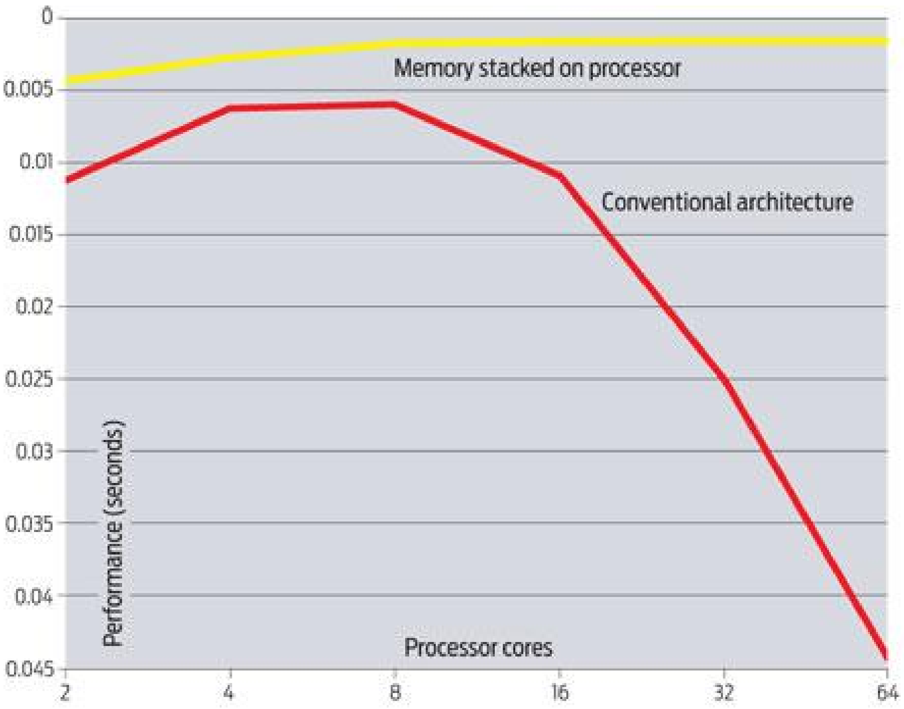


Figure 1.4 Trouble Ahead:More cores per chip will slow some programs[red] unless there is a big boost in memory bandwidth[yellow].Source:SANDIA [84]

quency with 85% of the supply voltage offer 180% better performance than single-core designs, yet consume approximately the same power [25]. It might suffice for two, four and perhaps eight cores. The user can run their programs on different cores in parallel. As a result, the traditional single-threaded applications no longer see significant gains in performance with each processor generation, unless these applications are re-architectured to take advantage of the multi-core processors. However, todays multi-core CPUs spend most of their transistors on logic and cache [25], with a lot of power spent on non-computational units. For informatics, more cores does not mean better performance, according to the simulations at Sandia National Laboratories in New Mexico [84]. Because of limited memory bandwidth and poor memory-management schemes, the performance would level off or even decline with more cores as shown in Figure 1.4. The heart of the trouble is the memory wall. Another cause is that there is no physical relationship between what a processor may be working on and where the next set of data it needs may reside. As concluded in [14], "sneaking up on the problem of parallelism via multicore solutions was likely to fail". Many-core approach may achieve sustained computational

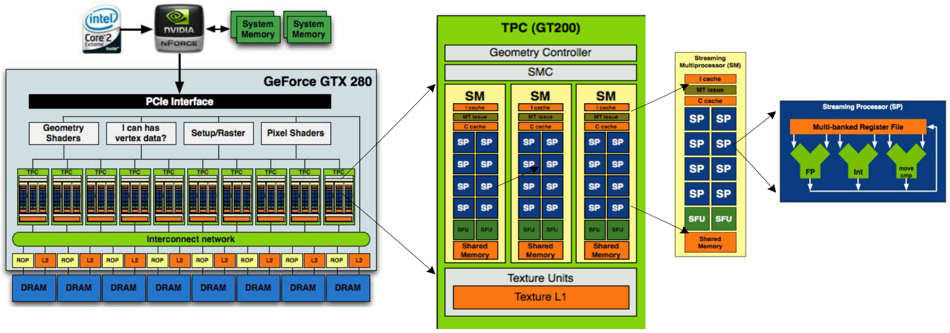


Figure 1.5 NVIDIA GT200 GPU Architecture [133]

performance. However, fully unleashing the potential many-core architecture will require fundamental advances in computer architecture and programming model [131].

In order to alleviate the side effect of the memory wall, cache memories are designed to reduce the memory latency. The cache usually has several hierarchies. The smallest and fastest cache is integrated with CPU into the same chip. The off-chip cache is larger and has higher latency. Another alleviation is Non-Uniform Memory Access(NUMA) on shared-memory machines [25] where memory is physically distributed between cores. However, caches are only efficient for predictable memory access. For those programs without temporal and space locality data intensive computing, using cache will degrade performance, as the cache transfers full cache lines across the memory bus when only a single element is needed without reuse [25].

Cluster is another platform based on traditional CPUs. However, the bandwidth required to synchronize multiple machines solving a single problem is the bottleneck [21]. The more computers working on it the less effective each one is. Only 10% of the power is used to solve the problem. Hence, clusters are not space and power efficient. The industry's response of doubling cores has reinvigorated study of more radical alternative approaches, such as FPGAs, GPU [131].

1.2.2 Graphics Processing Unit Architecture

Graphical Processing Units(GPUs) was original designed to accelerate 3D graphics rendering. Nowadays, more than 90% of new desktop and notebook computers have integrated GPUs, which are usually far less powerful than those on a dedicated video card [6]. GPU's highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. GPUs are designed to operate in a single instruction multiple data(SIMD) fashion. GPU are being actively explored for general purpose computations(GPGPUs) in recent times. There is a rapid increase in the number and diversity of scientific communities exploring the computational power of GPUs for their data-intensive algorithms. One example of GPU architecture is shown in Figure 1.5 [133]. The Tesla architecture is based on a scalable processor array which has 10 independent texture/processor clusters(TPCs). Each TPC has 24 streaming processor(SP) cores. The total $24 \times 10 = 240$ cores are organized into 30 streaming multiprocessors(SMs) [97]. In each SM there are 8 SPs, a multi-thread instruction fetch and issue unit(MT issue), an instruction cache(I Cache), a read-only constant cache(C Cache), and a 16KB shared memory. Each special function unit(SFU) contains 4 floating-point multipliers. Telsa SM is single instruction, multiple hardware-thread(SIMT) with zero scheduling overhead. Each SM thread has its own thread execution state and independent code path. SIMT architecture is similar to SIMD except that one instruction is applied to multiple independent threads in parallel, not just data lanes in SIMT. The data flows from top to bottom starting with the PCIe interface. The memory system consists of DRAM control and fixed-function raster operation processors(ROPs). More details of Nvidia Tesla GPU architecture can be found in [97].

1.2.3 FPGA Architecture

A Field-Programmable Gate Array(FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing-hence ”field-programmable” [6]. A hardware description language(HDL) such as Verilog and VHDL are usually used to design circuit on FPGAs. FPGA architecture consists of Logic Blocks, I/O pads, memory blocks and Switch Blocks(SB). Figure 1.6 shows a generic architecture of FPGAs. Logic blocks usually consists of LookUp Table(LUT), flip-flops, Multiplexers and other arithmetic components. An n-bit LUT can encode any n-input boolean function. The switch blocks are composed of wire segments and programmable switches. The connections of logic blocks are programmed by those switch blocks.

FPGAs, alternative to the custom ICs, can be used to implement an entire System On one Chip(SOC). Custom ICs are expensive and takes long time to design. In contrast, FPGA are easy to implement with a short time with the help of Computer Aided Design(CAD) tools. However, the circuit delay is usually much larger than that of custom ICs. Thus, the clock frequency of FPGA design is usually in the orders

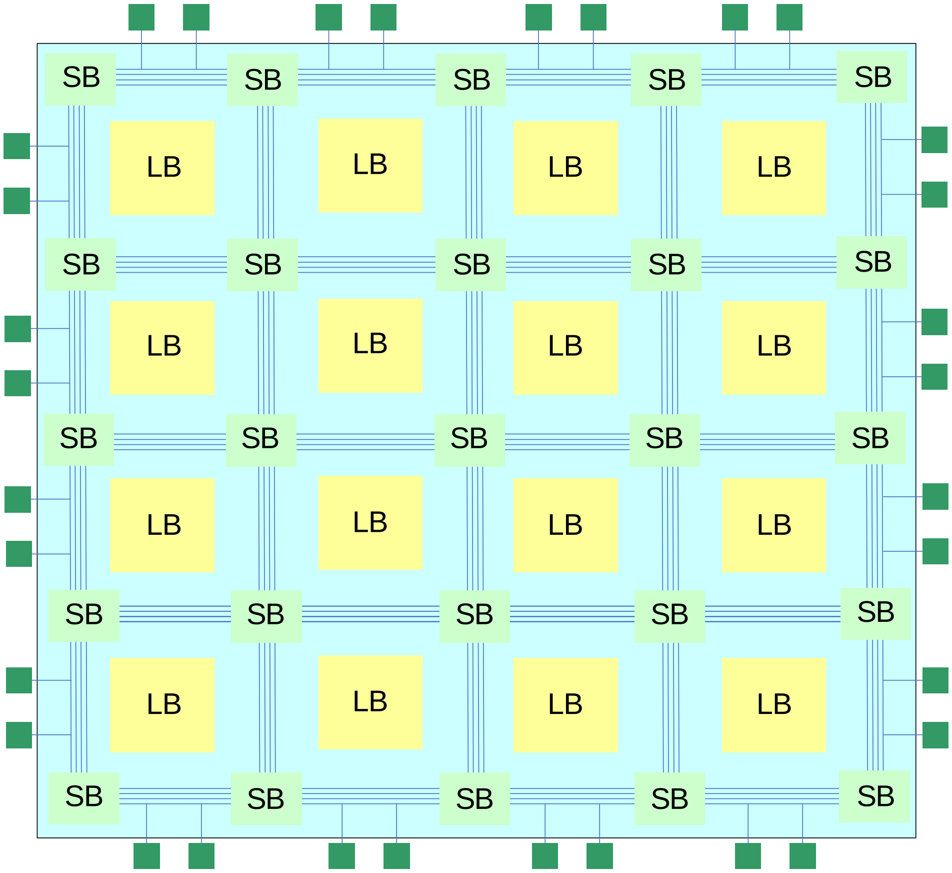


Figure 1.6 Generic FPGA Architecture

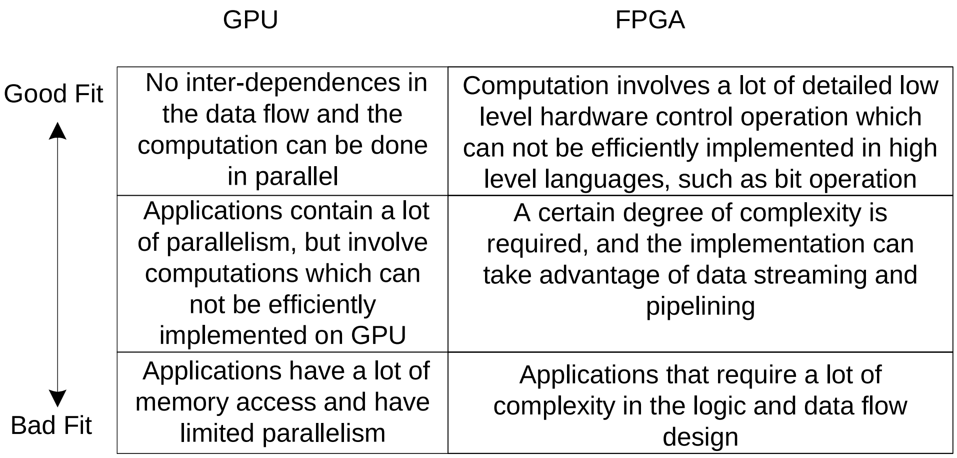
of hundreds of MHz. However, the power of FPGA lies in processing data in a parallel and pipelined way. Unlike traditional general processor, the computing units in FPGA can be arranged to handle data simultaneously. The data is pipelined into the units in each clock cycles. These two features highly increase the data throughput of FPGA-based system. The appendix has more details of FPGAs.

1.2.4 Architecture Properties Comparison

A comparison of CPU, GPU and FPGAs architecture properties is shown in Table 1.1 [25]. Considering the performance per dollar, the GPU outperforms CPU and FPGAs. Higher performance now equals higher profits [13]. FPGAs have the lowest frequency and the best performance per watt. With regard to the performance issue, applications typically exhibit vastly different performance characteristics depending on the platform. Figure 1.7 [33] summarizes the application fitness categorization. The GPU has the high performance only if the same operation is independently executed on a large set of

Table 1.1 Computing Architecture Comparison [25]. The data is reported per physical chip. The CPU is an Intel Core i7-965 Quad Extreme. The NVIDIA GPU is the Tesla C1060. The FPGA is the Virtex-6 SX475T.

	CPU	**GPU**	**FPGA**
Frequency(GHz)	3.2	1.3	<0.55
Single Precision GFLOPS	102.4	936	550
Double Precision GFLOPS	51.2	78	137
Single Precision GFLOPS/Watt	0.8	5	13.7
Single Precision GFLOPS/USD	70	550	138



	GPU	FPGA
Good Fit	No inter-dependences in the data flow and the computation can be done in parallel	Computation involves a lot of detailed low level hardware control operation which can not be efficiently implemented in high level languages, such as bit operation
	Applications contain a lot of parallelism, but involve computations which can not be efficiently implemented on GPU	A certain degree of complexity is required, and the implementation can take advantage of data streaming and pipelining
Bad Fit	Applications have a lot of memory access and have limited parallelism	Applications that require a lot of complexity in the logic and data flow design

Figure 1.7 Application Characteristic Fitness Categorization [33]

data. For those applications which stream large amounts of data through and do a lot of bit manipulations without floating point, or do a lot of complex data flow control, GPU cannot provide the peak performance. No fixed architecture is likely to solve a large cross-section of the problem space very efficiently. FPGAs come into place for optimum implementation of algorithms since the architecture can be defined based on the application. Both connectivity and processing capabilities can be tailored to suit the characteristics of the problems. The majority of the chip used by a CPU is not operating on each cycle [21]. In comparison, all of the hardware on FPGAs is executed in each clock cycle. FPGAs are more power efficient than CPU. The CUDA of NVIDIA GPU provides highly efficient coding environment. Application systems on FPGAs are usually developed using VHDL or Verilog which usually need a lot of effort and labor-intensive. As mentioned in [25], there probably is no one-size-fits-all architecture for scientific computing. CPU, GPU and FPGAs will be used in different scenarios.

1.3 High Performance Reconfigurable Computing

FPGAs have been around for tens of years in different application areas, such as prototyping and testing integrated circuits, custom hardware, but only recently found their way into HPC [17]. High performance reconfigurable computing(HPRC) platform[2] is based on conventional processors and FPGAs. HPRC system usually consists of high performance FPGAs, microprocessors and a comprehensive collection of peripheral components. The microprocessors are either hardcores such as PowerPCs, Intel CPUs, or soft-cores such as Microblaze and NIOSII. The hardcore processor has its own hardware separated from FPGA and can be fabricated into the same chip with FPGA or into different chips. The soft-core processor is implemented entirely in the programmable logic and memory blocks of FPGAs. The nature of softcore enables the designer to customize its components, such as defining custom instructions, pipeline depth, to meet specific application requirement. In the general HPRC design, the microprocessors fulfill the software function while the user-customized FPGA takes the role of hardware.

1.3.1 The Promise of HPRC

The synergistic HPRC systems have the potential to exploit coarse-grained functional parallelism as well as fine-grained instruction level parallelism through direct hardware execution on FPGAs [27]. People can full control over hardware at the lowest level and optimize it for each specific application by hand-crafting of caching strategies, pipelining, adders, multipliers, etc. Furthermore, FPGAs allows changing hardware for a specific application without fabrication cost. HPRC system has shown orders-of-magnitude improvement in performance, power, size, cost over conventional high performance computing(HPC) systems [27, 50, 13].

Another advantage of FPGAs is low power consumption. The power consumption and cooling issues in conventional HPC systems have been a big hurdle. As costs of power consumption and cooling system increase to exceed the total cost of software and hardware, numerous research claim that FPGAs are superior to other computing platforms. Hardware-software co-design has particularly strong potential with regards to energy efficiency [29]. For example, Xilinx virtex-5 device fabricated on a 65nm

[2] Reconfigurable computing is a new paradigm which is boarder than FPGA acceleration. Yet FPGAs are the focal point for the current research and commercial deployments.

process for typical designs is only 3 Watts, while Intel Core 2 Extreme is typically 60-70 Watts [17]. The clock frequency of FPGAs is usually one-tenth of that of CPUs.

Reconfigurable computing has been successfully deployed in data mining, aerospace and bioinformatics [11]. NASA, DOD, and other space-related agencies worldwide are increasingly featuring RC technologies in their platforms, as is the aerospace community in general [59]. Even though reconfigurable computing is viewed as lagging in effective tools for application development, many vendors and research groups are conducting to develop new tools and products to address this challenge. For example, Vforce [107] is a framework which insulating application-level code from hardware-specific details. Trident [146] open source compiler translates C code to a hardware circuit description. NSF center for High performance reconfigurable computing(CHREC) was opened in 2007 and is comprised of more than 30 leading organizations. CHREC directly support the research needs of industry and government partners and advance the knowledge and technologies in this emerging field [34]. Some high level languages are also developed, such as SRC Computers Carte C and Carte Fortran, Impulse Accelerated Technologies Impulse C, Mitrion C from Mitrionics, and Celoxicas Handel-C. There are also high-level graphical programming development tools such as Annapolis Micro Systems CoreFire, Starbridge Systems Viva, Xilinx System Generator, and DSPlogics Reconfigurable Computing Toolbox [50].

1.3.2 Challenges with HPRC

As will any new technology, there are promises and challenges. However, CPU and GPU are still more popular in HPC at present. There are two reasons for this. First, they are inexpensive, due to the massive market. Second, the design productivity is an important challenge at present for FPGA devices. To truly penetrate HPC, there are several challenges that FPGAs must overcome:

Portability After a new application is developed, they are expected to run on other hardware platform with minimum porting effort. Partitioning an algorithm into software and hardware parts among different platforms is a challenge task. There is no standardization for this as the MPI standard for MPPs(Massively Parallel Processing) [48].

Performance Application developers usually are not willing to switch the computing paradigm unless

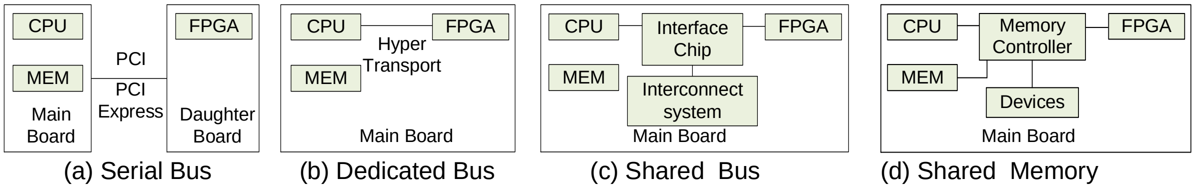


Figure 1.8 HPRC Architecture Overview

an order of magnitude improvement in overall application performance [48]. With the lower frequency, FPGAs must fully utilize the parallelism inside the FPGAs and applications to achieve this goal.

Cost Given a $5000 HPC node, integrating an FPGA chip can easily cost $10,000 [48]. FPGA based accelerators are expensive by HPC standards.

Productivity Even though the complexity of programming multi-core CPU will approach that of HPRC as the core numbers increase [17]. The effort to develop an application on HPRC platform is still very high. Adapting current HPC applications to run efficiently on a HPRC platform is a labor-intensive task.

1.3.3 HPRC Architecture

The original concept of HPRC can be traced back to 1960's [17]. The world's first commercial reconfigurable computer is Algotronix CHS2x4 in the beginning of 1990's. Since then the major FPGA vendors have collaborated with companies such DRC computer, XtremeData, Convey, and Cray to offer a plethora of high-performance reconfigurable platforms. Some of them will be examined below 1.8. For details of the history of HPRC, please refer [17].

Serial Bus FPGA daughter-boards are plugged into the CPU's serial bus, PCI or PCI express [17]. To get efficiency out of this kind of cheap system, the applications must have small volumes of IO data and a high compute to IO ratio. This is because the serial bus is easily overloaded and transferring data to and from the FPGA can take longer than running it on the host CPU.

Dedicated Bus There is directly point to point communication, such as QuickPath Interconnect and Hyper-transport [25], between FPGAs and CPUs. These platforms integrate FPGAs and traditional multi-core CPUs into one board using high-speed bus.

Shared Bus FPGAs and CPUs both are connected to the network interface chip which connects them to the host network. FPGAs cannot only communicate directly with CPUs, but also to other components. This provides arbitrary data and control communication patterns, rendering tremendous flexibility and complexity to the application developers [17]. However, there is contention between CPUs and FPGAs which may affect the performance negatively. One example of this architecture is Cray XD1.

Shared Memory The FPGAs are directly connected to a system-wide shared memory bus. This is a any-to-any network topology which requires the memory controller to maintain the coherency of all the connected devices. SGI's RASC architecture and the NUMAlink interconnect fabric fall into this category.

1.4 Roadmap

Chapter 2 describes a reconfigurable platform for frequent pattern mining(FPM) which is a data-compute intensive application. Then we focus on a data-intensive application called Sparse Matrix-Vector Multiplication(SMVM). An HPRC architecture for SMVM is presented in chapter 4. Due to the importance of floating-point accumulators in SMVM, a standalone chapter 3 is dedicated to a newly designed accumulator architecture. Both FPM and SMVM are essential tasks in data mining. To illustrate the advantage of our HPRC architecture, accelerating k-nearest neighbor in text classification using SMVM is discussed in chapter 5. Finally, we presented a general HPRC architecture in chapter 6 for data-intensive applications based on Map-Reduce computation model. The conclusion and future work are discussed in chapter 7.

CHAPTER 2. FREQUENT PATTERN MINING

In this chapter, we will introduce the first project toward achieving the goal of our research objectives. One prominent problem in data mining is called Frequent pattern mining which are designed to find commonly occurring sets in databases. It has found its way into a variety of fields, including medicine, biotechnology, and marketing. The class of frequent pattern mining algorithms is typically very memory intensive, leading to prohibitive runtimes on large databases. A class of reconfigurable architectures has been recently developed that have shown promise in accelerating some data mining applications. we propose a new architecture for frequent pattern mining based on a systolic tree structure. The goal of this architecture is to mimic the internal memory layout of the original pattern mining software algorithm while achieving a higher throughput. We provide a detailed analysis of the area and performance requirements of our systolic tree-based architecture, and show that our reconfigurable platform is faster than the original software algorithm for mining long frequent patterns.

2.1 Introduction

The goal of frequent pattern (or frequent itemset) mining is to determine which items in a transactional database commonly appear together. Given the size of typical modern databases, an exhaustive search is usually not feasible, making this a challenging computational problem. The *FP-growth* algorithm [73] stores all transactions in the database as a tree using two scans. The FP-growth algorithm was originally designed from a software developer's perspective and uses recursion to traverse the tree and mine patterns. It is cumbersome (and sometimes impossible) to implement recursive processing directly in hardware, as dynamic memory allocation typically requires some software management. For this reason, the dynamic data structures (such as linked lists and trees) which are widely used in software implementations are very rarely used in a direct hardware implementation. Consequently, it would be very difficult to directly translate this FP-growth algorithm into a hardware implementation.

In [139], we have previously introduced a reconfigurable systolic tree architecture for frequent pattern mining, and describe a prototype using a Field Programmable Gate Array (FPGA) platform. The systolic tree is configured to store the support counts of the candidate patterns in a pipelined fashion as the database is scanned in, and is controlled by a simple software module that reads the support counts and makes pruning decisions. In this chapter, we focus on improving the original scheme introduced in [139] by eliminating the counting nodes, and provide a new COUNT mode algorithm. For each mode we also present the design principle and formal proof of the correctness. A new database projection approach which is suitable for mining a systolic tree is proposed and implemented. Based on the reconfigurable platform presented, the area requirement of FPGAs and software/hardware mining time are studied. We performed experiments over several benchmarks. The experimental results show that our FPGA-based approach can be several times faster than a software implementation of the FP-growth algorithm.

In [139], we have previously introduced a reconfigurable systolic tree architecture for frequent pattern mining, and describe a prototype using a Field Programmable Gate Array (FPGA) platform. The systolic tree is configured to store the support counts of the candidate patterns in a pipelined fashion as the database is scanned in, and is controlled by a simple software module that reads the support counts and makes pruning decisions. In this chapter, we focus on the original scheme introduced in [139] by eliminating the counting nodes, and provide a new COUNT mode algorithm. For each mode we also present the design principle and formal proof of the correctness. A new database projection approach which is suitable for mining a systolic tree is proposed and implemented. Based on the reconfigurable platform presented, the area requirement of FPGAs and software/hardware mining time are studied. We performed experiments over several benchmarks. The experimental results show that our FPGA-based approach can be several times faster than a software implementation of the FP-growth algorithm.

2.2 Related Work

The rapidly increasing size of transactional databases has sparked research into the parallelization and optimization of those algorithms that aim to extract useful information from raw data. In [164], the authors propose a parallel version of the Apriori algorithm. In [123], the FP-growth algorithm is implemented on a PC cluster. Though these parallel computing platforms can improve the runtime of data mining algorithms, the observed acceleration has not scaled well with the number of processors. The parallel Apriori implementation obtained speedups of 13.4× and 22.6× with 16 and 32 nodes, respectively. The parallel FP-growth implementation scaled more poorly, with a reported 2× speedup on 8 nodes.

One advantage that FPGAs have over traditional computing platforms is the ability to parallelize algorithms at the operand-level granularity, as opposed to the module-level or higher. Consequently, there have been several recent efforts into hardware-based accelerators for data mining algorithms. In [110] the authors describe a hardware architecture for a Decision Tree Classification (DTC) algorithm, and show that optimizing the Gini score computation significantly increases the overall performance. Hardware acceleration for density-based clustering algorithms is investigated in [121]. By mapping the K-means clustering algorithm into FPGA hardware, the hardware implementation in [53] is 200× faster than the software version.

A parallel implementation of the Apriori algorithm on FPGAs was first done in [15]. Due to the processing time involved in reading the transactional database multiple times, the hardware implementation was only 4× faster than the fastest software implementation. The same authors further explored the parallelism in Apriori and developed a bitmapped Content Addressable Memory (CAM) architecture that provided 24× performance gain over the software version [16]. Achieving a better speedup using Apriori will be difficult due to the nature of the algorithm, which must read the entire database once for every item in the worst case. The HAPPI architecture proposed in [158] applies the pipeline methodology to resolve the the bottleneck of Apriori-based hardware schemes. HAPPI outperforms the architecture in [15] when the number of different items and the minimum support values are increased [158].

As previously mentioned, FP-growth algorithms are an alternative to Apriori-based solutions. Conclusions from several independent evaluations indicate that FP-growth is one of the best-performing association rules mining algorithms [61, 73][1]. While software solutions exist [24, 123], we are unaware

[1] According to the extensive benchmarks of the FIMI Workshop as part of ICDM'04 [18], both LCMv2 [150] and kDCI [99] are very competitive algorithms for common mining tasks when compared with FP-growth. For this reason, analyzing and accelerating LCMv2 and kDCI is a promising avenue for future work.

ID	Items	ID	Items
1	B,C,D	5	A,B,C
2	B,C	6	A,B,C
3	A,C,D	7	A,B,D
4	A,C,D		

Figure 2.1 A Simple Transactional Database

of any existing hardware implementations of FP-growth algorithms. Most notably, our architecture only requires two scans of the database, similar to FP-growth software implementations. The frequent items are identified in the first scan. In the second scan of the database, the non-frequent items are pruned from each transaction. The items are also reordered in order to enhance the compactness of the FP-tree. In contrast with the FP-tree approach, the reorder operation is not necessary when using a systolic tree because the size of the systolic tree is decided by the number of frequent items. However, each frequent item is assigned a sequence number which is used in itemset matching.

2.3 Systolic Tree:Design and Creation

2.3.1 Frequent Pattern Mining

Given a transactional database D, where each row represents a transaction as shown in Figure 2.1, let $I = \{i_1, i_2, \cdots, i_n\}$ be the set of all items in the database D and $D = \{t_1, t_2, \cdots, t_d\}$ be the set of all transactions. For any transaction t_i, $t_i \subseteq I$. Let $\Gamma(P)$ be the support count of a pattern (itemset) P, where P is a set of items. $\Gamma(P) = |\{t_i | P \subseteq t_i,\, t_i \in D\}|$ is equal to the number of transactions containing P. A pattern (itemset) P is frequent if $\Gamma(P)$ is no less than a predefined minimum support threshold ξ. The objective of frequent pattern mining is to find the set of patterns in D which satisfy $\Pi(D) = \{P | \Gamma(P) \geq \xi\}$. In Figure 2.1, $I = \{A, B, C, D\}$ and let $\xi = 4$. $\Gamma(\{A, C, D\}) = 2 < 4$, hence $\{A, C, D\}$ is not frequent; $\Gamma(\{B, C\}) = 4 \geq 4$, hence $\{B, C\}$ is frequent.

```
Algorithm 1   FP-tree Construction Algorithm

Input:  A transactional database DB
        A minimum support threshold $\xi$
Output:  The FP-tree for that DB
  1. Scan the transactional database DB once. Collect F, the set of
     frequent items, and the support of each frequent item. Sort F
     in support-descending order as FList, the list of frequent items.
  2. Create the root of an FP-tree, T , and label it as “null”.
     For each transaction Trans in DB do the following.
     Select the frequent items in Trans and sort them according to
     the order of FList. Let the sorted frequent-item list in Trans
     be $[p|P]$, where p is the first element and P is the remaining
     list. Call $insert\_tree([p|P], T)$.

  The function $insert\_tree([p|P], T)$ is performed as follows.
  If T has a child N such that N.item-name = p.item-name, then
  increment N’s count by 1; else create a new node N, with its
  count initialized to 1, its parent link linked to T, and its node-link
  linked to the nodes with the same item-name via the node-link
  structure. If P is nonempty, call $insert\_tree(P, N)$ recursively.
```

2.3.2 FP-Tree

The Frequent-Pattern (FP) tree [73] is a compact data structure to store frequent itemsets.

Definition 1 (**FP Tree**). *An FP-tree is a tree structure which is defined below:*

1. *It consists of one root labeled as “null”, a set of item-prefix subtrees as the children of the root, and a frequent-item-header table.*

2. *Each node in the item-prefix subtree consists of three fields: item-name, count, and node-link, where item-name registers which item this node represents, count registers the number of transactions represented by the portion of the path reaching this node, and node-link links to the next node in the FP-tree carrying the same item-name, or null if there is none.*

3. *Each entry in the frequent-item-header table consists of two fields, (1) item-name and (2) head of node-link (a pointer pointing to the first node in the FP-tree carrying the item-name).*

Based on the definition, the FP-tree construction algorithm from [73] is shown below: There are two scans of the database in constructing an FP-tree. The frequent items are determined in the first scan while the infrequent items are discarded. The frequent items are then sorted in decreasing order

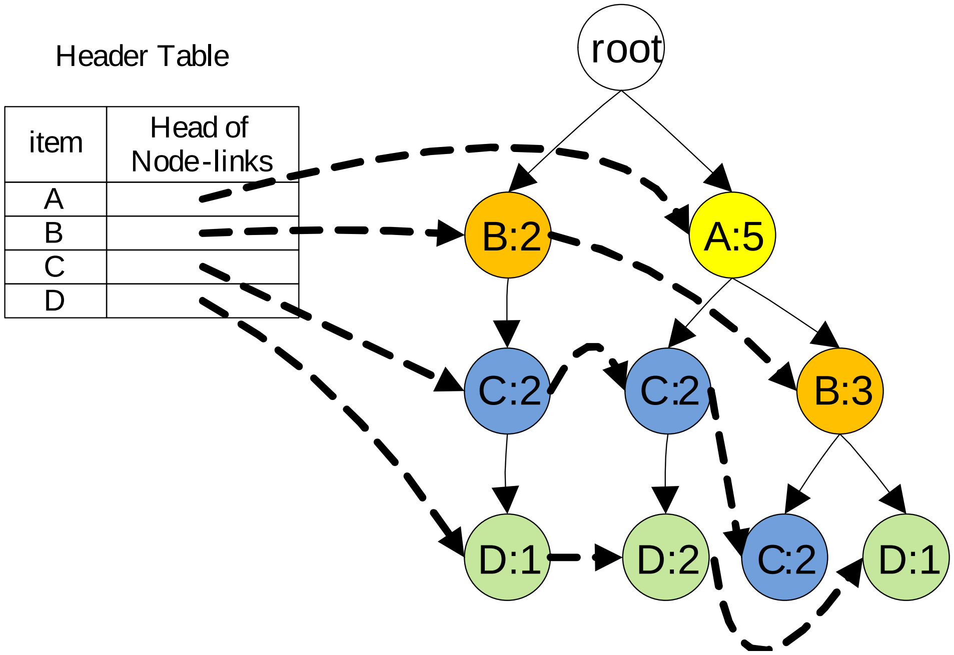


Figure 2.2 FP-Tree (Software Data Structure Representation)

of support counts so that fewer nodes in the FP-tree are used to represent the transactions. The sorting according to the support count is not required in the systolic tree due to the fixed structure of the systolic tree. However, all items must be shown in a consistent order. For example, item A must be placed before item B in a transaction as shown in Figure 2.1. And item B must be placed before item C, and so on.

The FP-tree is constructed in the second scan. The FP-tree is constructed by reading the transactions one at a time and mapping each transaction onto a path in the FP-tree. In Figure 2.2, each node contains an item along with an integer which is the count of that item in that path. Initially the FP-tree contains only the root node. After reading a new transaction, the algorithm starts from the root node to look for a node whose label matches the first item in the transaction. If such a node is not found, a new node is created with the label as the first item. Otherwise, the algorithm starting from the current node looks for a child whose label matches the second item of the transaction. A node's counter is increased by one in the case of a match. This process continues until every transaction is mapped to a path in the FP-tree. For Figure 2.1, suppose the minimum support threshold is 4 ($\xi = 4$) and all items are frequent. The FP-tree constructed from Figure 2.1 is shown in Figure 2.2. For illustration purposes, the items are not sorted in the support-decreasing order but alphabetic order. We will come back to the FP-growth algorithm in Section 2.5.2.

2.3.3 Systolic Tree

In VLSI terminology, a *systolic tree* is an arrangement of pipelined processing elements (PEs) in a multi-dimensional tree pattern [22]. The specific structure of the tree and each processing element are application dependent. The goal of our architecture is to mimic the internal memory layout of the FP-growth algorithm while achieving a much higher throughput. The role of the systolic tree as mapped in FPGA hardware is then similar to the FP-tree as used in software. Given a transactional database the relative positions of the elements in the systolic tree should be the same as in the FP-tree. To achieve this objective, the systolic tree in this work can be designed based on the following observations:

1. A control PE which corresponds to the root node in the FP-tree acts as the input and output interfaces to the systolic tree.

2. The systolic tree construction algorithm starts from the root node upon receiving a new transaction. For two nodes which store any two items in a transaction, one node must be an ancestor of the other. If two transactions share a common prefix, the shared parts are merged using one prefix path. The degree of a node is equal to the number of transactions which share the common prefix. In the worst case, the degree of a node is equal to the number of frequent items N. The PE in hardware cannot be created and deleted dynamically as in software[2]. If each PE is connected to N child PEs, the hardware structure and operation in each PE will be very complex. To avoid this hindrance, each PE in the systolic tree connects to its leftmost child. The other children connect to their leftmost siblings. To reach the rightmost child, the signal from the parent PE must travel through all the children on the left.

3. In the FP-growth algorithm, the first item in each transaction is stored in a child of the root while the other items are stored in its descendants. However, all PEs in the systolic tree operate in parallel and there is no ‘pointer’ concept in the hardware implementation. Due to the intrinsic characteristics of the tree, the operation of a transaction usually starts from the root. Thus, the

[2] If the reconfiguration latency is small enough, performing a dynamic reconfiguration of the systolic structure may be beneficial for performance. We are considering the use of partial run-time reconfiguration on FPGAs in the future.

control PE acts as the input/output interface of the systolic tree. Each clock cycle, an item is transferred to the control PE which may forward the item to its children.

Based on the observations shown above, the systolic tree can be defined as follows:

Definition 2 (**Systolic Tree**). *A systolic tree is a tree structure which consists of the following PEs:*

1. **Control PE**. *The root PE of the systolic tree does not contain any item. Any input/output data of the systolic tree must go through it firstly. One of its interfaces connects to its leftmost child.*
2. **General PE**. *All other PEs are general PEs. Each general PE has one bidirectional interface connecting to its parent. The general PE which has children has one interface connecting to its leftmost child. Those general PEs which have siblings may have an interface connecting to its leftmost sibling. The general PE may contain an item and the support count of the stored item.*

Each PE has a **level** *associated with it. The control PE is at level 0. The level of a general PE is equal to its distance to the control PE. The children of a PE has the same level.*

From the definition of the systolic tree, we have the following properties of the systolic tree.

Property 1. *Each general PE has only one parent which connects to its leftmost child directly. The other children connect to their parents indirectly through their left siblings.*

Property 2. *A systolic tree which has W levels with K children for each PE has $K^W + K^{W-1} + ... + K + 1 = \frac{K^{W+1}-1}{K-1}$ PEs.*

For example in Figure 2.3, there are in total $\frac{K^{W+1}-1}{K-1} = \frac{2^{3+1}-1}{2-1} = 15$ PEs with $W = 3$, $K = 2$. A PE has three modes of operation: *write* mode, *scan* mode, and *count* mode which will be discussed in the following sections. The systolic tree is built in write mode (Algorithm 2). Input items are streamed from the root node in the direction set by the defined write mode algorithm. The support count of a candidate itemset is extracted in both scan mode and count mode. We refer to this process as candidate itemset *matching*. In the next section, we will introduce the construction of the systolic tree.

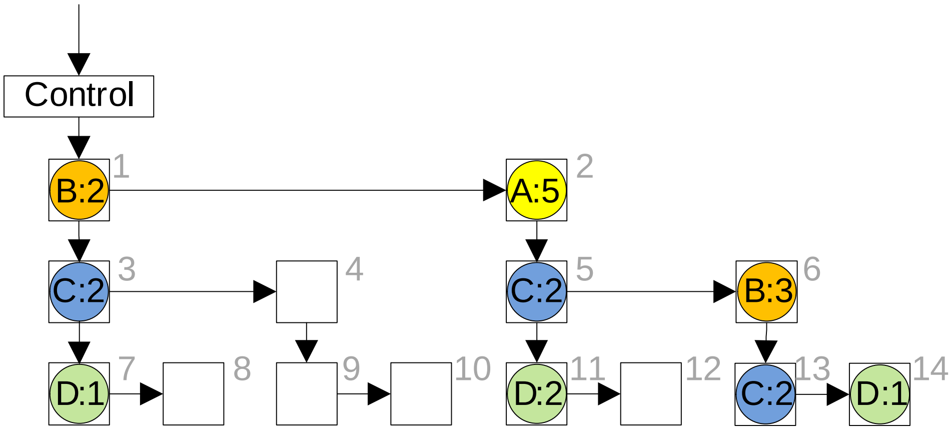


Figure 2.3 Equivalent Systolic Tree-based Hardware Architecture for Acceleration

2.3.4 Systolic Tree Creation

In the FP-growth algorithm, a new item is inserted into the FP-tree by searching the tree in a top-down fashion. The item in the child pointed by the parent is compared with the item in the transaction. If the item in the current child does not match, the next child will be searched. The process will continue until a match is found or a new child is created. During the FP-tree creation, the nodes are stored in memory and processed passively; the pointers are used to trace the path. In contrast, each PE in our hardware implementation processes the input data actively. Without pointers, the PEs are hard-wired together by interfaces. Therefore, the software algorithm to create the FP-tree can not be used in hardware. A hardware architecture to import the transactions into the systolic tree must be proposed. Conciseness and simplicity imply high area-performance efficiency in hardware. With this observation, a hardware algorithm which runs in each PE is presented in Algorithm 2.

The design principle of the WRITE mode algorithm is that the built-up systolic tree should have a similar layout with the FP-tree given the same transactional database. The i^{th} item in a transaction is mapped to the i^{th} level in the systolic tree. For any two PEs in the path of the same transaction, the PE in the i^{th} level is the ancestor of the PE in the $i + 1^{th}$ or higher level. A PE is never in the same path of a transaction with its siblings. Suppose the transactional database has N frequent items. The number of PEs in the first level is at most N. The depth of the systolic tree is at most N. Suppose all items in Figure 2.1 are frequent. The first items in all transactions only include items A and B. Therefore we only need two PEs in the first level of the systolic tree. However counting the number of items in each level may impair the overall pattern mining performance. Thus the values of K and W are usually set to be equal to N. The last item of a transaction may not be put into the leaf of the systolic tree if the number of items in the transaction is less than N. If two transactions share the same prefix, they will share a path in the systolic tree. For example, the sixth and seventh transactions in Figure 2.1 will share a common path which includes items A and B. In summary, the design intuition behind the WRITE mode algorithm is that the path an item travels through the FP-tree is the same as the path it travels in the systolic tree.

The WRITE mode algorithm is presented in Algorithm 2. The same algorithm runs in each PE of the systolic tree every clock cycle. That is, the inner hardware structure of each PE is the same. Initially all PEs are empty. An item is loaded into the control PE each clock cycle which in turn transfers each item into the general PEs. After all items in a transaction are sent to the systolic tree, a control signal that states the termination of an old transaction and the start of a new one is sent to the control PE. The signal will be broadcast to all PEs which reinitialize themselves for the next transaction. The initialization includes resetting $match$ and $Inpath$ flags in the first line of Algorithm 2. The input of the algorithm is an item i_t. The $match$ flag is set when the item in the PE matches i_t. The $Inpath$ flag is not set when the PE does not contain any item from the current transaction. For example, the PE under the control PE in Figure 2.3 should not contain item B in a new transaction {A,B,C}. Upon receiving a new item i_t, the three if statements in Algorithm 2 are evaluated sequentially. The two switches $match$ and $InPath$ are used to make sure that the latter items in a transaction will follow the path of the former items in the same transaction. The first if statement allocates a PE for the incoming item i_t if it appears in the current path for the first time. In this case, the transaction represented by the items contained in the PEs from the root to the current one is a new transaction which has never been put into the systolic tree before. The second if statement is executed if the current item matches the item in the current PE. In this case, the transaction represented by the items contained in the PEs from the root to the current one is a transaction which has been put into the systolic tree before. Matching happens in the first and second if statements; thus the items will not be forwarded further. If a PE is in the path of the input transaction, $match$ and $InPath$ must be set. As discussed above, a PE is never in the same transaction with its siblings. If the incoming item does not match the current PE, the third if statement is triggered. There are two scenarios when the mismatch happens. One is that the current

Algorithm 2 WRITE Mode Algorithm in Each PE

```
**Input:** Input item $i_t$
  match← 0; InPath← 1
  **if** PE is empty **then** //Step (1)
    store the item $i_t$; $count \leftarrow 1; match \leftarrow 1$; stop forwarding
  **else if** ($i_t$ is in PE) and (InPath=1) **then** //Step (2)
    $match \leftarrow 1$; count++; stop forwarding
  **else if** match=0 **then** //Step (3)
    $Inpath \leftarrow 0$; forward $i_t$ to the sibling
  **else** //Step (4)
    forward $i_t$ to the children
  **end if**
```

PE is in the path of the input transaction. The incoming item should be forwarded to its children. The match will happen after several transfers. The other is that the current PE is not in the path of the input transaction. It should transfer the incoming item to its rightmost sibling.

Let's illustrate the creation of a systolic tree with the example shown in Figure 2.3. In order to clearly differentiate PEs, a number in light scale is placed in the top-right corner.

1. Initially all PEs are empty and do not contain any item.

2. The first transaction $\{B, C, D\}$ in Figure 2.1 is scanned firstly. In the first clock cycle, item B enters the control PE. In the second clock cycle, item C enters the control PE and item B enters PE1. The first if statement in Algorithm 2 is executed in PE1. Item B is stored in PE1 and not forwarded further. The count corresponding to B is 1, and the $match$ flag is set to 1. In the third clock cycle, item D enters the control PE and item C enters PE1. Step (4) is triggered in PE1 and item C is send to PE3.

3. In the fourth clock cycle, a control signal stating the termination of the current transaction and the start of the new transaction is sent to the control PE. D enters PE1 and C enters PE3. Step (4) is executed in PE1 and D is sent to PE3. In PE3 the first if statement is executed and item C is stored and not forwarded.

4. In the fifth clock cycle, the first item B of the second transaction in Figure 2.1 enters the control PE. The control signal enters PE1. The $match$ and $InPath$ are initialized in PE1. Item D in the first transaction enters PE3. Step (4) is executed in PE3 and D is sent to PE7.

5. In the sixth clock cycle, the second item C of the second transaction in Figure 2.1 enters the control PE. Item D enters PE7 and is not forwarded further. Item B in the second transaction enters PE1 and the step (2) in Algorithm 2 is triggered. The count in PE1 is increased by 1 and $match$ is set. The control signal enters PE2 and PE3 and initializes the flags.

6. In the seventh clock cycle, a control signal stating the termination of the current transaction and the start of the new transaction is sent to the control PE. The item C enters PE1 and step (4) is triggered.

7. In the eighth clock cycle, the first item A of the third transaction enters the control PE. The control signal enters PE1. The item C enters PE3 and step (2) is executed. The count in PE3 is increased.

8. The items in the transactional database enter the systolic tree sequentially. After all transactions are stored in PEs, the state of the systolic tree is shown in Figure 2.3.

2.3.5 Completeness of the Systolic Tree

We can now provide some formal assurances that the Algorithm 2 builds the systolic tree correctly.

Lemma 1. *During the creation of the systolic tree, the items arriving later follow the path of the previous items in the same transaction.*

Proof A control signal initializes $match$ and $InPath$ in all PEs in the systolic tree in the beginning of a transaction. According to the Algorithm 2, the first item is either stored in a PE or forwarded to the sibling of the current PE. In the PE of the former case, $match = 1$ and $InPath = 1$; In the PEs of the latter case, $match = 0$ and $InPath = 0$. An empty PE or a PE containing the item will be found after some clock cycles in the latter case. Suppose the PE in the first level where the first item resides is $PE^i(i \geq 1)$. The $(i-1)$ PEs on the left have $match = InPath = 0$ and are not empty. In PE^i, $match = 1$ and $Inpath = 1$. The other PEs have $match = 0$ and $InPath = 1$. The second item of the transaction is sent to the $(i-1)$ PEs on the left firstly. Since these PEs are not empty and have $match = 0$, $InPath = 0$, step (3) is triggered. The second item is forwarded to the right PEs cycle by cycle until the PE^i. Since each item in a transaction is unique, the second item is different from the first item. According to the write mode algorithm, step (4) is triggered. The second item will be sent to a child of PE^i. A PE in the second level will be located for the second item. The third item will follow the path of the second item in a similar way. The other items can be analyzed similarly. □

Lemma 2. *A new transaction not sharing a prefix with previous transactions will always be stored correctly in the systolic tree.*

Proof The first item the new transaction is stored in the first level of the systolic tree. Since the new transaction does not share any prefix with stored transactions, an empty PE^i is located for the first item. As stated in Lemma 1, the latter items follow the path of the former items. Consequently, all children of PE^i must be empty. The second item will follow the path of the first item and be stored in the first child of PE^i. The other items in the new transactions will be stored in the systolic tree similarly. The $count$ is equal to 1 in all PEs storing the new transaction. □

Lemma 3. *A new transaction sharing a prefix with previous transactions will always be stored correctly in the systolic tree.*

Proof There may be multiple stored transactions sharing a prefix with the new transaction t_u. Suppose t_v is one of the stored transactions which share the longest prefix with t_u. Let t_u=\{$item_1^u$, $item_2^u$, $\cdots$, $item_{|t_u|}^u$\} and t_v=\{$item_1^v$, $item_2^v$, $\cdots$, $item_{|t_v|}^v$\}, $item_i^u$=$item_i^v$, for $i = 1, 2, \cdots, s$, where s is the length of the prefix. t_v is mapped to a path $p_v = \{PE_1^v, PE_2^v, \cdots, PE_{|t_v|}^v\}$ in the systolic tree. PE_i^v is on the i^{th} level of the systolic tree.

According to the Algorithm 2, the PEs which are the left siblings of $PE_i^v(i \leq s)$ execute step (3). Step (2) is executed in $PE_i^v(i \leq s)$ upon receiving the i^{th} item of the new transaction t_u. Step (4) is executed in $PE_i^v(i \leq s)$ upon receiving the $j^{th}(j > i)$ item in t_u. When PE_s^v receives the $s+1^{th}$ item, step (4) is triggered and the $s + 1^{th}$ item is sent to the children of PE_s^v. Since t_v shares the longest prefix with t_u, an empty PE in the $s + 1^{th}$ level is allocated for the $s + 1^{th}$ item. The rest of the items in t_u are stored in a similar way, as discussed in Lemma 2. □

Theorem 1. *Given a transactional database, the systolic tree will be built up correctly using the WRITE mode algorithm.*

Proof Initially the systolic tree is empty. After the systolic tree is built up, a path can be found from the control PE to any non-empty PE. For an arbitrary path $(PE_0, PE_1, PE_2, \cdots, PE_k)$ in the systolic tree where PE_0 is the control PE, let C_{PE_k} be the count at the PE_k and C'_{PE_k} be the sum of counts of children PEs of PE_k. According to the Algorithm 2, the path registers $C_{PE_k} - C'_{PE_k}$ transactions.

Based on Lemma 2 and Lemma 3, each transaction in the database is stored to one path in the systolic tree. Therefore, the systolic tree registers the complete set of transactions. □

2.4 Pattern Mining Using Systolic Trees

The FP growth algorithm generates frequent itemsets by exploring the FP-tree recursively in a bottom-up fashion [73, 141]. Starting from the leaves of the FP-tree, the FP-growth finds all frequent itemsets ending with a specific suffix in a divide-and-conquer strategy and splits the problem into smaller subproblems. The conditional FP-trees converted from the prefix-paths are created during the exploration. However, a recursive implementation is cumbersome in an FPGA implementation. To mine frequent patterns in the systolic tree, a collaborating hardware/software platform is required. The software sends a candidate pattern to the systolic tree. After some clock cycles, the systolic tree sends the support count of the candidate pattern back to the software. The software compares the support count with the support threshold and decides whether the candidate pattern is frequent or not. After all candidate patterns are checked with the support threshold in software, the pattern mining is done.

The approach to get the support count of a candidate pattern is called candidate itemset (pattern) matching which will be discussed in Section 2.4.1 in detail. The matching must be performed after the systolic tree is built. When the tree is in itemset matching phase, PEs are in SCAN mode. The platform architecture of the software and hardware co-design will be introduced in Section 2.6.

2.4.1 Candidate Itemset Matching

The main principle of matching is that any path containing the queried candidate itemset will be reported to the control PE. Note that such a path may contain more items than the queried itemset. The algorithm to find the paths is discussed in this section. Reporting the support count of the candidate itemset will be studied in Section 2.4.3. Before introducing itemset matching, we examine some useful properties of the systolic tree which will facilitate frequent pattern mining. As mentioned in previous sections, each frequent item is assigned a sequence number. The items in each transaction enter the systolic tree in an increasing order in both the WRITE mode algorithm and the SCAN mode algorithm.

Lemma 4. *The support count stored in a PE is no less than the support count stored in the children PE; the item stored in a PE is smaller than the items stored in the children PE.*

Proof According to the construction process of the systolic tree, the i^{th} item of a new transaction with i items is stored in the i^{th} level of the tree. The support count in each PE on the path is increased by 1. Those PEs which are on $i + 1^{th}$ or higher levels have the support count unchanged. Therefore, the support count in a parent PE is no less than the descendant PEs on the same path. Besides, the items in each transaction enter the systolic tree in an increasing order. The item stored in a PE must be smaller than the items stored in its children. □

Lemma 5. *For any non-empty PE_i, the support count of the itemset represented by the path from the root to PE_i is equal to the support count stored in PE_i; the support count of any itemset including $item_i$ in the path is equal to C_i.*

Proof Suppose the itemset represented by the path from the root to PE_i is $\{item_1\colon C_1$, $item_2\colon C_2$, $\cdots$,$item_i\colon C_i\}$. From Lemma 4, we know $C_1 \geq C_2 \geq \cdots \geq C_i$. The support count of itemset $\{item_1$,$\cdots$,$item_j\}(1 \leq j \leq i)$ is equal to C_j. Therefore, the support count of itemset $\{item_1$, $item_2$, $\cdots$, $item_i\}$ is C_i.

Since the support count of $item_i$ is C_i, the support count of any itemset including $item_i$ is no more than C_i. Meanwhile, the support count of any itemset including $item_i$ is no more than C_i since C_i is the smallest among C_1,$\cdots$,C_i. Therefore, the support count of any itemset in the path including $item_i$ is equal to C_i. □

To facilitate the understanding of other properties of a systolic tree related to candidate itemset matching, we first go through an example which performs matching on the constructed systolic tree in Figure 2.3. Suppose the candidate itemset to be dictated is $\{C, D\}$. The paths containing the itemset in Figure 2.3 are $\{PE1, PE3, PE7\}$ and $\{PE2, PE5, PE11\}$. The path $\{PE1, PE3, PE7\}$ can also be expressed as $\{B\colon 2, C\colon 2, D\colon 1\}$ where the number after ':' indicates the support count. The first path indicates that itemset $\{B, C, D\}$ appears once in the database. Notice the path indicates that itemset $\{B, C\}$ appears twice. And itemset $\{B, C\}$ appears once together with item D. According to Lemma 5, the count stored in PE7 is the support count of C, D. Therefore, PE7 should report its count C_7 to the software. In path $\{PE2, PE5, PE11\}$, $\{A, C, D\}$ appears twice and itemset $\{A\}$ appears five times in the database. However, $\{A\}$ only appears twice with items C and D. According to Lemma 5, PE11 should report its count C_{11} to the software. The total support count of the itemset $\{C, D\}$ is equal to $C_7 + C_{11} = 1 + 2 = 3$.

The first step to get the support count of a candidate itemset is to locate those PEs which contain the last item. Those PEs called $reporting\ PEs$ are responsible for reporting the stored count. In the second step, the support counts stored in reporting PEs are reported to the software. In this section, we mainly focus on the first step. The second step is described in Section 2.4.3. The design goal of the candidate itemset matching is to find the reporting PEs. Since each PE has independent hardware components and stored data, a flag $IsLeaf$ is set in the reporting PE. In the second step, each PE checks its own $IsLeaf$ flag. If it is set, the PE will report its count. The matching algorithm should run after the systolic tree is built up. A signal which indicates the SCAN mode is first broadcast from the control PE to all PEs. Similar to the creation of the systolic tree, such a control signal enters the control PE before the first item of each candidate itemset enters the systolic tree. The signal initializes the data in each PE.

The design principle of the SCAN mode algorithm is similar to that of the WRITE mode algorithm. The items arriving later in the candidate itemset follow the path of the previous items. However, the goal of the WRITE mode algorithm is to find the path which shares the same prefix with the new transaction. In contrast, the goal of the SCAN mode algorithm is to locate all the paths which contain the dictated itemset. In the SCAN mode algorithm, the PEs where the last item resides are the reporting PEs. Since there may be multiple reporting PEs, some items in the candidate itemset must be duplicated when forwarded in PEs. As discussed in previous sections, each PE has two output interfaces. One is connected to its right sibling; the other is connected to its leftmost children. In each clock cycle, the new incoming item is only forwarded to the paths which contain the candidate itemset. Thus we assume there are two doors which are corresponding to the two interfaces in each PE. The bottom door is connected to the leftmost child. The right door is connected to the sibling. The door is locked when the incoming item should not be sent to the sibling or children. Suppose the itemset to be dictated is $\{item_1$, $item_2$,$\cdots$,$item_s\}$ and the items arriving later in the candidate itemset follow the path of the

```
Algorithm 3   SCAN Mode Algorithm in Each PE
Input: Input item $i_t$
  Open the bottom door; $IsLeaf \leftarrow 0$
  if PE is empty then //Step (1)
    stop forwarding
  else if ($i_t$ is in PE) and (bottom door is open) then //Step (2)
    $Isleaf \leftarrow 1$; forward $i_t$ to the sibling
  else if $i_t <$ the item in PE then //Step (3)
    $IsLeaf \leftarrow 0$; close the bottom door; forward $i_t$ to the sibling
  else if $i_t >$ the item in PE then //Step (4)
    $IsLeaf \leftarrow 0$;forward $i_t$ to the sibling;
    forward $i_t$ to the child if the bottom door is open
  end if
```

previous items (we will provide a formal proof for this later). The items are transferred into the built-up systolic tree one in each clock cycle. All the PEs run in a pipelined manner. The design of the SCAN algorithm is based on the following facts:

Lemma 6. *The bottom door in a PE should be locked if the input item is smaller than the stored item.*

Proof Suppose the current PE is PE_c and the stored item is $item_c$. The set of the paths which include PE_c is P^c. According to Lemma 4, the item stored in a PE is smaller than the items stored in the children PE. If the incoming $item_t$ is smaller than $item_c$, $item_t$ must be smaller than the stored items in the descendants of PE_i. The algorithm will not find a PE in P^c whose stored item is equal to $item_t$. Therefore, the dictated itemset will never be contained in P^c. It is meaningless to forward $item_t$ to the children of PE_c. Thus the bottom door should be locked. □

From this lemma, a corollary can be directly derived:

Corollary 1. *The PE with a locked bottom door will never be in a path which includes the candidate itemset.*

In step (3), even though $item_t$ is smaller than the stored item $item_c$, $item_t$ should be sent to the siblings of PE_c. This is because the items in the siblings may be smaller than $item_c$ or equal to $item_t$. If the input item $item_t$ is larger than the stored item $item_c$, the stored items in the siblings or children of PE_c may be equal to $item_t$. Thus $item_t$ should be forwarded to all open doors as shown in step (4). For example, we want to check the support count of a candidate itemset $\{B, D\}$ in the systolic tree shown in Figure 2.3. In the clock cycle when the item D enters the PE1, it should be sent to both PE2 and PE3 since both of the paths $\{PE1, PE3, PE7\}$ and $\{PE2, PE6, PE14\}$ contain $\{B, D\}$. The SCAN mode algorithm is shown in Algorithm 3. The rationale of step (3) and step (4) has been explained above. According to Corollary 1, a matching happens if $item_t$ is equal to $item_c$ and the bottom door is open as shown in step (2). The $item_t$ still needs to be sent to the siblings of PE_c in case some descendants of the siblings has a stored item matching $item_t$. In sum, $item_t$ should be sent to the sibling of PE_c regardless of its relationship with $item_c$. That is, the right door of PE_c should always be open. Thus the right door is open by default in the SCAN mode algorithm. The $IsLeaf$ flag is set if PE matches the last item in the queried candidate itemset. The PE with $IsLeaf$ set is responsible for reporting the number of the candidate itemset to the parent. Since the item in the candidate itemset is sent one by one, the flag $IsLeaf$ is cleared if another item in the candidate itemset which is larger than the stored item passes through the PE. Whenever a PE receives the control signal which indicates a new itemset matching, the bottom door is opened and the $IsLeaf$ flag is cleared.

Let's illustrate the whole matching process with the example shown in Figure 2.4. Suppose the threshold support count is four ($\xi = 4$). Now we want to check whether the candidate itemset {B,D} is frequent.

1. In the first clock cycle, a control signal which indicates the SCAN mode is first broadcast from the control PE.

2. In the second clock cycle, the item B is sent to the control PE. The control signal is sent to PE1 and then broadcast to other PEs in subsequent clock cycles.

3. In the third clock cycle, PE1 receives B. Step (2) in Algorithm 3 is triggered and B is forwarded to PE2. The $IsLeaf$ flag is set in PE1. The item D enters the control PE.

4. In the fourth clock cycle, PE2 receives the item B. Step (4) is triggered in PE2. The item B is sent to PE5. PE1 receives the item D. Step (4) is executed in PE1. The $IsLeaf$ flag is cleared in PE1.

5. In the fifth clock cycle, the item D enters PE2 and PE3. The item B enters PE5. Step (4) is executed in PE2 and PE3. Step (3) is executed in PE5. The bottom door of PE5 is locked. The item B is sent to PE6.

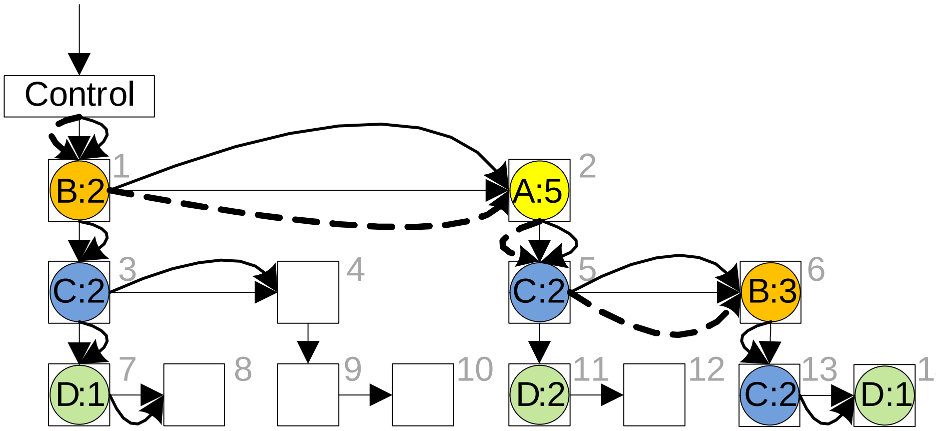


Figure 2.4 SCAN Mode Example

6. In the sixth clock cycle, the item D enters PE4, PE7 and PE5. The item B enters PE6. Step (1) is executed in PE4. Step (2) is executed in PE7. The $IsLeaf$ flag is set in PE7. Step (4) is triggered in PE5. Step (2) is executed in PE6.

7. In the seventh clock cycle, the item D enters PE8 and PE6. Step (4) is executed in PE6. The $IsLeaf$ flag is cleared.

8. In the ninth clock cycle, the item D enters PE14 and step (2) is executed. The $IsLeaf$ flag is set in PE14.

In the end the $IsLeaf$ flag is set in PE7 and PE14. As will be explained in the following section, these two PEs report their item count to the parent. The count in both PE7 and PE14 is one. Thus the total count of the candidate itemset $\{B, D\}$ is two which is less than four. Therefore, this candidate itemset is not frequent. In Figure 2.4, the dashed lines show the track of the item B; the bold lines show the path taken by item D. Notice that since the bottom door of PE5 is locked PE11 will not receive the item D.

2.4.2 Completeness of the Itemset matching

We can provide some formal assurances that the SCAN mode algorithm can dictate the candidate itemset correctly.

Lemma 7. *During the matching, the items in the candidate itemset arriving later follow the path of the previous items in the same transaction.*

Proof A control signal initializes $IsLeaf$ and opens the bottom door before dictating a candidate itemset. In the horizontal direction, the items are always sent to the right sibling as discussed above. Therefore, if we can prove the items in the candidate itemset arriving later follow the path of the previous items in the vertical direction, the assertion is also true in the horizontal direction.

Next we show that the assertion is true in the vertical direction. Suppose the candidate itemset is $\{item_1, item_2, \cdots\}$. For an arbitrary path $\{PE_s, PE_{s+1}, \cdots, PE_t\}$ in the systolic tree, PE_{s+i} is the parent of PE_{s+i+1} and $item_{s+i}$ is the item stored in PE_{s+i}. If a PE is empty, no item is forwarded. We assume all PEs are non-empty. Consider the clock cycle t at which $item_1$ enters PE_s in Algorithm3.

1. If step (2) is triggered in PE_s, $item_1$ is equal to $item_s$ and $item_1$ is not forwarded in the vertical direction any more. Since $item_2 > item_1$ and $item_1 = item_s$, $item_2 > item_s$. Thus step (4) is triggered in PE_s in clock cycle $t + 1$. Consequently, $item_2$ is sent to the direction of PE_{s+1}.

2. If step (3) is triggered in PE_s, the bottom door is locked and $item_1$ is forwarded to the sibling. In clock cycle $t + 1$, either step (3) or step (4) is triggered. In both steps, $item_2$ is sent to the sibling. In step (4), $item_2$ is not forwarded to PE_{s+1} since the bottom door of PE_s is locked.

3. If step (4) is triggered in PE_s, $item_1$ is sent to PE_{s+1} and the sibling. Since $item_2 > item_1$ and $item_1 > item_s$, then $item_2 > item_s$. Thus step (4) is triggered in PE_s in clock cycle $t + 1$.

In all cases discussed above, $item_2$ follows the path of $item_1$. Similarly $item_2$ follows the path of $item_1$ in other PEs. In a similar way we can prove $item_{i+1}$ follows the path of $item_i$ in the candidate itemset. □

Lemma 8. *The first item of a candidate itemset is correctly located in the systolic tree.*

Proof Suppose the first item of the candidate itemset is $item_1$. Since $item_1$ is the first item in the candidate itemset, the bottom doors of all PEs are open. Consider the leftmost path $\{PE_1, PE_2, \cdots, PE_w\}$ which consists of the leftmost PEs of each level. $PE_i (1 \leq i \leq w)$ is the parent of PE_{i+1} and $item_i (1 \leq i \leq w)$ is the item stored in PE_i.

1. If the leftmost path contains $item_1$, there is a stored item (say $item_r$) equal to $item_1$. According to the SCAN mode algorithm and Lemma 4, step (4) is triggered in $PE_i(1 \leq i < r)$. Step (2) is triggered in PE_r. In both cases, $item_1$ is sent to the siblings and paths which potentially contain it. $PE_i(r < i \leq w)$ will not receive $item_1$. Thus any PE which is a descendant or siblings of $PE_i(r < i \leq w)$ will not contain $item_1$. This is because all of them are descendants of PE_r. According to the construction process of the systolic tree and Lemma 4, the items stored in them are larger than $item_1$.

2. If the leftmost path does not contain $item_1$, either step (3) or step (4) is triggered in $PE_i(1 \leq i \leq w)$. In step (4), $item_1$ is sent to all connected PEs. $item_1$ is not sent to the descendants in step (3). This is justified in Corollary 1.

A similar analysis can be applied to other paths. Therefore, $item_1$ will be sent to any PE which is likely to contain it. Therefore, the first item of a candidate itemset is correctly located in the systolic tree. □

Theorem 2. *A candidate itemset is dictated correctly using the SCAN mode algorithm.*

Proof Suppose the candidate itemset is $item_1, item_2, \cdots$. Based on Lemma 7 and Lemma 8, $item_1$ is correctly located in the systolic tree and other items follow the path of $item_1$. For an arbitrary path of which $item_1$ is located in a PE_{s1}, suppose $item_1$ enters PE_{s1} in clock cycle t. In clock cycle $t + 1$, $item_2$ enters PE_{s1}. Step (4) is triggered and $item_2$ is forwarded the leftmost child of PE_{s1}. Using a similar proof as in Lemma 8, $item_2$ is correctly located in the sub-tree rooted in PE_{s1}. Similarly, other items in the candidate itemset are correctly located in the systolic tree. Therefore, the candidate itemset is located correctly using the SCAN mode algorithm.

Also notice that the $IsLeaf$ flag is set when $item_1$ enters PE_{s1}. Since step (4) is triggered while $item_2$ entering PE_{s1}, the $IsLeaf$ flag is cleared. After all items in the candidate itemset are processed, only PEs whose stored items match the final item set $IsLeaf$ and serve as the reporting PEs. Therefore, the candidate itemset is dictated correctly using the SCAN mode algorithm. □

Algorithm 4 COUNT Mode Algorithm in Each PE

```
Input: $N_{Right}, N_{Bottom}$
  $Sent \leftarrow 0; N_{Child} \leftarrow N_{Right} + N_{Bottom}$
  if (Sent=0) and (IsLeaf=1) then
    $Sent \leftarrow 1;$
    forward $(N_{Myself} + N_{Child})$ to its parent direction
  else
    forward $N_{Child}$ to its parent direction
  end if
```

2.4.3 Candidate Itemset Count Computation

According to Property 1 of the systolic tree, there is only one path tracing back from any PE to the control PE since each PE has a unique parent. Once all items in a candidate itemset are sent to the systolic tree, a control signal signifying the COUNT mode is broadcast to the whole systolic tree. Looking back at Figure 2.3, we have shown that the first child's input interface is always connected to its parent while others accept input from the sibling in WRITE and SCAN mode. After a candidate frequent itemset is delivered into the systolic tree, the PEs report the support count of the candidate itemset to its unique parent. The PE which is not directly connected to its parent sends its count to the left sibling. The parent PE collects the support counts reported by the children PEs and sends them to its own parent direction. The COUNT mode algorithm in each PE is given in Algorithm 4, where the support counts are transferred to each PE's parent in a pipelined fashion. The inputs of the algorithm are two count values sent by its sibling and child respectively. The N_{Myself} variable is the number of the locally stored item. The $Sent$ flag is set when the local number has been reported to the parent. Only the PEs with the $IsLeaf$ flag set can report its local N_{Myself} value while other PEs deliver the count values sent by the child and sibling to their parents. The control PE adds up all count values and sends it to the output signal.

Since each PE has K degrees and the depth of the tree is W, the COUNT mode signal reaches the right-bottom PE after $W + (K-1) \times W = K \times W$ cycles. It takes another $K \times W$ cycles for the count values broadcasting from the right-bottom PE to the control PE. Notice that the number of clock cycles to dictate an itemset is $C - 1 + KW$ where C is the number of items in the dictated itemset. During the time of matching, the PEs can also perform count computation of the last itemset dictated. In this case, the control signal which indicates the SCAN mode for the current itemset to be dictated is

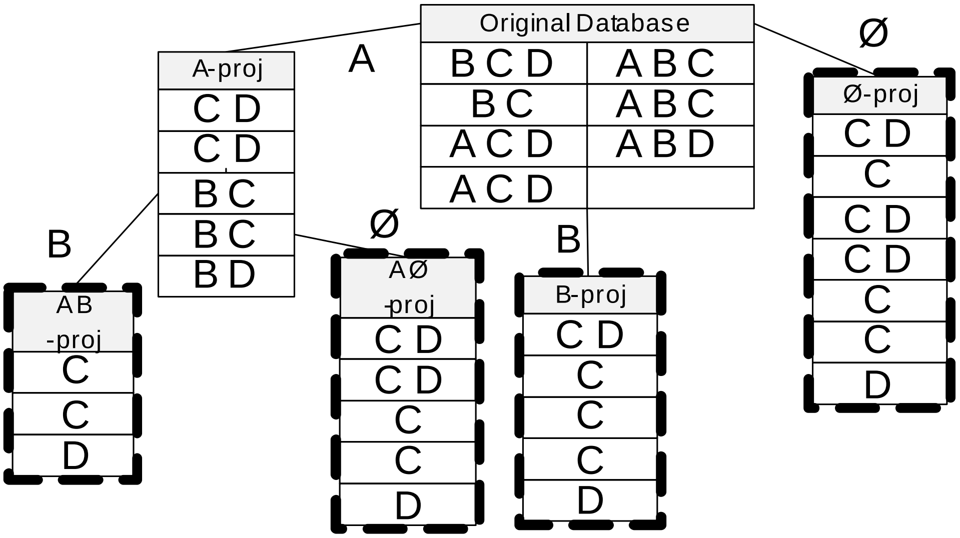


Figure 2.5 Projected Transactional Database

also served as the control signal which signifies the COUNT mode for the last itemset dictated. In other words, each PE both sends and receives data from its parent in each clock cycle. Since $C - 1 + KW$ is larger than KW, the runtime for COUNT mode is overlapped with the runtime of matching.

2.5 Tree Scaling by Database Projection

Both the FP-growth approach and our systolic tree approach use the tree structure to compress the representation of transactions in the database. The size of the tree is dependent on the characteristics of the database. A dense tree leads to a smaller tree size. In the case that memory or logic is not large enough to hold the whole tree, the database must be divided into multiple smaller databases with fewer frequent items. Without the technique of *database projection*, the brute-force candidate itemset matching will take an intolerable amount of time when the number of frequent items is large. Therefore, it is unreasonable to assume that a tree-based representation can fit in the available hardware resources (either memory or logic) for any arbitrary database. In this section, we will first introduce the concepts of database projection. Then we will propose a projection approach based on FP-growth algorithm.

2.5.1 Introduction to Database Projection

To use the systolic tree to mine frequent itemsets, the original database is projected into sub-databases. Each of the projected database has no more than $N = min(K, W)$ frequent items and is guaranteed to fit into the FPGAs. Let's illustrate the database projection with an example. Suppose the FPGA logic can at most hold a systolic tree with two frequent items. The database in Figure 2.1 has four frequent items and should be projected into sub-databases each of which has at most two frequent items. The frequent items are usually sorted in frequency-decreasing order which introduces a dense tree structure. For illustration purpose here, we arrange the frequent items in an alphabetic order, i.e., A, B, C, D. Starting at frequent item A, the set of transactions that contain A are collected as A-projected database. Since there are three frequent items B, C, D in the A-projected database, it should be further projected into two sub-databases. One is AB-projected database. The transactions in AB-projected database are obtained by removing B from those transactions containing B in A-projected database. The other is $A\phi$-projected database which is composed of those transactions in A-projected database by removing B. Since the transactions in both of them only contain C and D, the projection process terminates. Similarly, the B-projected database is generated by removing A, B from those transactions which contain B. Since the transactions in the B-projected database only contain C and D, the projection process terminates. Finally, a ϕ-projected database which contains only C and D are projected by removing A and B from all transactions in the original database. The projected databases which will be put into FPGAs are shown as dotted leaves in Figure 2.5. Notice the difference between A-projected database and $A\phi$-projected database. The former database contains all the transactions starting with A while the latter contains only those transactions starting with A but not including items B. The formal definition of database projection is shown below [73]:

Definition 3 (Projected Database). *Given a transactional database, DB,*

1. *Let a_i be a frequent item. The a_i-projected database for a_i is derived from DB by collecting all the transactions containing a_i and removing from them (1) infrequent items, (2) all frequent items before a_i in the $list\ of\ frequent\ items$, and (3) a_i itself.*

2. *Let a_i be a frequent item in α-projected database. Then the αa_j-projected database is derived from the α-projected database by collecting all entries containing a_j and removing from them (1)*

infrequent items, (2) all frequent items before a_j in the list of frequent items, (3) a_j itself.

The total size of the projected databases is much larger than the original database. Therefore, the projected databases are usually generated on the fly instead of storing them in the memory or disk.

Lemma 9. *Given a database with n frequent items, the number of projected sub-databases whose transaction only contain the last $N = min(K, W)$ frequent items is at most 2^{n-N} where $n \geq N$.*

Proof Suppose the n frequent items are ordered in arbitrary order as i_1, i_2, i_3, $\cdots$, i_n. The items in each transaction are also reordered according to the sequence of i_1, i_2, i_3, $\cdots$, i_n. Those transactions begin with item $\{i_{n-N+1}\}$,$\{i_{n-N+2}\}$, ..., or $\{i_n\}$ can be put into the ϕ-projected database. The ϕ-projected database starting with the last N frequent items contains no more than N frequent items in each transaction. The number of i_{n-N}-projected databases is 1. The items in each transaction of it are included in $\{i_{n-N}$,i_{n-N+1},..., $i_N\}$. The number of sub-databases whose transactions begin with i_{n-N-1} is 2. One is $i_{n-N-1}i_{n-N}$-projected database; the other is $i_{n-N-1}\phi$-projected database. The number of sub-databases starting with $i_j(1 \leq j \leq n-N)$ is 2^{n-N-j}. Therefore, the total number of sub-databases is $\sum_{j=1}^{j=n-N} 2^{n-N-j} = 2^{n-N}$. □

2.5.2 Algorithm for Database Projection

Observing lemma 9, we know that the process of projection encounters the combinatorial problem of the candidate generation which is resolved in the FP-growth algorithm. Inspired by this observation, we will explore how to perform database projection on the fly during the running of FP-growth algorithm. A new field called $subroot$ is created in each node. The $subroot$ is a list of pointers each of which points to a subset of the FP-tree. The algorithm to create $subroot$ for each node is illustrated in Algorithm 5, where γ is a set of N frequent items which will be put into the projected databases. The selection of the N items from the n frequent items is critical to the performance of the system. Recall that our goal is to take advantage of the hardware systolic tree and use it to do the cumbersome mining work. Hence, the first N most frequent items should be put into γ. For example, the items $\{C, A\}$ or $\{C, B\}$ instead of $\{C, D\}$ should be put into the projected databases in Figure 2.5. Therefore, the sequence of the frequent items in our header table is different from the original FP-growth algorithm. The first N frequent items are moved the end of the header table before projecting the database as shown in Figure 2.6. In this

Algorithm 5 Subroot Creation Algorithm

```
Input: The FP-tree of DB
Output: The subroot in each node
  Set $subroot$ to $null$ for each node
  for $i = n - N$ to 1 do
    for Each node $n^p$ following the nodelink of the $i^{th}$ item do
      for Each Child Pointer $T_j^p$ of $n^p$ do
        if the node $n_j^p$ pointed by $T_j^p$ contains an item $\in \gamma$ then
          $n^p.subroot \leftarrow n^p.subroot \cup T_j^p$
        else
          $n^p.subroot \leftarrow n^p.subroot \cup n_j^p.subroot$
        end if
      end for
    end for
  end for
  the $subroot$ of the root is the $subroot$ unions of its children
```

way, the size of the FP-tree without the first N frequent items is much smaller than the original tree and the number of projected databases is greatly reduced.

The FP-trees pointed by $subroot$ hold those transactions which contains only the last N frequent items in the new header table. To facilitate the understanding of projection process, we first go through the example in Figure 2.2. Suppose each transaction in the projected database only contains the last two frequent items, i.e., $N = 2, n = 4$. The dashed lines in Figure 2.6 are the $subroot$ pointers in each node. One can see that the $subroot$ of the parent node is a superset of the $subroot$ of its children. The $subroot$ algorithm starts from the nodelink of D. The first node (D:2) on the left branch has two children (C:1) and (A:1). Hence, its $subroot$ is composed of two pointers pointing to the two children. The second node (D:2) on the right branch which has only one child has only one pointer in its $subroot$. Following the nodelink of B in the header table, the node (B:5) has two children. One is (D:2) which does not contain an item in β. According to Algorithm 5, the $subroot$ of (D:2) is merged into the $subroot$ of (B:5). The other children (C:3) contains an item C which is in γ. Thus, the pointer which points to node (C:3) is merged into the $subroot$ of (B:5). Similarly, the $subroot$ of the root node is a collection of pointers pointing to all the projected transactions which contains only the last N frequent items in the header table.

Before discussing database projection based on FP-growth, we first present the FP-growth algorithm from [73] as shown in Algorithm 6. Those lines marked with stars are not in the original FP-growth

Algorithm 6 Database Projection Based on FP-growth Algorithm

```
Input: An FP-tree of a database DB; A support threshold $\xi$
Output: The complete set of frequent patterns
  Call FP-growth(FP-tree, null)
  *Call subroot creation algorithm
  *generate projected database $Proj_{\phi}$ for global FP-tree
  **generate freq_pattern_set$(Proj_{\phi})$)

  Procedure FP-growth$(Tree, \alpha)$
  if $Tree$ contains a single prefix path then
    let $P$ be the single prefix path part of $Tree$
    let $Q$ be the multipath part with the top branching node
    replaced by a null root
    for each combination $\beta$ of the nodes in the path $P$ do
      generate pattern $\beta \cup \alpha$ into freq_pattern_set(P)
      **generate patterns $(\beta \cup \alpha)\times$freq_pattern_set$(Proj_{\alpha})$
      **into freq_pattern_set(R)
    end for
  else
    let $Q$ be Tree
  end if
  for each item $a_i$ in $Q$ do
    generate patten $\beta = a_i \cup \alpha$ with support=$a_i$.support
    construct $\beta$'s conditional pattern-base
    construct $\beta$'s conditional FP-tree $Tree_{\beta}$
    *generate $subroot$s for $Tree_{\beta}$
    *generate $\beta$'s projected database $Proj_{\beta}$
    **generate patterns $\beta\times$freq_pattern_set($Proj_{\beta}$)
    if $Tree_{\beta} \neq \phi$ then
      call FP-growth$(Tree_{\beta}, \beta)$
    end if
    let freq_pattern_set(Q) be the set of patterns so generated
  end for
  return freq_pattern_set(R)$\cup$freq_pattern_set(P)$\cup$
  freq_pattern_set(Q)$\cup$freq_pattern_set(P)$\times$freq_pattern_set(Q)
```

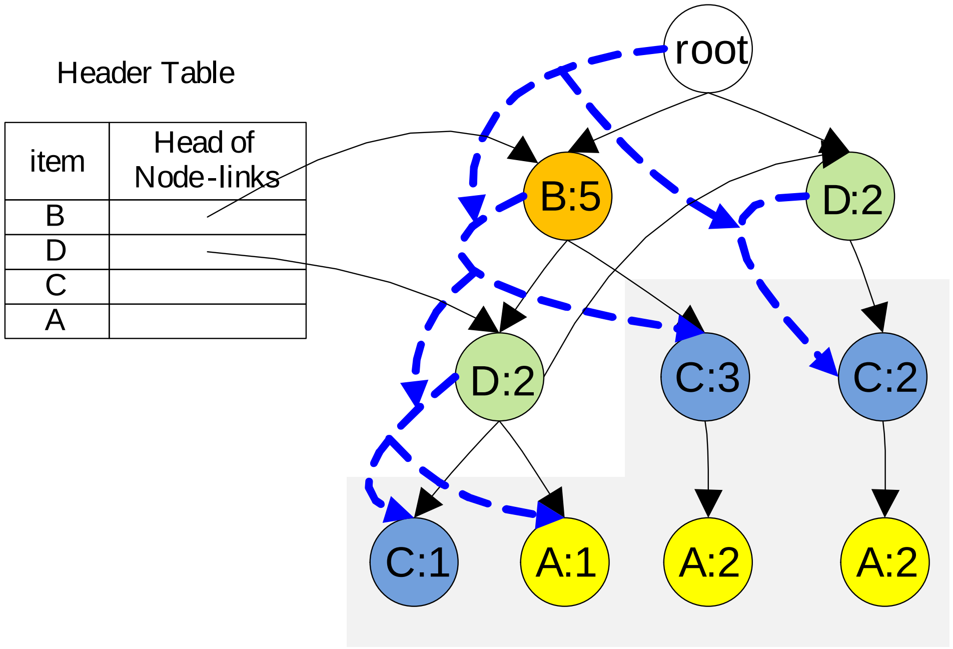
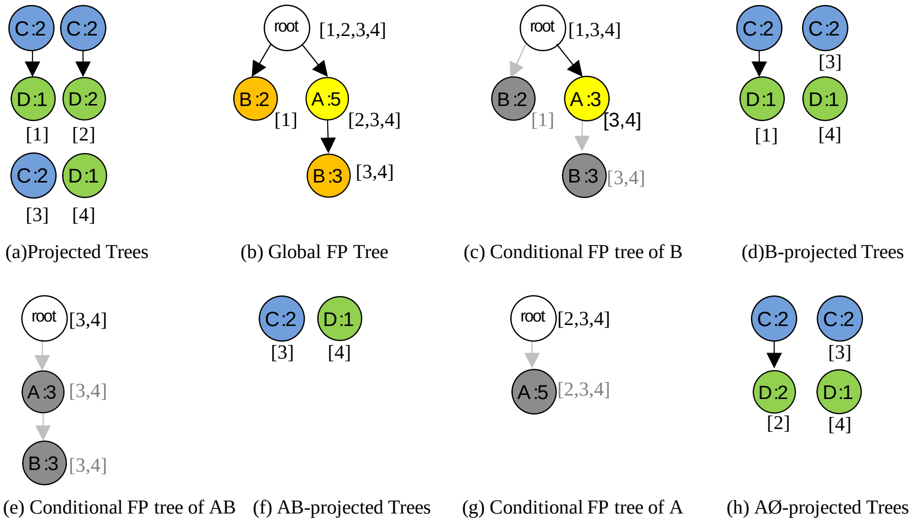


Figure 2.6 Subroots

algorithm and will be discussed later. According to this algorithm, the conditional trees generated from Figure 2.2 is shown in Figure 2.7. For illustration purpose, the items are arranged in alphabetic order instead of the order shown in Figure 2.6. To use FP-growth algorithm to generate projected databases, the mining process starts from the $(n - N)^{th}$ item in the header table. The $subroots$ construction of conditional FP-trees is shown in Algorithm 5. In Figure 2.7(a), each sub-tree which will be put into the projected databases has a label in square bracket. The *subroots* generated for the global FP-tree are shown in Figure 2.7(b). The $subroot$ creation algorithm for conditional trees is a little different from Algorithm 5. Instead of generating $subroot$ for each node in the conditional tree, only the $subroot$ in the root and leaves are computed for each conditional tree. This will greatly reduce the runtime of the $subroot$ creation algorithm. Those trees pointed by *subroots* are called *projected trees*. The projected trees of each conditional tree are shown separately in Figure 2.7(d)(f)(h). For example, the *B*-projected database is (C:2,D:1),(C:2),(D:1); and the AB-projected database is (C:2), (D:1) which is a subset of *B*-projected database. Theorem 3 is the key for the FP-growth based database projection. The database projection algorithm is shown in Algorithm 6 with those lines marked with one star added to the original FP-growth algorithm. One can find out that the projected databases in Figure 2.7 are the same those



(a)Projected Trees (b) Global FP Tree (c) Conditional FP tree of B (d)B-projected Trees

(e) Conditional FP tree of AB (f) AB-projected Trees (g) Conditional FP tree of A (h) AØ-projected Trees

Figure 2.7 Database Projection Based on FP-growth

in Figure 2.5. Note that the ϕ-proj database in Figure 2.5 is corresponding to the projected tree of the global FP-tree in Figure 2.7(a).

Theorem 3. *For each frequent itemset α, the α-projected database can be derived from the projected trees of the α conditional FP-tree.*

Proof Let T_i^α be an arbitrary transaction in the *projected trees* of the α's conditional FP-tree. Based on the *subroot* and FP-tree construction algorithms, there must be a transaction in the original database containing both T_i^α and α. Based on the completeness of FP-tree [73], any transaction containing both T_i^α and α can be found in α's conditional FP-tree and its projected trees. □

2.5.3 Frequent Itemset Generation with Database Projection

For each frequent itemset β, the frequent itemsets generated from the projected database of β(β-proj, or $Proj_\beta$) are combined with β. This process is denoted as $\beta \times$*frequent_pattern_set*$(Proj_\beta)$ in Algorithm 6. The optimization of the single prefix-path FP-tree is the same as the approach proposed in [73]. However, the database projection needs not be performed for the single prefix path *P* since it has been generated into *frequent_pattern_set*$(Proj_\alpha)$. The frequent itemset generated from the projected database of the tree containing a single prefix path should be combined with each pattern in that single prefix path separately(denoted as *freq_pattern_set(R)* in Algorithm 6). This is because the top

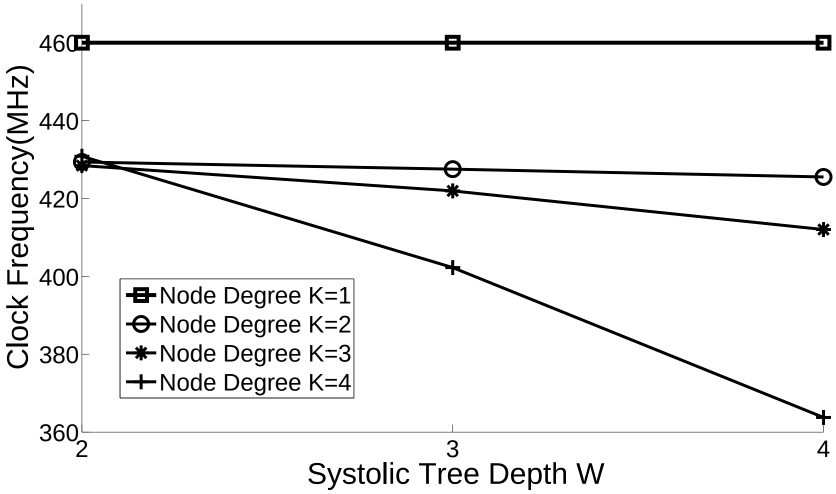


Figure 2.8   Systolic Tree Clock Frequency

branching node of Q is replaced by a null root. Each frequent pattern β generated in tree Q is combined with the frequent patterns in $Proj_\beta$. *freq_pattern_set(Q)* is combined with *freq_pattern_set(P)* where the occurrence frequency of each combined frequent itemset is the minimum support among those items. Those lines marked with double stars deal with frequent itemset generation from projected database. The Algorithm 6 provides a way to mine the frequent itemsets in parallel. In section 2.6, we will demonstrate how to use the newly proposed algorithm to fully utilize the power of reconfigurable platform where software and hardware mine the frequent itemsets concurrently.

2.6 Experimental Evaluation

In this section, we will demonstate how to map the hardware/software algorithms to a reconfigurable platform. We also compare our experimental results with the original FP-growth algorithm.

2.6.1 Area Requirements and Performance of Hardware

From Property 2 of the systolic tree, we know that there are $\frac{K^{W+1}-1}{K-1} = O(K^W)$ nodes in a systolic tree. The depth of the tree has exponential influence on the total size. In our VHDL implementation, different sizes of trees are generated automatically by specifying the depth and degree parameters. We placed and routed various configurations of this implementation using Xilinx ISE 9.1.03i. The targeted

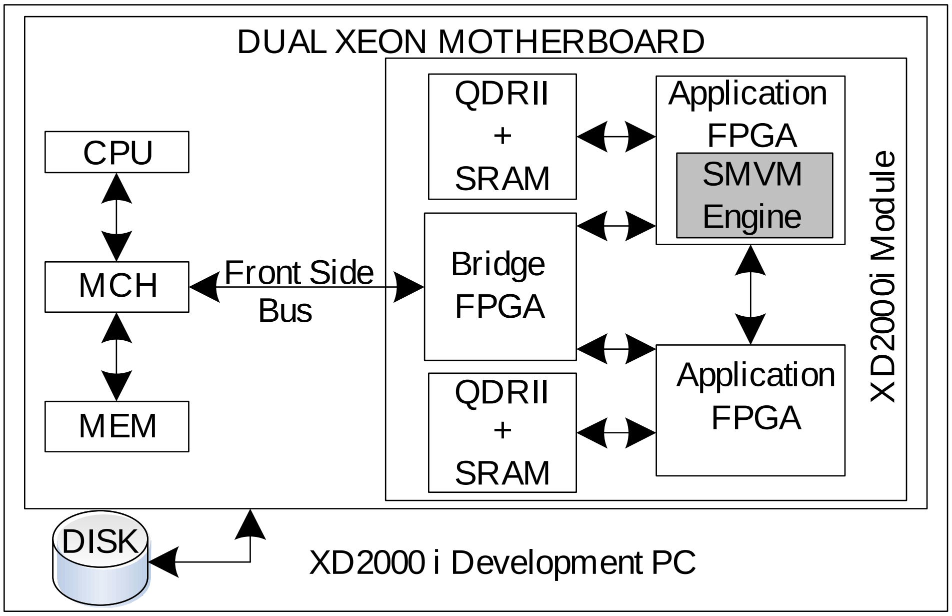


Figure 2.9 Simplified XtremeData XD2000i Architecture

device is a Xilinx Virtex-5 XC5VLX330 FPGA with package ff1760 and -2 speed grade. From the place and route report, each node requires 50 LUTs. Approximately 25% of the available slices are used when both K and W are four. When both K and W are five, the required slices is more than 100%. Therefore, our Xilinx Virtex-5 FPGA can at most accommodate a systolic tree of size four. The clock frequency of post placement and routing for different combination values of K and W is shown in Figure 2.8. As can be seen in this figure, the clock frequency drops gradually when the node degree or the tree depth increases due to the increasing number of interfaces and interconnections between PEs and the increasing routing delay for a larger circuit.

2.6.2 Performance Comparison

For performing a fair comparison, we selected as our target environment the XtremeData XD2000i, which is a complete platform to explore the benefits of FPGA coprocessing with traditional microprocessor computing systems [8]. XtremeData's XD2000i development system comprises of a Dual XEON motherboard with one Intel XEON processor and two Stratix III EP3SE260 FPGAs, running the CentOS Linux operating system as shown in Figure 2.9. The XEON processor has four cores running at 1.6GHz and communicates with the FPGAs and the 4GB system memory through Intel Memory Con-

Table 2.1 Benchmark Information

Benchmark	**Items**	**Num. Trans.**	**Max Trans.**	**Min Trans.**	**Total Items**
chess	75	3196	37	37	118252
BMS-WebView-2	3340	77512	161	1	358278
connect	129	67557	43	43	2904951
BMS-POS	1657	515597	164	1	3367020
pumsb	7117	49064	74	74	3629404
kosarak	41270	990002	2498	1	8019015

troller Hub (MCH). The CPU prepares a data buffer of a known size in system memory used for sending and receiving data from FPGAs. The sending request may return immediately so that additional CPU work may be performed before submitting the receiving request.

In our simulated environment, the systolic tree of size four is integrated with other hardware IP components provided by XtremeData in one of the FPGAs. The Intel CPU executes the software Algorithm 6 which is implemented in C++. For each frequent itemset β, a projected tree is generated as a set of pointers pointing to the paths in the original FP-tree. The algorithm reads the transactions pointed by those pointers and sends them to the systolic tree in FPGA. The sending operation returns immediately. The FPGA hardware receives the transactions and creates the systolic tree using Algorithm 2. Thereafter, the candidate itemsets are generated and dictated by hardware without software intervention in a pipelined style. Algorithms 3 and 4 are executed in each node of the systolic tree. After all frequent itemsets are collected, they are sent back to the software. After receiving the frequent itemsets from the hardware, the software combines them with β.

During this process, the original database is projected sequentially which is similar to *partition projection* mentioned in [73]. The experimental results presented below are based on this sequential mode. Reconfigurable development systems with multiple FPGAs integrated in the same board would not benefit from this model since the running time of FPGAs in this case is smaller than the running time to generate a projected database. *Parallel projection*, which scans each transaction in the original database and projects it into multiple projected databases in parallel can fully utilize the multi-FPGA system. Designing an efficient parallel projection software algorithm and applying it in a parallel FPGA implmentation is a planned avenue of future work. We can expect that the parallel FPGA implementation will much faster than the sequential model used in this work.

The experiments are performed on real datasets described in [18] and from the KDD Cup 2000

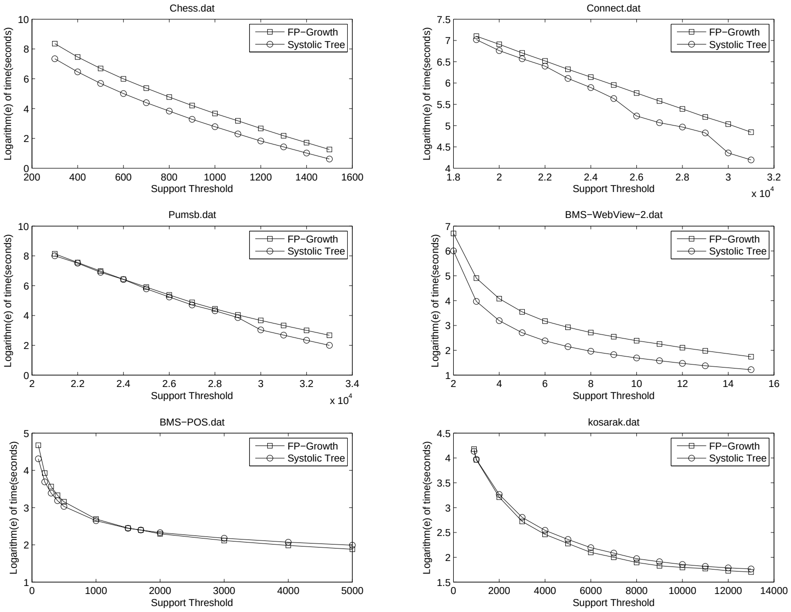


Figure 2.10 Performance Comparison with FP-growth

data [86]. Table 2.1 characterizes the datasets in terms of the number of distinct items and transactions, the maximum and minimum size of the transactions. The transactions in chess/connect/pumsb have the same size. The sizes of the transactions in other benchmarks vary from several items to thousands of items. The performance measure was the execution time of the systems on the benchmarks with different support threshold. The execution time only includes the time for disk reading, memory I/O, but excludes the disk writing time. The original software FP-growth algorithm which is implemented in C++ is from [62]. In order to have a fair comparison, the software FP-growth algorithm also runs on our target XtremeData 2000i platform without using the FPGA hardware.

The performance result of each benchmark is listed in Figure 2.10. The systolic tree algorithm is almost two times faster than FP-growth for the chess and BMS-WebView-2 datasets. For the kosarak dataset, the mining time of the systolic tree is larger than FP-growth. The mining time the FP-growth grows larger than the systolic tree algorithm with the decrease of support threshold in BMS-POS. The advantages of the systolic tree over FP-growth becomes dramatic with long frequent patterns, which is challenging to the algorithms that mining the complete set of frequent patterns.

In retrospect, the systolic tree algorithm removes the most frequent items and mines them in the pro-

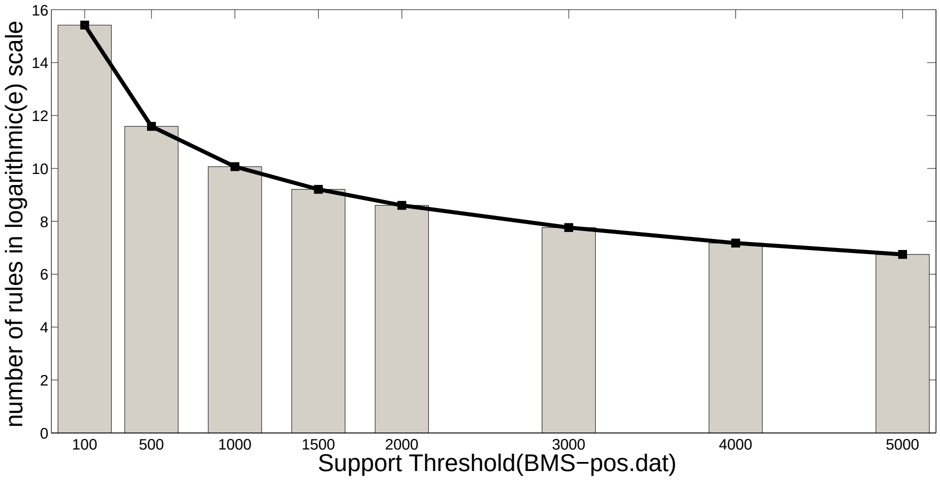


Figure 2.11 Number of Rules Mined in Hardware

jected databases. Compared with the original FP-growth algorithm, the systolic tree algorithm reduces the runtime by mining the dense part of the FP-tree in hardware and the sparse part in software simultaneously. However, it introduces the overhead of database projection. If the overhead is not amortized by the runtime reduction, the systolic tree algorithm is slower than the original FP-growth algorithm. This is illustrated in the kosarak benchmark. One can expect that the mining time will decrease with more frequent patterns mined in hardware when the size of the systolic tree is fixed. In Figure 2.11, the value of the y axis is the number of frequent patterns mined in hardware which includes the patterns generated from the hardware directly and the patterns combined with those generated from the software as shown in Algorithm 6. The number grows exponentially with the decrease of the support threshold. Even though the projection overhead also increases with the decrease of the support threshold, the number of patterns in hardware goes up sharply when the support threshold is less than 1000. This explains why the systolic tree is faster than FP-growth for BMS-POS.dat in Figure 2.10 when the support threshold is less than 1000.

2.7 Contribution

A new approach to mine the frequent itemset using systolic tree is proposed. The correctness of the operations in systolic tree is formally proved. The area requirement and performance of the systolic tree is analyzed. Due to the limited size of the systolic tree, a transactional database must be projected into smaller ones each of which can be mined in hardware efficiently. A high performance projection algorithm which fully utilizes the advantage of FP-growth is proposed and implemented. It reduces the mining time by partitioning the tree into dense and sparse parts and sending the dense tree to the hardware. The algorithm was implemented on an XtremeData XD2000i high performance reconfigurable platform. Based on the experimental results on several benchmarks, the mining speed of the systolic tree was several times faster than the FP-tree for long frequent patterns. Improving the mining efficiency on sparse patterns will be included in our future work.

CHAPTER 3. FLOATING-POINT ACCUMULATOR

Floating-point accumulator is an important component in sparse-matrix vector multiplication which is discussed in chapter 4. In this chapter, we will propose and evaluate an FPGA-based high performance floating-point accumulator. Compared to previous work, our accumulator uses a fixed size circuit, and can reduce an arbitrary number of input sets of varying sizes without requiring prior knowledge of the bounds of summands. In this chapter, we describe how the adder accumulator operator can be heavily pipelined to achieve a high clock speed when mapped to FPGA technology, while still maintaining the original input ordering. Our experimental results show that our accumulator design is very competitive with previous efforts in terms of FPGA resource usage and clock frequency, making it an ideal building block for large-scale sparse matrix computations as implemented in FPGA-based high performance computing systems.

3.1 Introduction

Floating-point accumulators (and vector reduction operators in general) form the basis of many scientific computing applications, and can be found in such diverse fields as weather forecasting, nuclear physics simulations, and image processing [105]. Given an input vector S, an accumulator performs a simple summation of the individual elements:

$$S = \sum s_i \tag{3.1}$$

While this is a trivial computation to describe mathematically, the need for fast floating-point accumulators (along with other operators) led in part to the emergence of vector processing in the 1980s. Accumulation is so commonplace in high performance computing applications that it has been described by some as the “fifth floating-point operation” [90].

One common computing scenario that requires a high performance accumulator is in Sparse Matrix-Vector Multiplication (SMVM) [159]. For example, Google's PageRank eigenvector problem is arguably the world's largest matrix computation (there were approximately 25 billion pages on the Internet in 2008) [103]. The PageRank algorithm is dominated by SMVM computations given that the Google matrix is very large (billions of rows and columns) and very sparse. Clearly, the performance of SMVM and by extension, floating-point accumulation has real-world impact.

Special care must be taken when designing FPGA-based accelerators for applications that make use of floating-point accumulators. The aforementioned SMVM operation calculates $Y = AX$, where $y_i = \sum_{j=1}^{n} a_{ij}x_j$ must be computed efficiently. A simple circuit to compute y_i is shown in Figure 3.1. In the case of a very large n (as is the case for Google PageRank), the elements in one row have to be partitioned into $\lceil \frac{n}{m} \rceil$ blocks. One block is sent to the circuit in each clock cycle. The partial sum $y_i^r = \sum_{j=(r-1)m+1}^{j=rm} y_{ij}$ produced in each clock cycle is accumulated near the output of the circuit where $y_i = \sum_{r=1}^{\lceil n/m \rceil} y_i^r$. The circuit module performing equation (3.1) is called the Adder Accumulator (AAC) or alternatively reduction circuit in [171, 169, 170]. The specific design we propose in this chapter is referred to as a Floating-point Adder Accumulator (FAAC), to distinguish it from other AAC circuits that use fixed-point arithmetic.

The performance of any FPGA design is highly dependent on the clock frequency at which the circuit can run without timing constraint violations after placement and routing. The clock frequency in turn is determined by the delay of the critical path in the circuit. Adding two values in one clock cycle, as shown in Figure 3.2(a), will increase the critical path delay thus leading to a lower clock frequency. Such a single-cycle FAAC design would severely reduce the performance of the whole FPGA system. Consequently, any high performance FPGA-based accumulator must partition the floating-point addition across multiple clock cycles in order to improve the resulting clock frequency.

Also, as highlighted by [49], FPGA I/O capacity can often limit the performance of the system. Designing a scalable and pipelined stall-free accumulator to fully utilize the I/O bandwidth and computing resources is critical to achieve a high performance goal. Designing a floating-point accumulator with pipeline stalls is trivial. One straightforward way is to use a standard k-stage floating-point adder as shown in Figure 3.2(b). There are k clock cycles stalls between two additions. In other words, the sum of the first addition to be added with a third value will not come out of the adder until k clock cycles

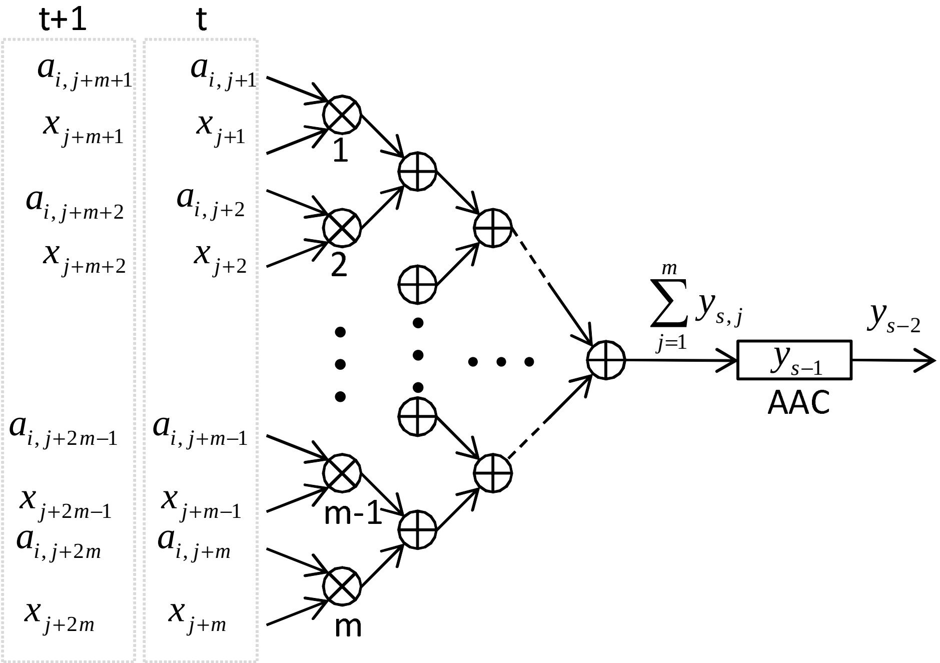
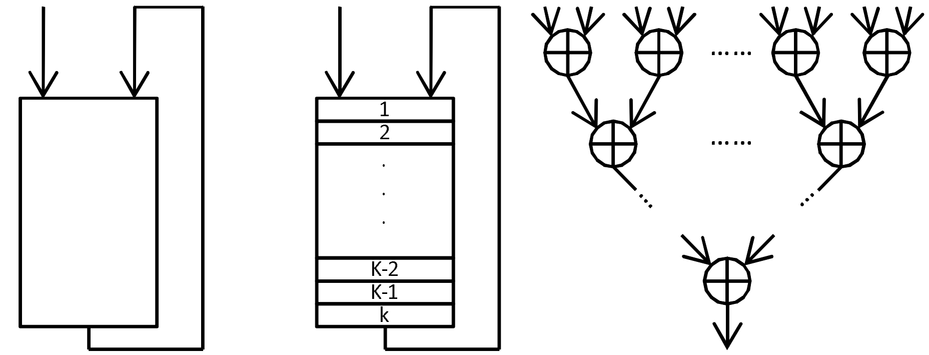


Figure 3.1 Matrix-vector Multiplication Example

after the first two values entering the pipelines.

Removing these pipeline stalls is a challenging task since the pipeline input fed by the later pipeline output introduces feedback. That is, a new input data cannot be processed until the previous sum comes out from the end of the pipeline. An intuitive approach in designing a deeply pipelined and stall-free floating-point accumulator is the adder tree as shown in Figure 3.2(c). An adder tree with $n-1$ adders can be used to accumulate n floating-point values. The first row in the tree has $\lceil \frac{n}{2} \rceil$ adders; the i^{th} row has $\lceil \frac{n}{2^i} \rceil$ adders; the last row in the tree has only one adder whose output is the accumulated result. However, this approach is infeasible when n is large or the n input values cannot arrive in the same clock cycle. The resource utilization on a single FPGA device will be prohibitively high when n is large. Also, the number of pipeline stages to compute a sum will be intolerably large since the depth of the adder tree is $\lceil log_2^n \rceil \times k$, where k is the number of pipeline stages in an adder.

Our FAAC design as described in this chapter accumulates positive and negative numbers separately so that the accumulation adder does only additions of numbers of the same sign. Saving leading zero count and cancellation shift reduces the latency of the adder. A subtractor which computes the difference of the two partial sub-sums is not in the accumulator loop. By using the log-sum technique two standard



(a) One Clock Cycle (b) K Clock Cycles (c) Adder Tree

Figure 3.2 Impractical FAAC Architectures

adders reduce intermediate sub-sums. The summation order is different from the order of the inputs, the impact of this feature on result accuracy is discussed in Section 3.6. Preliminary results show that our heavily pipelined (48 stages) FAAC design can obtain high clock frequencies (and subsequently high performance) while using a relatively small amount of FPGA resources.

3.2 Problem Formulation

The input of the FAAC is a sequence of floating point numbers which may be positive or negative. The numbers belong to different groups. The numbers in the same group are fed into the FAAC consecutively. At each clock cycle, only one number enters the pipelines. In a formal way:

Suppose the input data stream is $\{a_{1,1}, a_{1,2}, ..., a_{1,n_1}\}, \{a_{2,1}, a_{2,2}, ..., a_{2,n_2}\}, \cdots, \{a_{i,1}, a_{i,2}, ..., a_{i,n_i}\}, \cdots$, where $a_{i,j}$ is the j^{th} data in the i^{th} group. The numbers in the bracket belong to the same group. If $a_{s,j}$ arrives earlier than $a_{t,j}$, all the data in the s^{th} group must arrive earlier than the data in the t^{th} group. The problem is to sum up all the data in a group into a single value such that $a_i = \sum_{j=1}^{n_i} a_{i,j}, i = 1, 2, ...$ In the extreme case, each group has only one data value and the accumulator behaves as a queue.

3.3 Previous Work

The design of floating-point accumulators can be traced back to the 1980s with the emergence of pipelined vector machines [85, 105]. An adder accumulator adopting the architecture in [105] is well-suited for accumulating one input group. However, the buffer will overflow for multiple input groups. In [169], the reduction circuit uses only one adder with $\Theta(log(n))$ space and $\Theta(log(n))$ latency. However, the design can not handle the accumulation correctly when the number of inputs in a group is not a power of 2. The reduction circuit in [171] uses only one floating-point adder but needs two buffers of size k^2 where k is the number of pipeline stages of the adder. However, the latency of reducing one input group depends on the sizes of subsequent groups and the latency of reducing may be unacceptably large if the size of the subsequent groups is very large. Also, the output order of the sums may not be the same as the input order of the input groups. The design trade-offs and different approaches for accumulators are studied in [170]. The accumulators presented in both [38] and [75] require the user to specify bounds on the values to be accumulated beforehand.

3.4 Design Principle

To accumulate n numbers, one straightforward observation is that the number of partial sums is at most equal to the number of stages of the adder. At most k adders will be used where k is the number of pipeline stages; one is to produce the partial sum, the other $k-1$ adders form an adder tree to sum up the $k-1$ partial sums. Designing an accumulator using this approach is prohibitive in a single FPGA device when k is large. Since the data is sent to the accumulator in a serial fashion, we use the log-sum technique [85] to reduce the partial sums. To reduce k partial sums produced by the first adder, the pipeline needs only another $\lceil log_2^k \rceil$ adders and is totally independent of the intrinsic number of stages in those adders. Therefore, the number of stages in the first adder is critical and determines the number of adders to be used and the delay of the whole accumulator. Figure 3.3 shows the two cases for $k = 4$ and $k = 8$. Notice that the number of adders is not determined by the value of p.

The relationship between clock frequency and pipeline stages can be formulated as $Frequency \propto (Pipeline\ Stages/Total\ Delay)$. To maintain a high clock frequency while using fewer pipeline stages, we have to reduce the total delay of the adder. We observe that the floating point adder which only performs addition has a much lower delay than the adder which performs both addition and subtraction. For example, the former does not perform the mantissa comparison, the leading-zero counting and has fewer exception signalings and barrel shifters. Hence, we reduce the total delay by requiring

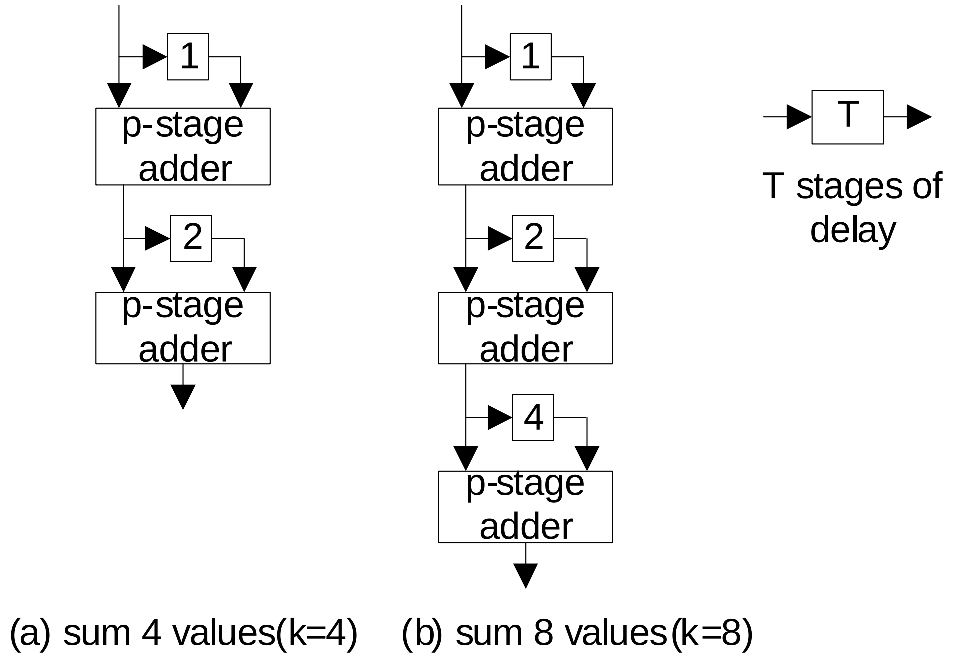


Figure 3.3 Log-sum Technique

that the first adder only performs addition.

The overview of our architecture is shown in Figure 3.4. The first adder performs addition on values of the same sign only. If the sign of the input is different from the sign of the output partial sum, the partial sum is pushed into the stack with the same sign; otherwise, the partial sum acts as one of the input. The 4-stage adder accumulates any number of sequence values into at most 4-pairs of partial sums. The subtractor performing only subtraction produces at most 4 partial sums for an arbitrary number of values. Using the log-sum technique shown in Figure 3.3, another two standard adders are used to reduce the 4 intermediate partial sums. As shown in Figure 3.4, the first adder which only performs addition has $k = 4$ pipeline stages to produce the partial sums. In this chapter, we refer to these pipeline stages as $bands$. The increased number of pipeline stages will lead to a higher clock frequency. For example, an 8-band accumulator will typically have a higher frequency than a 4-band accumulator.

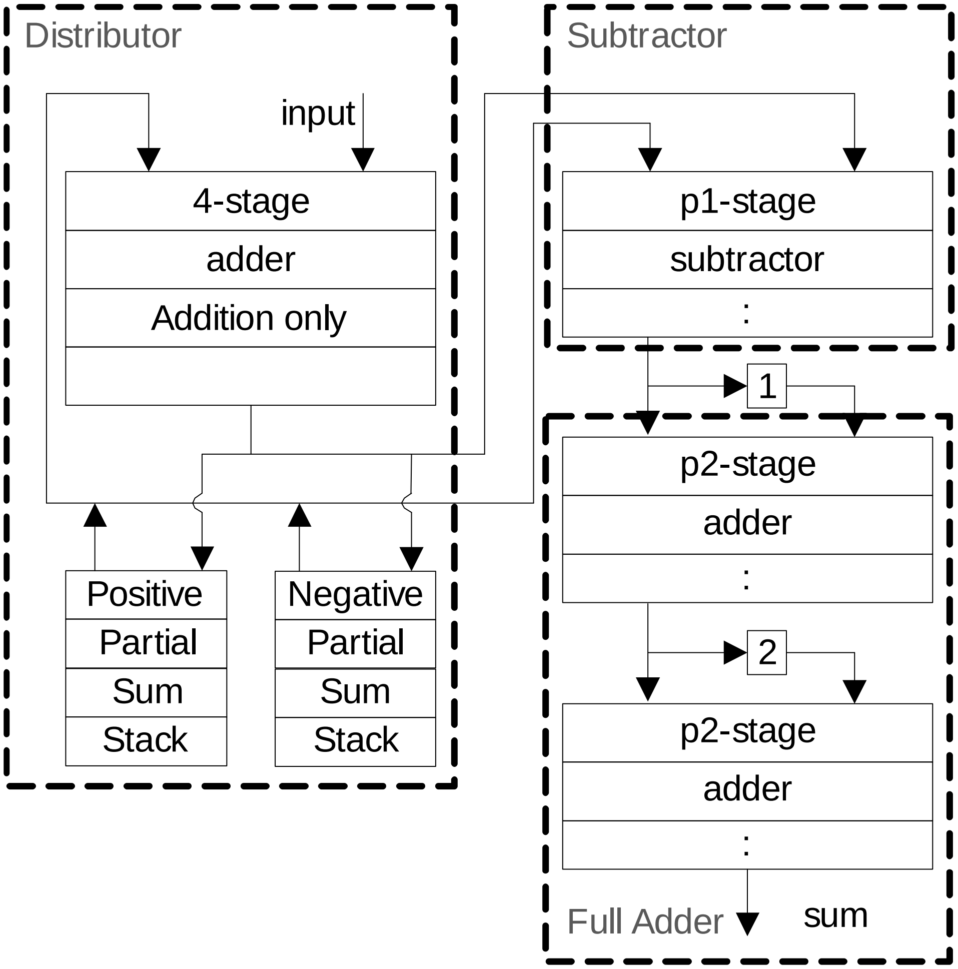


Figure 3.4 Accumulator Architecture Overview

3.5 Architecture

Details of our proposed FAAC architecture are shown in Figure 3.5. It is mainly composed of three modules: the distributor, the subtractor and the full adder. The input data is sent to the distributor whose outputs are connected to the inputs of the subtractor. The outputs of the subtractor are in turn tied to the inputs of the full adder. The outputs of the accumulator are hard-wired to the outputs of the last full adder.

3.5.1 Distributor

The distributor is composed of a 4-stage adder, two stacks, and some control logic. One stack stores the positive data while the other stores the negative data. In each clock cycle, the incoming data is added

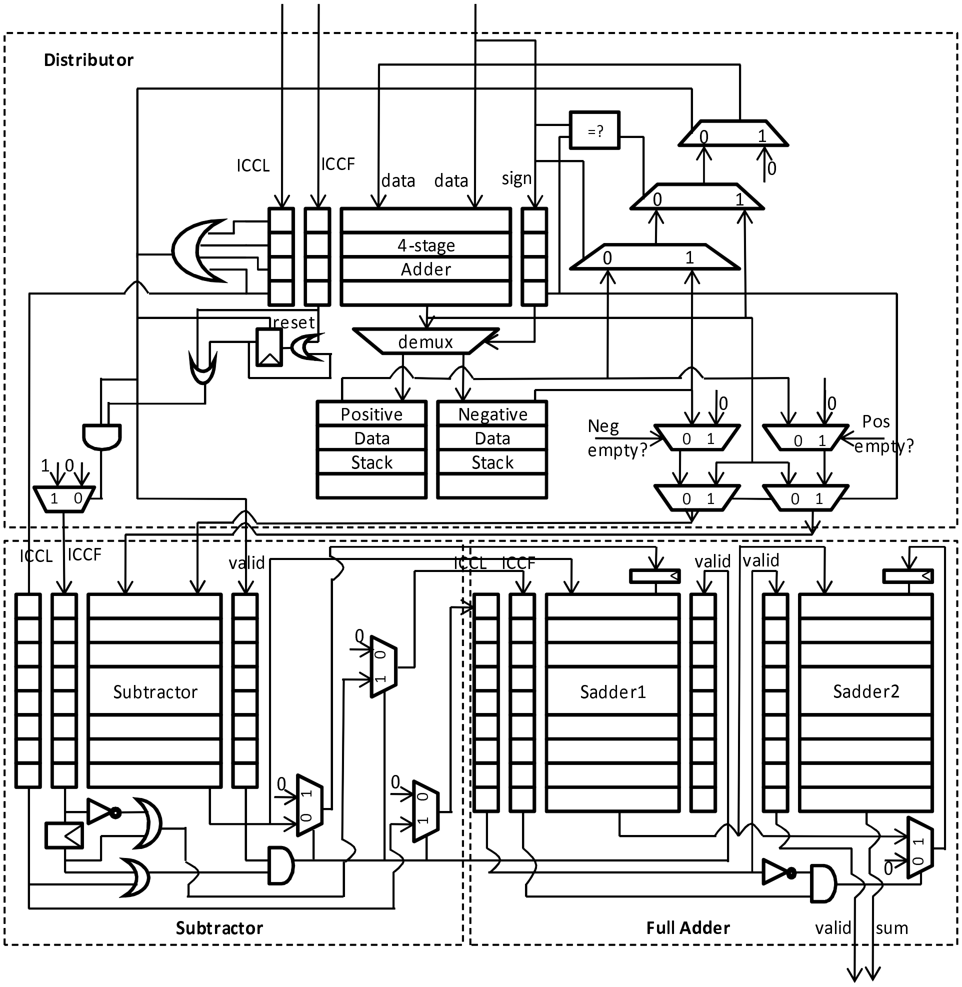


Figure 3.5   Architectural Details of a 4-band FAAC

with zero, output of the adder or the data on the top of the stack with the same sign. Since the number of intermediate results for the adder is at most k, the depth of the stack is equal to k. Each input data has two auxiliary bits called the Input Control Codes (ICCs). The ICCF bit is equal to 1 if the input data is the first incoming number in a new group; otherwise, it is equal to 0. The ICCL bit is equal to 1 if the incoming data is the last number in a group; otherwise, it is equal to 0. If a group has only one data value, the two bits are both equal to 1. Two queues with the same depth as the 4-stage adder serve as FIFO buffers for the ICCs. Another queue $sign$ is for the sign bit of the input data since the signs of the two inputs are the same. The control logic is used to schedule the inputs of the 4-stage adder and the outputs of the distributor.

The input scheduling algorithm is shown in Algorithm 7. $input(in)$ is the incoming data; $input(reg)$

Algorithm 7 Input Scheduling Algorithm of Distributor

```
if last=0 then
    if $sign(3)$=$sign(in)$ then
        $input(reg) \leftarrow sum$;
    else
        push $sum$ into $stack(sign(3))$;
        if $stack(sign(in))$ is empty then
            $input(reg) \leftarrow 0$;
        else
            $input(reg) \leftarrow stack(sign(in))$;
        end if
    end if
else
    $input(reg) \leftarrow 0$;
end if
```

is the other input of the 4-stage adder. $last$ is the logic OR result of the ICCL queue elements. sum is the output of the 4-stage adder. $last = 0$ happens only when the current group has more than four numbers; $last = 1$ happens only when the last number of the current group is in the pipelines. In the latter case, the new incoming data should be added with zero and the output of the adder should not be added with the new incoming data, but be sent out as output as shown in Alg 8. This is because the incoming data and the output of the 4-stage adder belong to different groups. In the former case, the newly generated partial sum is added with the incoming data if they have the same sign; otherwise, it is pushed onto the stack and the new incoming data is added with a number popped off of the stack with the same sign. In the case that the stacks are empty or the incoming data is one of the first four numbers in a group, zero is added with the incoming data.

The distributor has two data outputs in each clock cycle; one is positive pos, the other is negative neg. One of the outputs must come from the partial sum, the other is popped by the stack with the opposite sign or zero when that stack is empty. One of the outputs must be zero when the group being accumulated has less than five numbers. As shown in Figure 3.5, we use multiplexers to express the if statements in the algorithms.

To run the algorithms correctly, we must assure that both stacks are empty when the last partial sum of a group comes out of the 4-stage adder. Therefore, the first four partial sums of a group are equal

Algorithm 8 Output Scheduling Algorithm of Distributor

```
if $last$=1 then
   if $sign(3)$=+ then
      $pos \leftarrow sum$;
      $neg \leftarrow$ pop $stack(-)$;
   else
      $neg \leftarrow sum$;
      $pos \leftarrow$ pop $stack(+)$;
   end if
end if
```

to the numbers themselves since they are added with zeros. We define A_p and A_n as the number of positive and negative data pairs in the pipelines and S_p and S_n as the number of positive and negative data in the stacks.

Lemma 3.5.1. *At any moment t, $A_p \geqslant S_n$ and $A_n \geqslant S_p$ when the last data is not in the k-stage adder pipelines and the stacks are empty initially.*

Proof This lemma can be proved by deduction. When the last data is not in the adder pipelines, $last = 0$ is true. Initially S_p and S_n are both zeros. $A_p^0 \geqslant S_n^0$ and $A_n^0 \geqslant S_p^0$ are true. Suppose the conclusion is true in clock cycle t. Now we prove it is also true in clock cycle $t+1$ by two scenarios:

In the first scenario, the partial sum is positive at clock cycle $t+1$, i.e. $sign(k-1) = +$. If $sign(in) = +$, the partial sum serves as one of the input. $A_p^{t+1} = A_p^t, A_n^{t+1} = A_n^t, S_p^{t+1} = S_p^t, S_n^{t+1} = S_n^t$ remain unchanged. $A_p^{t+1} \geqslant S_n^{t+1}$ and $A_n^{t+1} \geqslant S_p^{t+1}$ are true. If $sign(in) = -$, the partial sum is pushed onto the positive stack, i.e., $S_p^{t+1} = S_p^t + 1$. The adder pipelines lose one positive number but catch a negative input, i.e., $A_p^{t+1} = A_p^t - 1, A_n^{t+1} = A_n^t + 1$. Observing from the input scheduling algorithm where $last = 0$, a negative number is popped off of the negative stack if it is not empty; zero is used as one of the input if the stack is empty. That is, $S_n^{t+1} = S_n^t - 1$, or $S_n^{t+1} = S_n^t = 0$. Under this circumstance, $A_n^{t+1} = A_n^t + 1 \geqslant S_p^t + 1 = S_p^{t+1}$;

$$A_p^{t+1} = A_p^t - 1 \geqslant \begin{cases} S_n^t - 1 = S_n^{t+1} & \text{if } S_n^t > 0 \\ 0 = S_n^{t+1} & \text{if } S_n^t = 0 \end{cases}$$

Therefore, $A_p^{t+1} \geqslant S_n^{t+1}$ and $A_n^{t+1} \geqslant S_p^{t+1}$ are also true when $sign(in) = -$.

The proof for the second scenario where the partial sum is negative at a clock cycle $t + 1$ is similar.

□

Theorem 3.5.1. *The stacks in the distributor are empty at the moment the first partial sum comes out of the k-stage adder.*

Proof Intuitively, a k-stage adder can only at most produce k positive or negative partial sums according to the input and output data scheduling algorithms. These partial sums act as k pairs of inputs of the subtractor. Therefore, no data should be left in the stacks when the last partial sum of a group comes out of the adder. This can be proved more formally. Consider only the unique case of the first group. The stacks are empty when the data of the first group arrives. If we can prove that the stacks remain empty after the last partial sum of the first group comes out of the adder, the subsequent groups will clear the stacks in the same manner.

If the number of data to be accumulated in the first group is less than 5($n \leqslant 4$), the stacks must be empty since $last$ is equal to 1. In this scenario, the first and subsequent partial sum are sent to the output directly. Note that there are no $push$ operations in the input scheduling algorithm with $last = 1$ and the output scheduling algorithm. Therefore the stacks must be empty when $n \leqslant 4$.

If the number of data to be accumulated in the first group is no less than 5($n \geqslant 5$), the operation of distributor can be divided into two phases. During the first phase the last data of the first group is not in the pipeline of the k-stage adder, i.e., $last = 0$. In this phase, the output algorithm is not triggered. In the second phase, the last data of the first group is in the pipeline of the k-stage adder, i.e., $last = 1$. The partial sum of the adder is no longer pushed into the stack or added with the incoming data, but sent to the input of the subtractor. As proven in lemma 3.5.1, $A_p \geqslant S_n$ and $A_n \geqslant S_p$ are true in the first phase. In the second phase, A_p positive numbers in the pipelines are coupled with S_n negative numbers in the negative stack and $A_p - S_n$ zeros as the output of the distributor. A_n negative numbers in the pipelines are coupled with S_p positive numbers in the positive stack and $A_n - S_p$ zeros as the output of the distributor. By the time the last partial sum of the first group leaves the pipeline, all data in stacks are sent to the output of the distributor. In other words, the stacks are empty. □

Theorem 3.5.2. *At any moment $A_p \geqslant S_n$ and $A_n \geqslant S_p$ are true.*

Proof From lemma 3.5.1 and theorem 3.5.1, the conclusion is true when all the numbers in the pipelines belong to the same group. In the case of numbers in the pipeline belonging to multiple groups, no partial sums are pushed onto the stacks in conformity with the input and output scheduling algorithms. Thus the conclusion is also true. □

The other three outputs of the distributor are valid, ICCF and ICCL. The valid signal is set when the data output should be treated as valid outputs. The ICCL/ICCF signal is set if the output is the last/first data pair of a group. In Figure 3.5, the valid signal is directly connected to $last$. The reason is that the output scheduling algorithm is triggered whenever $last$ is set. The ICCL signal is directly connected to the output of the ICCL queue since the last data of a group must be in the last output pairs.

The output ICCF signal should not be set until the valid outputs are available. Moreover, it should be set only when the current output pair is the first one of a group. There are two situations in the latter case. In the first situation, the number of data in the group is no more than four. The first partial sum is in the first output pair. Therefore, the output of the ICCF queue can be used as the output ICCF directly. That is, $ICCF^{t_i}_{out} = valid^{t_i} \bigcap ICCF^{t_i}_{queue} = last^{t_i} \bigcap ICCF^{t_i}_{queue}$, where $\bigcap$ and $\bigcup$ are logical AND and OR operations, respectively. In the second situation, the number of data in the group is larger than four. The first partial sum is not in the first output pair. However the circuit must remember that the first partial has been generated. A D flip-flop is used to serve as the memory cell. It is set once the first partial sum comes out of the pipeline and cleared once a new group comes out. That is, $ICCF^{t_i}_{out} = last^{t_i} \bigcap (\bigcup_{t=t_s}^{t=t_i} ICCF^{t}_{queue})$, where t_s is the clock cycle at which the first partial sum comes out of the adder pipeline; Obviously, $ICCF^{t_s}_{queue} = 1$. Combining the two cases together, we have $ICCF^{t_i}_{out} = last^{t_i} \bigcap (\bigcup_{t=t_s}^{t=t_{i-1}} ICCF^{t}_{queue} \bigcup ICCF^{t_i}_{queue})$.

3.5.2 Subtractor and Full Adder

The input of the subtractor in each clock cycle is a pair of positive and negative data values. It only performs floating-point subtraction. The full adder component has two floating-point full adders(also called Sadders) which can perform addition and subtraction. The input of the full adders can be positive or negative.

One input of Sadder1 is hard-wired to the output of subtractor directly. The other input has one

Table 3.1 Valid Input Patterns of Sadder1 ($k = 4$)

Left Input	First,Last	Last	Second	Last	Last
Right Input	0	First	First	0	Third
Input Number	1	2	3,4	3	4

extra flip-flop used strictly as a single clock cycle delay. The number of outputs from the subtractor for each group is at most k. The Sadder1 will sum up at most two outputs at each clock cycle. The valid input patterns of the Sadder1 in the case of $k = 4$ are shown in Table 3.1. The left and right inputs correspond to the inputs of the Sadder1 in Figure 3.5. In the first column of Table 3.1, there is only one number which is both the last and first one of a group. The input pattern in the third column happens when the number of outputs from the subtractor is 3 or 4. Observing carefully Table 3.1, the input pattern is valid when the last data is the left input or the first data is the right input and the second data is the left input. Hence the $valid$ input signal for the sadder in clock cycle t can be expressed as $valid^t_{in} = valid^t_{out} \bigcap (ICCL^t_{out} \bigcup (ICCF^{t-1}_{out} \bigcap \overline{ICCF^t_{out}}))$ where $valid_{in}$ is the input signal of Sadder1; $valid_{out}$ $ICCL_{out}$ $ICCF_{out}$ are the output signals of the subtractor. The right input must be cleared once a pair of valid inputs is sent to Sadder1. The corresponding circuit is shown in the bottom-left of Figure 3.5. For $k > 4$ where k is power of 2, we need to deal with two input numbers which contain neither the first nor the last data. For example, the input pattern which has the 5th and 6th inputs from the subtractor is valid while the input pattern with the 4th and 5th inputs is not valid. A single memory bit ov can be used to indicate whether the output of the subtractor has the even sequence order. It is also reset after a valid pair of data is sent to the Sadder; otherwise, it is flipped every clock cycle. The valid signal in this case can be expressed as $valid^t_{in} = valid^t_{out} \bigcap (ov^t \bigcup ICCL^t_{out} \bigcup (ICCF^{t-1}_{out} \bigcap \overline{ICCF^t_{out}}))$

The ICCL input of Sadder1 is connected to the ICCL output signal of the subtractor only when the input pattern is valid; otherwise, it is cleared. The ICCF input of Sadder1 is set if the ICCF output is set in the previous or current clock cycle and the input pattern is valid. For $k > 4$, the circuit for ICCL and ICCF inputs of Sadder1 is also correct.

The connection between Sadder1 and Sadder2 is similar to that of Sadder1 and the subtractor. In the $k = 4$ case, each group has at most two inputs for Sadder2. The input pattern is valid only if the ICCL input is set. The right input is cleared after a pair of valid inputs is sent to Sadder2.

Table 3.2 Comparison with Previous Work

Design	No. of Adders	Buffer Size	Clk Frequency	Total Latency	Latency per Group	Out-of-order
PCBT [170]	$\lceil log_2^n \rceil$	$2\lceil log_2^n \rceil$	Decrease with n	$n + \alpha\lceil log_2^n \rceil$	Predictable	No
FCBT [170]	2	$3\lceil log_2^n \rceil$	Decrease with n	$\leqslant 3n + (\alpha - 1)\lceil log_2^n \rceil$	Predictable	No
DSA [170]	2	$\alpha\lceil log_2^n + 1\rceil$	stable	$n + (\alpha - 1)\lceil log_2^\alpha + 1\rceil$	Not Predictable	Yes
SSA [170]	1	$2\alpha^2$	stable	$\leqslant n + 2\alpha^2$	Not Predictable	Yes
FAAC	$< 2 + \lceil log_2^k \rceil$	$2k$	stable	$n + k + \alpha\lceil (1 + log_2^k)\rceil$	Predictable	No

3.6 Data Accuracy

The accumulator performs the summation in a different order than the order of inputs. Since the floating point addition is not truly associative, the accumulated result may not be compliant with the result of sequential floating point additions. In floating point addition, adding a large number with a tiny number will lose precision. For example, the sum of E80 and 2 is still E80 where 2 is shifted out of range to be 0. For a sequence of input (E80,-E80,2,-1), our design returns 0 while an in-order accumulator would return the exact result 1; for a sequence of input (E80,2,-E80,-2), an in-order accumulator returns -2 while our design returns the exact result 0. The value of the positive or negative sub-sum in our design will never decrease. Thus a sequence which would not overflow if summed sequentially may overflow in our design. This problem could be lessened by adding extra bits to the exponent and mantissa. Consider a long sequence of data whose sum converges to 0. The numbers arriving later become smaller and smaller. Since the positive and negative sub-sums in our accumulator will never decrease, the small number will be shifted out as 0. If the sum of the sequence converges to 0, our accumulated result will not be as close to 0 as the result of a sequential accumulator.

3.7 Area and Performance Evaluation

3.7.1 General Analysis

In this section, the characteristics of the FAAC are compared with those designs proposed in [170]. Table 3.2 summarizes the different characteristics of our design and those in the previous work [170]. n_i is the number of data values in a group whereas $n = \sum_{i=1}^{p} n_i$ is the total amount of data in p groups. p is the maximum number of groups which can be put into α segments [170]. In practical applications (e.g. SMVM), n is usually much larger than n_i which in turn is much larger than k. As mentioned in [170], the practical value of n is over tens of thousands and α is under 20 while k in our design is only 4.

As discussed in Section 3.4, the number of adders to accumulate k partial sums is $\lceil log_2^k \rceil$. Suppose the adder which only performs addition in our architecture has k stages instead of 4. The number of adders we use are the k-stage adder, the subtractor and $\lceil log_2^k \rceil$ full adders which is less than $2 + \lceil log_2^k \rceil$ adders. The buffers used in the FAAC are two stacks whose sizes are equal to $2k$.

The delay to accumulate a set of groups is referred to as the $total\ latency$ which starts from the clock cycle at which the first number of the first group comes into the accumulator until the clock cycle at which the sums for all groups come out of the pipeline. The $latency\ per\ group$ is the latency to accumulate a group with n_i numbers starting from the clock cycle at which the first number enters into the accumulator until the clock cycle at which the sum comes out of it. Suppose the last data in a group entered into the accumulator at clock cycle t_s. The sum of the numbers in the group is ready in clock cycle $t_s + C$ where C is the sum of the pipelines stages in the FAAC. In our implementation, $C = 4 + 14 + 15 + 15 = 48$. If a full adder has α pipeline stages and the subtractor is considered to be a full adder, C is equal to $k + \alpha + \alpha \lceil log_2^k \rceil = k + \alpha \lceil (1 + log_2^k) \rceil$. The $latency\ per\ group$ of the FAAC is therefore only dependent on the amount of data in the current group and equal to $n_i + C$. In each clock cycle, input data enters into the FAAC without stalls. It takes n clock cycles for the last number in the last group to enter into the FAAC. After C clock cycles, the sum of the last group is ready. Therefore, the $total\ latency$ of the FAAC is $n + C$.

As mentioned in [170], the PCBT design is infeasible to be implemented in an FPGA due to the large number of floating-point adders. From Table 3.2, the area of PCBT FCBT and DSA increases with the number of data to be accumulated; SSA has the smallest number of adders while the FAAC has the smallest buffer size. With the rapidly increasing gate count in modern FPGAs, the number of adders in the FAAC which grows logarithmically with k is acceptable. As we will see later in this section, the FAAC circuit with $k = 4$ occupies only a small fraction of a Xilinx Virtex 5 device which makes it suitable to be used as a building block in a larger FPGA-based system on the same chip.

The clock frequency of PCBT and FCBT will decrease dramatically with n [170]. Thus, they are not proper choices to accumulate large sets of data. The PCBT and FCBT designs cannot start the computation without knowing the maximum size of datasets beforehand. The $total\ latency$ of DSA and SSA depends on not only the amount of data in the current group but also on those of previous and

Table 3.3 Characteristics of FAAC Modules on Xilinx FPGAs

Module	Pipe Stages	XC2VP30		XC5VLX110T	
		Slices (%)	MHz	Slices (%)	MHz
Adder	4	548 (4%)	162	267 (1.5%)	244
Distributor	4	1303 (9%)	162	562 (3%)	244
Subtractor	14	1367 (9%)	187	494 (2%)	297
Sadder1	15	1806 (13%)	176	740 (4%)	286
Sadder2	15	1798 (13%)	191	702 (4%)	280
Total:	48	6252 (45%)	162	2269 (13%)	244

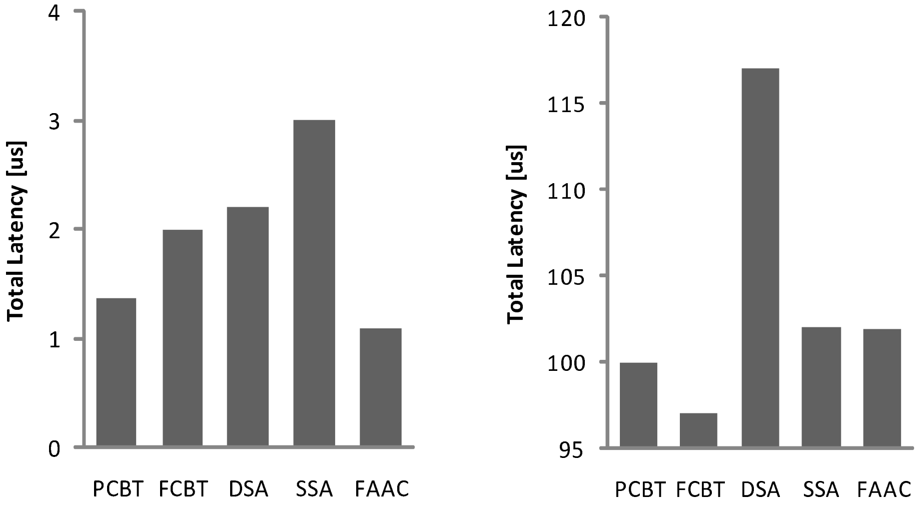
subsequent groups. The unpredictable latency of DSA and SSA imposes great difficulty when applying them to large-scale SMVM and other applications. The dependency comes from the fact that DSA and SSA schedule inputs from different groups to the adders. The previously arrived group whose size is very large might still be coalescing while a small group finishes. This also results in another drawback of the DSA and SSA designs that the order of the sums is different from the order of the input groups. [170] suggests writing the outputs into a buffer and reading them out in order. An additional tag has to be attached to each group to differentiate them in the end. In contrast, the FAAC has a reasonable and predictable latency for each group. The latency for a group in the FAAC is only determined by the size of the current group and the total number of pipeline stages in the accumulator. The output order of the sums in the FAAC is the same as the input order of the groups which eliminates any post processing.

3.7.2 Design Implementations

We made use of a double precision floating-point core from the OpenCores library [116], which we enhanced to perform accumulation using our own VHDL modules. Mentor Graphics Modelsim 6.4c and Xilinx ISE 10.1 were used for simulation and implementation. The proposed architecture was implemented in both a Xilinx Virtex-V XC5VLX110T and a Xilinx Virtex-II Pro XC2VP30 FPGA. The characteristics of all components are listed in Table 3.3. The *Adder* is the 4-stage adder which only performs addition. The $Subtractor$ is the adder which only performs subtraction. The stages of the $FAAC$ are the sum of the stages in $Distributor$, $Subtractor$ and two $Sadders$. However, the area of the $FAAC$ is larger than the sum of the components since the multiplexers and logic gates shown in Figure 3.5 are not included in those components. In the Virtex-II Pro implementation, no more than 1% of the area is used for control logic. The total area of the FAAC is less than 4 full adders. In this initial

Table 3.4 Implementation Characteristics (n=128, $\alpha = 4$, $k = 4$)

Design	Adders	Slices	BRAMs	Clk Freq	Latency
PCBT	7	6808	–	165 MHz	226 μs
FCBT	2	2859	10	170 MHz	$\leqslant$ 475 μs
DSA	2	2215	3	142 MHz	232 μs
SSA	1	1804	6	165 MHz	$\leqslant$ 520 μs
FAAC	< 4	6252	0	162 MHz	176 μs



(a) a single set of 128 inputs

(b) 128 sets of 128 inputs

Figure 3.6 Latency for Accumulating Datasets

implementation, only the *rounding toward zero* mode is supported. Even though the 4-stage adder has a low delay, the critical path of the FAAC lies in the 4-stage adder of the distributor.

We compared our architecture implemented in the Virtex-II Pro device with those designs in [170]. The characteristics of the different designs are summarized in Table 3.4 where $n = 128$, $\alpha = 14$, and $k = 4$. The latency is computed according to clock cycles based on the formula provided in Table 3.2. Though the FAAC uses no more than four adders, the number of the slices it uses is almost the same as that of PCBT. This is due in part to the fact that we used an inefficient open-source floating-point core. Shrinking the resource usage of the adder is part of our future work.

3.7.3 Performance Comparison

Figure 3.6(a) shows the total latency of the designs for accumulating a single set of 128 inputs. Even though the clock frequency of the FAAC is not the highest, it achieves the smallest latency to reduce a single set. This is because the FAAC requires the smallest number of clock cycles. Figure 3.6(b) shows the total latency for accumulating 128 sequential sets of 128 inputs. The FAAC has larger latency than PCBT and FCBT since the clock period dominates the latency with a large number of inputs. However as mentioned in [170], PCBT is not realistic for most FPGA-based designs; PCBT and FCBT must know the largest size of datasets beforehand.

3.8 Contribution

We described an area and performance efficient architecture for a floating-point adder accumulator which can be used to reduce an arbitrary number of groups with arbitrary sizes. Compared with previous work, our approach has several advantages: (1) the area and the clock frequency remain unaffected by the inputs; (2) the accumulation time for each group does not depend on any other group; (3) the number of clock cycles to accumulate a group is equal to the sum of the group size and the number of pipeline stages in the FAAC; (4) the output order of the sums is the same as the input order of the groups; (5) the design does not require any preprocessing of the input data. Besides those advantages mentioned above, the performance analysis shows that our architecture is very competitive compared with other designs.

Though the area usage of the FAAC is greater than most of profiled existing designs, the total amount of area fits comfortably into current-generation (Virtex 5) and older-generation (Virtex-II Pro) mid-range FPGA devices. We believe our FAAC circuit is a good choice for FPGA-based high performance computing applications.

CHAPTER 4. SPARSE MATRIX-VECTOR MULTIPLICATION

In this chapter, we will present a I/O bandwidth sensitive Sparse Matrix-Vector Multiplication(SMVM) engine implemented on reconfigurable platform. Sparse Matrix-Vector Multiplication (SMVM) is a fundamental core of many high-performance computing applications, including information retrieval, medical imaging, and economic modeling. While the use of reconfigurable computing technology in a high-performance computing environment has shown recent promise in accelerating a wide variety of scientific applications, existing SMVM architectures on FPGA hardware have been limited in that they require either numerous pipeline stalls during computation (due to zero padding) or excessive input preprocessing during run-time. For large-scale sparse matrix scenarios, both of these shortcomings can result in unacceptable performance overheads, limiting the overall value of using FPGAs in a high-performance computing environment. In this chapter, we present a scalable and efficient FPGA-based SMVM architecture which can handle arbitrary matrix sparsity patterns without excessive preprocessing or zero padding and can be dynamically expanded based on the available I/O bandwidth. Our experimental results using a commercial FPGA-based acceleration system demonstrate that our reconfigurable SMVM engine is highly efficient, with benchmark-dependent speedups over an optimized software implementation that range from $3.5\times$ to $6.5\times$ in terms of computation time.

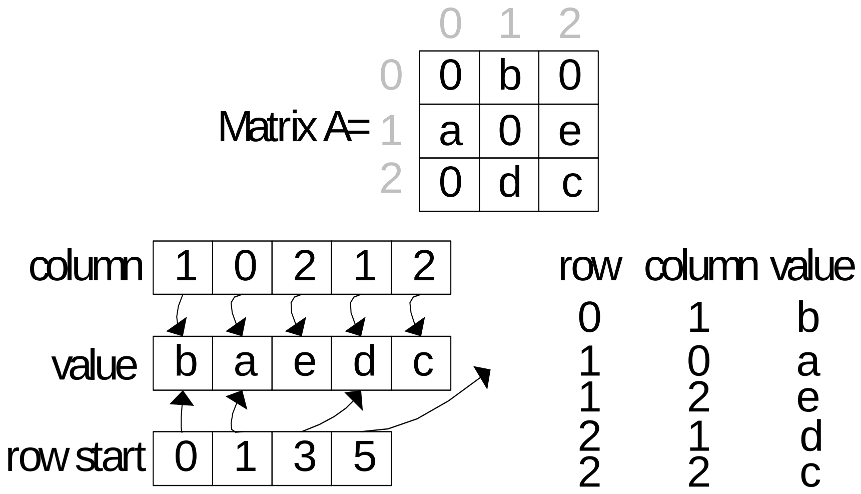
4.1 Introduction

Floating-point Sparse Matrix-Vector Multiplication (SMVM) plays a paramount role in many scientific and engineering applications, including image construction, economic modeling, industrial engineering, control system simulation and information retrieval [159, 172]. In solving large linear systems $y = Ax$ (where A is an $n \times n$ sparse matrix with n_z non-zero entries, and x and y are $1 \times n$ matrices) and eigenvalue (λ) problems $Ax = \lambda x$ using iterative methods, SMVM can be performed hundreds or even thousands of times on the same matrix. For example, Google's PageRank eigenvector problem is dominated by SMVM, where the size of the matrix (n) is of the order of billions [113]. A naive sequential solution of the PageRank problem using the power method would take days to converge. As the size of the datasets used in scientific and engineering applications (including Google PageRank) continue to grow rapidly, the runtime of SMVM is likely to continue to dominate these applications.

SMVM can be characterized as containing large amounts of floating-point computation, coupled with irregular memory access patterns [108]. Memory bandwidth has been a significant performance bottleneck in traditional microprocessor architecture. While multi-level cache hierarchies are successfully used to improve the memory throughput of data-intensive applications, the irregularity of SMVM memory access has led to limited performance scalability of these applications on traditional microprocessors [120, 77]. More recently, Graphics Processing Units (GPUs), in addition to being used for accelerating graphics rendering applications have been exploited for accelerating general-purpose computations [4]. These modern GPUs have a hybrid caching / memory hierarchy with various latencies and optimal access patterns. This architectural complexity (coupled with a highly parallel execution model) has made it a challenge to optimize an irregular data-intensive application such as SMVM for GPUs [108].

Several companies have leveraged the highly parallel and specialized computational fabric of modern FPGAs to develop FPGA-based high-performance computing systems (e.g. SRC [5], Cray [3], XtremeData [8], and Convey [2]). While large-capacity FPGAs have been successfully used in these platforms to accelerate important computational kernels, it is clear that improving on I/O throughput is necessary for accelerating FPGA-based data-intensive applications, including SMVM [49]. Our main contribution is to create a new FPGA-based architecture for SMVM, which attempts to minimize memory access overhead without performing excessive preprocessing of the input matrix, as required in previous approaches. Our design efficiently keeps track of inter-row computation in the SMVM pipeline, eliminating the need for zero padding (stalls), while adding little hardware overhead as compared to previous approaches.

To save storage space and improve computing performance, only nonzeros in sparse matrix are stored and processed. The sparse matrix representation is critical to the architecture design. There are a multitude of sparse matrix formats. Different formats require distinct storage-computation features



(a) Compressed Sparse Row Format (b) MatrixMarket Coordinate Format

Figure 4.1 Compressed Sparse Row Format Example

and manipulate entries of the matrix dissimilarly. For example, diagonal format represents those sparse matrices whose nonzero values are restricted to a small number of matrix diagonals. ELLPACK [19] is an appropriate choice when the maximum nonzeros per row does not substantially differ form the average. The aim of our architecture is to deal with arbitrary sparsity patterns though a particular architecture tailored for a highly specific class of matrices has higher performance. The *compressed sparse row*(CSR) format is a most popular and efficient sparse matrix presentation regardless the structure of the nonzero entries. The CSR format for a sparse matrix is composed of three vectors: $value$, $column$ and $row\ start$. $value$ has the nonzeros in a row-major order; $column$ stores the column indices for nonzeros in $value$; $row\ start$ points to the locations in the $value$ vector that starts a row. The relationship of these vectors to represent a matrix is illustrated in figure 4.1. we use the MatrixMarket Coordinate format to represent general sparse matrices. The coordinates and values are explicitly given for each nonzero element in a matrix.

4.2 Relative Work

Due to its singular importance , enormous effort has been devoted to SMVM with the aim of maximizing the performance on different computing systems since last century [115, 144]. For example, the work in [154, 66, 112, 145, 120, 88, 135, 76, 153] targeted to improve SMVM performance on traditional microprocessor architectures. The strategies to optimze SMVM performance on multi-processor platforms can be found in [12, 102, 23, 94, 128, 30, 31, 119, 79, 87, 83, 104, 151]. And the solutions of SMVM on message-passing based mutli-computer system were presented in [148, 60, 82, 111]. Some other work for hybrid architectures, recently emerged GPU, multicore and FPGA architecture, can be found in [159, 93, 66, 157, 20, 52, 172, 136, 41] [154] improved the performance of SMVM by splitting the sparse matrix into a sum where each term in the sum is stored in a block structure. [144] describes techniques that increase instruction-level parallelism on superscalar RISC processors. [112] presented performance models and compact data structure for SMVM cache blocking which significantly reduces cache misses in SMVM. [77] described optimization techniques to increase register blocking. [145] described techniques that increase instruction-level parallelism for superscalar RISC processors. [120] proposed data structures along with reordering algorithms to reduce the number of memory indirections. [88] used index and value compression to achieve impressive SMVM speedups. [135] outlined an approach to implement irregular sparse matrix solvers in high performance fortran. [76] modeled the data locality in the execution SMVM. [153] developed upper and lower bound on the performance of SMVM. In [115], the balancing of distributed storage of non-zero elements among parallel processor arrays is investigated. [12] implemented algorithms to compute SMVM on a distributed memory multiprocessor system, PARSYS SN1000 in 1992. [102] performed an average analysis of data distribution for SMVM on a many-processor hypercubic network. [23] gave vector partitioning algorithms which have a good balance of the communication load among the processors. [94] proposed adaptive runtime tuning mechanisms to improve the parallel SMVM performance on distributed memory systems. Their mechanisms balances computational load at runtime with negligible overhead. [128] proposed a MRD(Multiple Recursive Decomposition) partitioning algorithm which maps the data into a multiprocessor network with mesh topology. Because of the deficiencies of the graph model for decomposing sparse matrices for parallel SMVM, [30, 31] proposed hypergraph models. [119] improved the data locality on shared memory multiprocessors(SMP). [79] described optimization techniques for SMVM on uniprocessor and SMPs, such as register blocking, cache blocking, and matrix reordering. [87] discussed scalable implementations of SMVM on a cc-NUMA machine SGI Altix3700. [83] proposed a SIMD algorithm for computational electromagnetic and scattering matrix models in distributed memory computing. [104] maximized SMVM performance on RISC based MIMD computers by using a cache mirror strategy. [151] presented a two dimensional data distribution approach for SMVM which spread the computation and communication work evenly over the processors. [148] described general software utilities for performing SMVM on distributed-memory message-passing computers. [60] incorporated several optimization techniques into an efficient implementation using message-passing to improve the SMVM performance with symmetric matrices. [82] addressed the implementation issues in a shared virtual memory system on a distributed memory parallel computer. [111] combined fast load-balancing with efficient message-passing techniques to alleviate computational and inter-node communications challenges. [66] performed an extensive experimental evaluation of the SMVM kernel for single and multi-threaded version on a variety of modern commodity architectures. [157] implemented SMVM on different supercomputers, such as massively-parallel systems, vector computers and distributed shared-memory systems. Different programming models are used in these systems, including hybrid programming models which combined message-passing and shared-memory programming models. The emergence of Graphics Processor Units(GPU) as powerful parallel systems in recent years is an attractive choice for SMVM. Several optimization approaches of SMVM kernels on GPU architecture have been presented [19, 20, 108, 106, 58]. Some online toolkits and libraries for GPUs are also available [36, 114]. [20] explored several efficient implementation techniques for SMVM in CUDA. [52] defined a storage format which improve the SMVM performance on vector computers.In recent years, several architectures on different platforms have been proposed. SMVM is implemented on several multicore platforms in [159] to explore the potential advantages of explicit multicore programming. [159] presented several optimization strategies to implement SMVM on multicore systems. [93] presented optimization for SMVM and its generalization to multiple vectors, SMMVM.

FPGAs have great potential in coping with the irregularity of SMVM due to their intrinsic pipelined ability, intrinsic parallelism and configurable architecture. A reconfigurable hardware accelerator for fixed-point matrix-vector multiply is proposed in [28]. The design of floating-point SMVM in [172] employs double precision floating point multipliers, adders, and performs multiple floating-point operations as well as I/O operations in parallel. The Xilinx Virtex-II Pro XC2VP70 is used as the targeted device. The evaluated result shows that their design achieves over 350MFLOPS for all test matrices when the memory bandwidth is 8GB/s. However, the performance of their design is greatly affected by

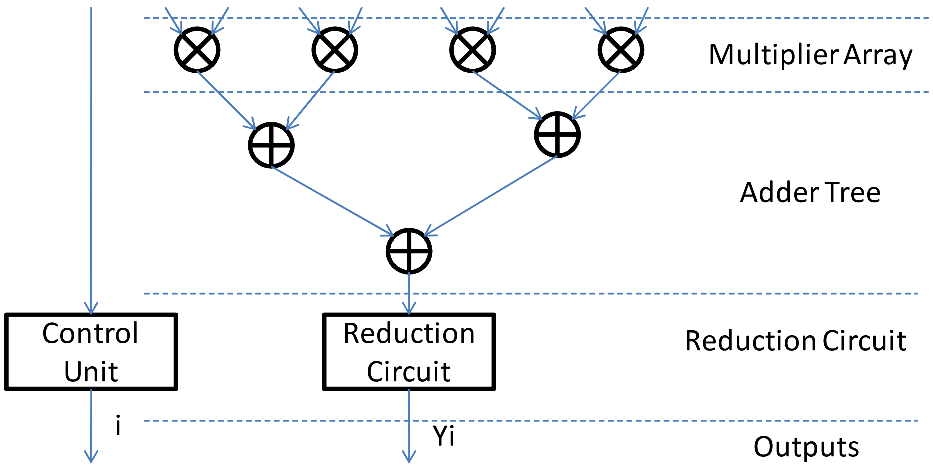


Figure 4.2   Ling-Viktor Architecture(k=4)

the overhead. The smaller the overhead, the higher the performance. The implementation in [41] partitions the set of n dots products across multiple Processing Elements(PEs). A matrix mapping algorithm is critical to minimize computation latency while minimizing the inter-PE communication. Also,since the BlockRAM memory(BRAM) is used only to store the matrix, the matrix size is constrained by the BRAM. All of the previous works require the pre-preprocessing of the matrix or vector. This brings high overhead in the case of large matrix. The preprocessing of the super matrix will take very long time which amortizes the performance gains of FPGAs. In this chapter, we design an expendable and high performance SMVM Engine architecture without preprocessing and zero padding. The computation throughput is expandable according to the I/O bandwidth. It can handle arbitrary size of matrix.

4.3 Motivation

The SMVM architecture proposed in [172] is shown in Figure 4.2. The architecture is composed of a multiplier array, a binary adder tree, reduction circuit and control unit. The k floating-point multipliers are in the leaf nodes of the adder tree. Each of the $k-1$ internal nodes in the tree contains a floating-point adder. The root node of the tree is fed into a reduction circuit which in charge of accumulating the intermediate addition values. The control unit controls the I/O operation and the operations of the multipliers and adders.

During the computation, the input matrix data are stored in the external memory and the vector are stored in the local storage of all the multipliers. In each clock cycle, the multipliers multiply the non-zero elements in the m^{th} column of a matrix row and the m^{th} element of the vector. The results of the multipliers are transferred into the inputs of the adders and are summed up. If the number of non-zeros of each row equals k, the output of the root node yields the final result for each y_i, hence the reduction circuit is not necessary. However, the number of non-zeros l_i in each row i of the matrix is unpredictable. If the l_i is larger than k, row i will be partitioned into $\lceil \frac{l_i}{k} \rceil$ subrows. All subrows except the last one contain exactly k non-zeros. The last subrow are padded with zero and expanded to have k elements. The subrows of each row are read into the system consecutively, and the partial sums are transferred to reduction circuit by the root adder. The reduction circuit accumulates the sums of the sub-rows and produces y_i. If l_i is smaller than k, zero-padding are also inserted by the control unit. The ling-viktor architecture is affected adversely by the zero padding. They measure the performance by the metric ”overhead”.

Definition 4 (Zero Padding Overhead). *For a matrix A with n_z non-zeros, the zero-padding overhead ov is $\frac{n_p}{n_z}$, where n_p is the total number of zeros needed for padding. Obviously,*

$$ov = \frac{\sum_{i=1}^{n}(\lceil \frac{l_i}{k} \rceil \times k - l_i)}{n_z} = \frac{(\sum_{i=1}^{n}\lceil \frac{l_i}{k} \rceil \times k) - n_z}{n_z} = \frac{\sum_{i=1}^{n}\lceil \frac{l_i}{k} \rceil \times k}{n_z} - 1$$

It is clearly that ov is dependent on the number of multipliers k and the sparsity pattern of the matrix. Given a matrix, n_z is fixed. In the extreme case, if l_i is equal to 1 for all row i, $n_z = n$,

$$ov = \frac{\sum_{i=1}^{n}\lceil \frac{l_i}{k} \rceil \times k}{n_z} - 1 = \frac{\sum_{i=1}^{n} k}{n} - 1 = \frac{n \times k}{n} - 1 = k - 1$$

We analyzed the zero-padding overhead of the benchmarks used in [172] based on different value of k. The result is shown in Figure 4.3. From the figure, we can see that the zero-padding overhead is almost 2.5 for some benchmarks.

4.4 Our Approach

From the previous analysis, we can improve the performance by reducing or diminishing the number of padded zeros. Although not an optimization technique per se, to save storage space and improve performance, it is extremely common to store and process only the nonzero values in a sparse matrix.

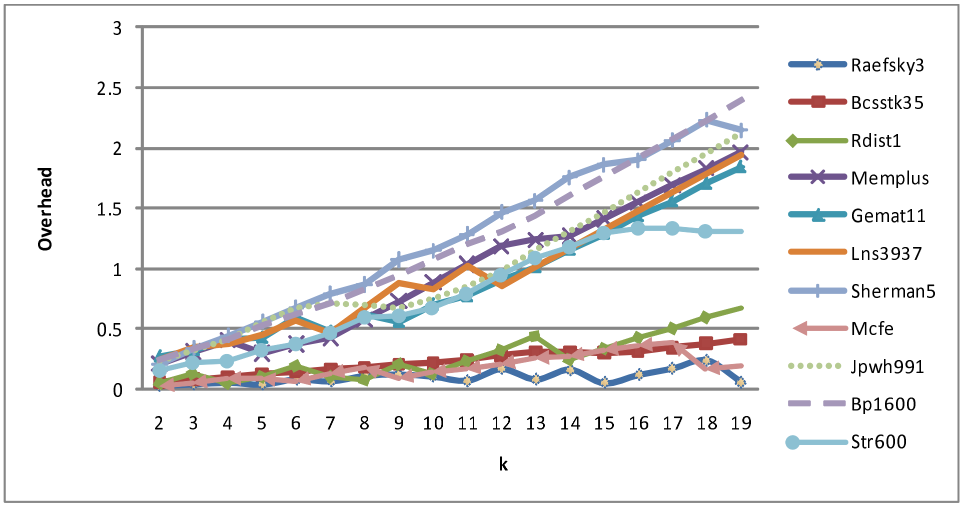


Figure 4.3 Overhead

Our architecture assumes that the sparse matrix is stored in row-major order and aims at improving the performance of arbitrary sparsity patterns. One straightforward implementation is composed of a multiplier array (k multipliers) and a binary adder tree. In each clock cycle, the multipliers in the array multiply the non-zero elements in the same row in the sparse matrix with the elements in the vector. If the number of non-zeros of each row l_i is equal to k, this architecture is quite efficient. However, if l_i is smaller than k, there will be zero paddings; if l_i is larger than k, there will be pipeline stalls. The goal of our design is to remove these hurdles without overly complicating the architecture.

4.4.1 Input Patterns

Before going into the details of our architecture, we first introduce some important concepts. Suppose there are four (k=4) multipliers in the architecture of [172]. The elements of the matrix are transferred to the system in one clock cycle, disregarding whether they are from the same row or consecutive rows. For example, we send b, a, e, d from the matrix in Figure 4.1 to the system in the same clock cycle even though b, a, e and d are in different rows. The non-zeros of a sparse matrix in row-major order can be considered as a stream of numbers which can be divided into a bulk of segments. Each segment has k numbers. The Input Pattern Vector (IPV) for each segment is defined as the following:

Definition 5 (Input Pattern Vector). *The i^{th} bit of a k-bit IPV is 1 if the i^{th} number in a segment is the last non-zero of a row; otherwise, the i^{th} bit of a k-bit IPV is 0. For example, IPV=1010 (k=4) if*

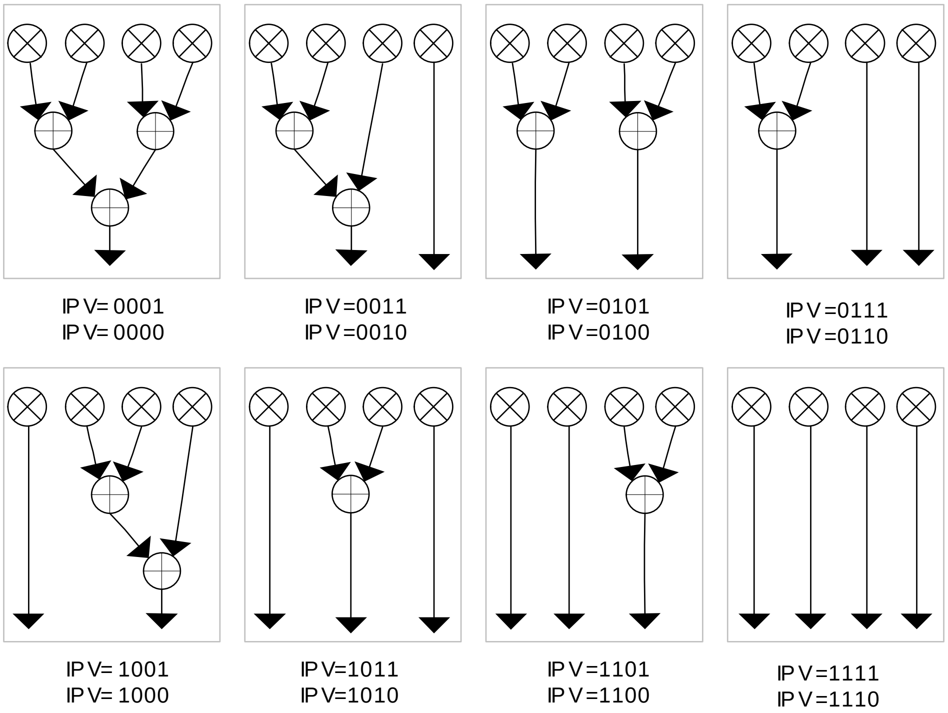


Figure 4.4 Input Pattern and Tree Structure (k=4)

the input elements are b, a, e, d *as shown in Fig 4.1.*

The generation of IPVs is straightforward according to the definition. Based on the sparse matrix storage format, IPVs can be attached as meta-data if the value of k is known for the architecture to be used. Alternately, IPVs can be generated during the computation which will introduce some computation overhead.

Different IPVs can correspond to different structural configurations of the adder tree. In Figure 4.4, the four inputs from the matrix are from four different rows when IPV=1111/1110. Thus, the four outputs from the multipliers need not be sent to the adder tree. The first input may have to be accumulated with the outputs of the previous clock cycle. The last input must be accumulated with the outputs of the next clock cycle when IPV=1110. We will discuss accumulation in detail later. In Figure 4.4, the first three inputs must be added together and sent to the adder tree in case IPV=0011/0010 because they are from the same row. To assure the uniqueness of the adder-tree structure, we require the addition operation to be left associative. For example, the first two inputs are summed up and then added with the third input when IPV=0011/0010. This requirement does not add any negative effect to the com-

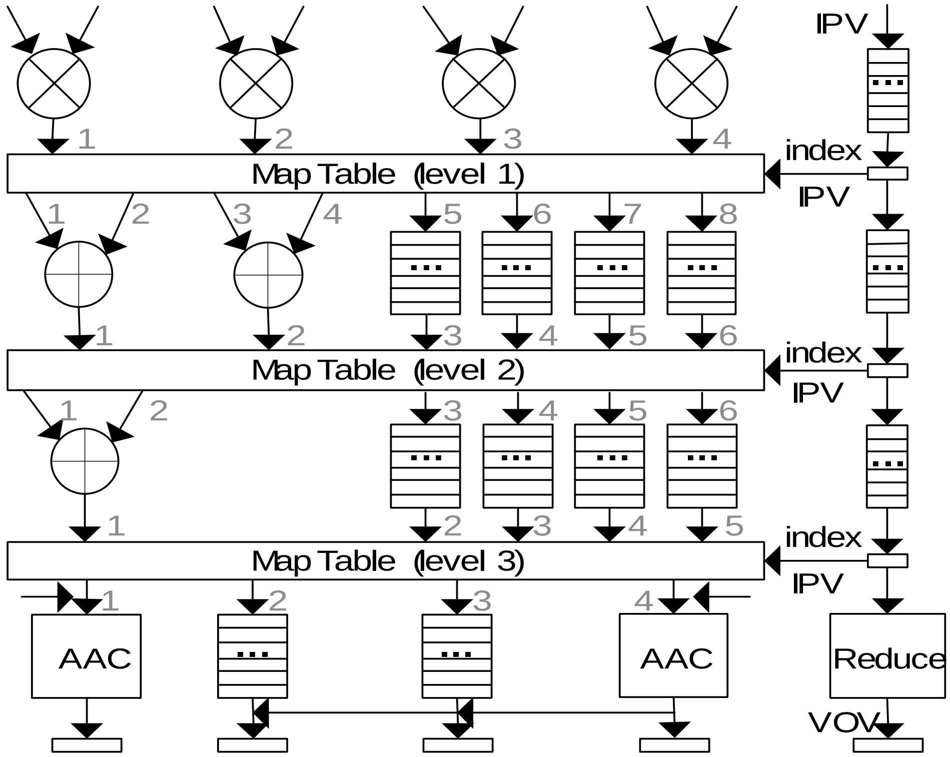


Figure 4.5 Proposed SMVM Architecture (k=4)

putation accuracy due to the associativity of addition operation. The relationship between the IPV and adder tree structure is proved in Theorem 4.4.1.

Theorem 4.4.1. *The number of adder trees with distinct structures is 2^{k-1} if the length of IPV is k.*

Proof The tree structure is uniquely determined by the row distribution among the k products. Suppose the k products are from i rows where $i \leq k$. The number of distinct tree structures is C_{k-1}^{k-i}. Therefore, the total number of distinct tree structures is $\sum_{i=1}^{k} C_{k-1}^{k-i} = C_{k-1}^{0} + C_{k-1}^{1} + \cdots + C_{k-1}^{k-1} = 2^{k-1}$. Another straightforward proof can be observed from Figure 4.4. The last bit of the IPV has no effect on the structure of the tree. Thus, the first $k-1$ bits which has 2^{k-1} values completely make up the tree structure. □

4.4.2 SMVM Architecture

Our architecture as shown in Figure 4.5 is composed of a multiplier array, an adder tree, an addder accumulator (AAC), and a map table. All components in the architecture are pipelined and run in parallel. The data flows top-down in a pipelined fashion. In each clock cycle, k numbers from the matrix are multiplied pairwise with k numbers from the vector. The number of multipliers (k) is a configurable parameter which can be any natural number. The connections between the outputs of multipliers and the inputs of adder tree are configured by the map table on the fly. Similarly, the connections among the adder tree and register arrays, and between the adders/register arrays and accumulators, are determined by the map table during the run time.

4.4.3 Multipliers and Adders

The double-precision floating-point multipliers and adders used in the SMVM architecture are pipelined and have two inputs. In each clock cycle, k pairs of numbers from the sparse matrix and vector are fed into the inputs of the k multipliers. After some clock cycles, the k products are ready at the outputs of the multipliers. Based on the input pattern vector, some of the products are summed up together since they come from the same row of the input matrix. Since each adder has only two inputs, an adder tree is leveraged to sum more than two products. A product does not necessarily go through the adder-tree if it is the only non-zero in a row during a specific clock cycle. For example, the first and second products will not be put into the adder-tree if IPV=1101/1100 in Figure 4.4. In this scenario, these products are sent into register arrays between the map tables in Figure 4.5 which have the same delay as the adders to guarantee they arrive at the next level at the same clock cycle as those sums of products. The register arrays are just 64-bit FIFO queues. One 64-bit number (corresponding to a single double-precision floating-point value) is pipelined out of an array in each clock cycle. Due to the associativity of addition, $k-1$ adders are needed to sum up k numbers. Therefore, $k-1$ floating-point adders are enough to reduce no more than k products in our architecture. The adder tree has in total $\lceil log_2 k \rceil$ levels. The number of adders in each level is determined when all the inputs are from the same row in a particular clock cycle which is generated by the algorithm shown in Algorithm 9. In the case the inputs are from different rows, register arrays must be used in some level. Suppose all the k inputs are from the same row in a clock cycle. After some clock cycles, there are k outputs from the k multipliers. These k outputs serve as the k inputs to the adders in the first level of register-adder tree where they are sent to $k/2$ adders. If k is an odd integer, one of the outputs is sent to the register array. After some clock cycles, the number of intermediate results sent to the register-adder tree in the next

Algorithm 9 Generate the Adder Number in Each Level

```
Input: the number of multipliers, $k$
Output: the number of adders in each level, $\widehat{N}[\,]$
  $K \leftarrow k$;
  for $i = 1$ to $\lceil log_2 k \rceil$ do
    $\widehat{N}[i] \leftarrow K/2$;
    $K \leftarrow K/2 \; + \; K\%2$;
  end for
```

level is $k/2 + k\%2$. The same principle is applied to each level as shown in Algorithm 9.

4.4.4 IPV Reduction

As stated in Section 4.4.1, the structure of the adder tree is entirely determined by the IPV. On the other hand, the register-adder tree structure is dynamically generated at run-time to fully utilize the reconfigurable ability of the FPGA. Therefore, the IPV is an ideal source to code and index the map table and is critical to bring forth valid outputs. The process of IPV reduction and mapping to register-adder tree is illustrated in Figure 4.6. The IPV is checked bit by bit from left to right. If the current bit is 0, it is paired off with the next bit. Each bit is only in one pair. The pattern of a pair is either (0,0) or (0,1) which is reduced to 0 or 1 and sent to the next level. Those bits which are not in any pairs are sent to the next level directly. This reduction procedure is repeated in each level and executed $\lceil log_2 k \rceil$ times. The length of the IPV decreases or remains unchanged during the reduction process. Initially, each bit in the IPV is corresponding to a multiplier in SMVM architecture. If a bit in the IPV is reduced with another adjacent bit, the product of the corresponding multiplier is summed up with its adjacent product; otherwise, the product is sent to the register array. The number of valid outputs in the register adder tree is equal to the length of the reduced IPV in the same level. The structure of the register-adder tree is a bijection of the IPV. The isomorphism between them is the proof of correctness of the map table generator discussed later. The Reduced IPV (RIPV) has the same number of 1s as the original IPV with all set bits “pushed” to the left.

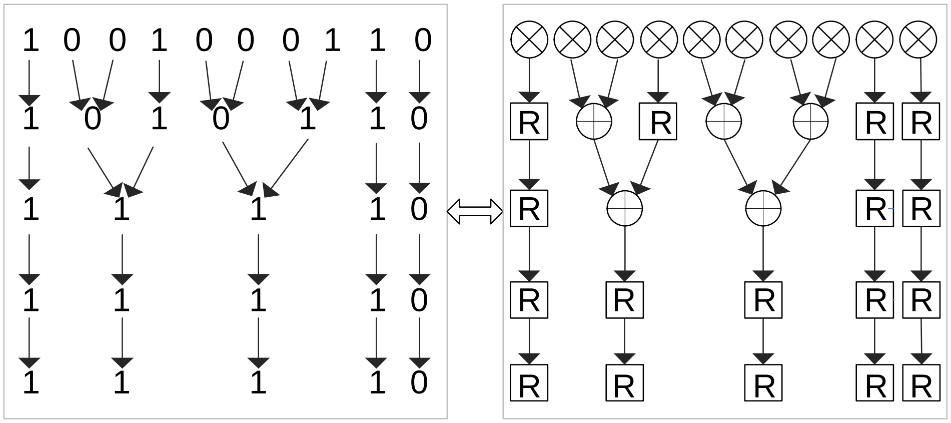


Figure 4.6 IPV Reduction and Tree Structure (k=10, IPV=1001000110)

4.4.5 Adder Accumulator

When a row has more than k non-zeros, say n_z^i, the products have to be moved into the SMVM engine in $\lceil \frac{n_z^i}{k} \rceil$ clock cycles. The value of n_z^i is unpredictable in a large sparse matrix. Since floating-point adders are usually deeply pipelined to achieve high clock frequency and the subsums of each row are generated in different clock cycles, designing a floating-point adder accumulator is not trivial. Several AAC (also referred to as reduction circuit) architectures have been proposed in the recent years [170, 173, 38, 75]. The AAC in [140] can reduce an arbitrary number of groups with arbitrary size and does not require any preprocessing of the inputs. Consequently, it is suitable to handle the accumulation operation for SMVM with an irregular input matrix. The AAC accepts a data input of a group in each clock cycle. Once all the elements in a group go into the AAC, the sum of the group will be ready as an output after some number clock cycles. Figure 4.7 is a timing diagram of computation model of AAC in [140]. The products for row A, B, C and D are sent to AAC sequentially. The number of pipeline stages of the AAC is two. The sum of a row is ready two clock cycles after the last product is sent to the AAC. In the extreme case, the AAC acts as a FIFO queue if each row only includes one product.

Taking AAC as a building block renders neatness and compactness to the SMVM architecture. For some patterns in Figure 4.4, the sum of products of a row in current clock cycle t_i should be accumulated with the sum of products of the same row in the $(t_i - 1)^{th}$ and/or the $(t_i + 1)^{th}$ clock cycle. In a clock cycle, the products may come from several rows. Only the sum of products in the first and last row in the current clock cycle need be accumulated with products in adjacent clock cycles. The products in other rows are already reduced into one sum by the register-adder tree in the last level. Therefore, only two AACs should be used to accumulate the sum of products generated by the register-adder tree. The left AAC accumulates the sum of products of the first row in the current clock cycle with the sum products of the last row in the last clock cycle. In this case, the first row in the current clock cycle and the last row in the last clock cycle are the same. The right AAC accumulates the sum of products of the last row in the current clock cycle and the sum products of the first row in the next clock cycle. In this case, the last row in the current clock cycle and the first row in the next clock cycle are the same.

For example, if we have IPV=1000 in clock cycle t_{i-1} and IPV=1010 in clock cycle t_i, the sum of the last three products in clock cycle t_{i-1} should be accumulated with the first product in clock cycle t_i. The second and third products in clock cycle t_i come from the same row and are reduced into one sum by the register-adder tree which should not be accumulated with any other products. The last product in clock cycle t_i should be accumulated with the first sum of products in clock cycle t_{i+1}. If there is only one row in a clock cycle, the singular sum of products reduced by the register-adder tree is sent to the left AAC. There are at most k reduced sums in each clock cycle if the k products come from k different rows. Hence, another $k - 2$ register arrays are placed in the same level with the two AACs. The inputs to these register arrays are the reduced sums of rows which are resolved in the register-adder tree. If the products in the previous clock cycle are from at least two rows and the last row has products in both previous and current clock cycle, the positions of the two AACs must be switched in the current clock cycle(illustrated as the horizontal arrow for AAC inputs in Figure 4.5). This assures the sum of products of a row are sent to the same AAC for accumulation.

4.4.6 Map Table

The map table is a two-dimensional array and provides connections between the inputs and outputs of adjacent levels during run-time. For an architecture with k multipliers, k numbers from the matrix are multiplied with k numbers from the vector in each clock cycle. At the same time, a k-bit IPV is transferred to the register array pipeline. The map table entries used in each level and clock cycle are solely determined by the integer value of the IPV. That is, the integer values of the IPVs are used to index the map table. The register arrays for IPVs shown in the rightmost column of Figure 4.5 are k-bit

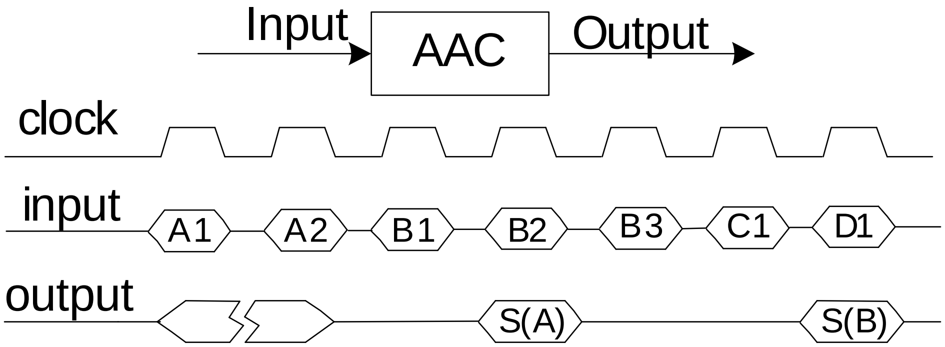


Figure 4.7 Timing Diagram of AAC Computation Model

FIFO queues.

In order to compact the table, the inputs/outputs of each level are assigned an index. The inputs and outputs of the adders are numbered sequentially ahead of the register arrays. For each level, the index starts from 1 instead of 0 since 0 is reserved for those inputs which are unconnected. In general, the number of inputs in level i are $2 \times a_i + k$ where a_i is the number of adders in level i. There are in total $k - 1$ adders and $\lceil log_2 k \rceil$ levels, while each level has k register arrays. The last level which is a register-AAC group always has k inputs. By adding up the number of inputs in each level, there are in total $2 \times (k - 1) + k \times \lceil log_2 k \rceil + k$ inputs which is also the number of columns in the map table. The number of rows in the table is 2^{k-1} where each row corresponds to a distinct IPV.

Table 4.1 shows the map table in the case for k=4. The first level is the mapping between the multipliers and the register-adder tree. The second level is the mapping inside the register-adder tree. The third level represents the mapping between the register-adder tree and the register-AAC group. Each input in a specific level l whose column index is i and row index is $\widehat{ipv}$ in the table has a value $\widehat{T}[\widehat{ipv}][i]$. This indicates that that input in level l should receive data from the $\widehat{T}[\widehat{ipv}][i]^{th}$ output of the level $l - 1$ when the current IPV is equal to $\widehat{ipv}$.

In other words, an entry in the table is the index of an output in previous level. The maximum value of an entry is $k + \lfloor \frac{k}{2} \rfloor$ which is the number of outputs of the first level in the register-adder tree. Therefore, $\lceil log_2(k + \lfloor \frac{k}{2} \rfloor) \rceil$ bits are employed to store each entry. The number of bits to store a map table is $2^{k-1} \times (2 \times (k-1) + k \times \lceil log_2 k \rceil + k) \times \lceil log_2(k + \lfloor \frac{k}{2} \rfloor) \rceil$. The number of inputs in each level is always no less than the number of outputs in the previous level. If $\widehat{T}[\widehat{ipv}][i]$ is equal to 0, the input is

Table 4.1 Map Table (k=4)

		Level 1								Level 2						Level 3			
IPV	**Index**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**1**	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**
000X	**0**	1	2	3	4	0	0	0	0	1	2	0	0	0	0	1	0	0	0
001X	**1**	1	2	0	0	3	4	0	0	1	3	4	0	0	0	1	0	0	2
010X	**2**	1	2	3	4	0	0	0	0	0	0	1	2	0	0	2	0	0	3
011X	**3**	1	2	0	0	3	4	0	0	0	0	1	3	4	0	2	3	0	4
100X	**4**	2	3	0	0	1	4	0	0	1	4	3	0	0	0	2	0	0	1
101X	**5**	2	3	0	0	1	4	0	0	0	0	3	1	4	0	2	3	0	4
110X	**6**	3	4	0	0	1	2	0	0	0	0	3	4	1	0	2	3	0	4
111X	**7**	0	0	0	0	1	2	3	4	0	0	3	4	5	6	2	3	4	5

not connected to any interface in current clock cycle and the input value of it is 0. Notice that the last output to the last level is always sent to the right AAC. The rationale behind this is that if the outputs in the last level are from different rows, the output from the last row may need to accumulated with the sum of the products in the same row of the next clock cycle. If there is only one row, the sum of the products reduced by the register-adder tree is sent to the left AAC as stated in Section 4.4.5.

The map table generation is closely related to the IPV reduction as discussed in Section 4.4.4. The map table generator checks the IPVs bit by bit from left to right. Based on the action of reduction, it puts the outputs of the previous level into the inputs of adders or register arrays. The software algorithm of the map table generator is shown in Algorithm 10 which iterates on different values of the IPV. For each given IPV, the algorithm produces the entries of the map table level by level. For each level, $\widehat{B1}$ is the IPV which is reduced in the current level; $\widehat{B2}$ is the IPV generated by reducing $\widehat{B1}$ in the current level. $\widehat{S1}$ is the output index vector of the previous level while $\widehat{S2}$ is the output index vector of the current level. If the index of the i^{th} output in the previous level is j, then $\widehat{S1}[i] = j$.

The indices are the numbers assigned to the adder outputs and register arrays outputs in Figure 4.5. Recall that the adders and register arrays are numbered sequentially. However, the register arrays are possibly used ahead of the adders during the reduction of IPVs. Hence, we need $\widehat{S}$ to record the output index sequence. Obviously, the value of $\widehat{S}$ of the first level is the same as the serial number assigned to the multipliers, i.e., $\widehat{S1} = \{1, 2, \cdots, k\}$. i_a^{out} and i_r^{out} are the index counters of those numbers assigned to the outputs of the adders and register arrays respectively. $i_{\widehat{B}}$ is the index counter for $\widehat{B}$ and $\widehat{S}$. In Figure 4.5, a unique number is assigned to each input of the adders and register arrays in a level. i_a^{in} and i_r^{in} are the input index counters of those numbers for the adders and register arrays respectively. i_s^{in}

is the index of the map table for each ipv. $\widehat{N}[i]$ is the number of adders in level i.

An IPV is reduced in a $while$ loop as shown in Algorithm 10. As byproducts of the $while$ loop, a reduced IPV $\widehat{B2}$ and output index vector of the current level $\widehat{S2}$ are generated. There are three scenarios while reducing IPVs. In the first scenario, the current index is the last input of the last level. The last output in previous level is sent to $\widehat{T}[ipv][2 \times (k-1) + k \times \lceil log_2 k \rceil + k]$, i.e., the input of the right AAC. Otherwise, if the current bit in $\widehat{B1}$ can not be reduced or is the last bit, the current indexed output of previous level is sent to the current indexed register, i.e., $\widehat{T}[ipv][i_r^{in}] \leftarrow \widehat{S1}[i_{\widehat{B1}}]$. If the current bit should be reduced with the next bit in $\widehat{B}$, as shown in the last scenario, the corresponding two outputs in the previous level should be sent to the two inputs of an adder in the current level. Meanwhile, the two bits $\widehat{B1}[i_{\widehat{B1}}]$ and $\widehat{B1}[i_{\widehat{B1}}+1]$ in the current IPV are reduced into $\widehat{B2}[i_{\widehat{B2}}]$ in the next IPV. The value of the newly reduced bit $\widehat{B2}[i_{\widehat{B2}}]$ is equal to $\widehat{B1}[i_{\widehat{B1}}+1]$ since $\widehat{B1}[i_{\widehat{B1}}]$ is equal to 0.

The output of the algorithm for k=4 is similar to Table 4.1. The output table is manually hard-coded into the hardware description language as a two-dimensional array which is synthesized and put into distributed memory of FPGA chips. For an architecture with k multipliers, the map table is solely determined by the value of k and is independent of the characteristics of matrices or vectors. Note that the software algorithm is not implemented as part of the architecture and has no impact on the run-time performance - map table generation is a one-time task that is generated offline for a given value of k. As illustrated in Table 4.1, the map table entries used in each clock cycle are solely determined by the value of the IPV. The register array for IPV shown in the rightmost column of Figure 4.5 is utilized to index the map table in each clock cycle. For an architecture with k multipliers, k numbers from the matrix are multiplied with k numbers from the vector in each clock cycle. At the same time, a k-bit IPV is transferred to the register array pipeline.

4.4.7 Outputs

The AACs and $k-2$ register arrays in Figure 4.5 send the sums of products to the k output registers in each clock cycle. However, there will not be k valid outputs from k different rows unless the first $k-1$ bits of the corresponding IPV are 1s. Otherwise, some outputs are just partial sums of products from different rows. In this sense, we need a k-bit Valid Output Vector (VOV) to indicate the valid outputs in each clock cycle. If the i^{th} bit of the VOV is 1, the i^{th} output register has a valid output;

Algorithm 10 Map Table Generation Algorithm

```
Input: the number of multipliers, $k$
Output: the map table, $\widehat{T}[\,][\,]$
  for $ipv = 0$ to $2^{k-1} - 1$ do
    Convert $ipv$ to a binary vector $\widehat{ipv}$;
    $\widehat{B1} \leftarrow \widehat{ipv}$; $\widehat{B2} \leftarrow \emptyset$; $\widehat{S1} \leftarrow \{1 \cdots k\}$; $\widehat{S2} \leftarrow \emptyset$;
    for $level = 1$ to $\lceil log_2 k \rceil$ do
      $i_{\widehat{B1}}, i_{\widehat{B2}} \leftarrow 0$; $i_a^{out} \leftarrow 1$; $i_r^{out} \leftarrow \widehat{N}[level] + 1$;
      $i_a^{in} \leftarrow i_s^{in}$; $i_r^{in} \leftarrow i_s^{in} + 2\widehat{N}[level]$;
      while $i_{\widehat{B1}} < l_{\widehat{B1}}$ do
        if $level = \lceil log_2 k \rceil$ and $i_{\widehat{B1}} = l_{\widehat{B1}} - 1$ then
          $\widehat{T}[ipv][3k - 2 + k\lceil log_2 k \rceil] \leftarrow \widehat{S1}[i_{\widehat{B1}}]$;
          $\widehat{S2}[i_{\widehat{B2}}] \leftarrow k$; $\widehat{B2}[i_{\widehat{B2}}] \leftarrow \widehat{B1}[i_{\widehat{B1}}]$;
          $i_r^{out}, i_r^{in}, i_{\widehat{B1}}, i_{\widehat{B2}} \leftarrow increase\ 1$;
        else if $\widehat{B1}[i_{\widehat{B1}}] = 1$ or $i_{\widehat{B1}} = l_{\widehat{B1}} - 1$ then
          $\widehat{T}[ipv][i_r^{in}] \leftarrow \widehat{S1}[i_{\widehat{B1}}]$;
          $\widehat{S2}[i_{\widehat{B2}}] \leftarrow i_r^{out}$; $\widehat{B2}[i_{\widehat{B2}}] \leftarrow \widehat{B1}[i_{\widehat{B1}}]$;
          $i_r^{out}, i_r^{in}, i_{\widehat{B1}}, i_{\widehat{B2}} \leftarrow increase\ 1$;
        else
          $\widehat{T}[ipv][i_a^{in}] \leftarrow \widehat{S1}[i_{\widehat{B1}}]$;
          $\widehat{T}[ipv][i_a^{in} + 1] \leftarrow \widehat{S1}[i_{\widehat{B1}} + 1]$;
          $\widehat{S2}[i_{\widehat{B2}}] \leftarrow i_a^{out}$; $\widehat{B2}[i_{\widehat{B2}}] \leftarrow \widehat{B1}[i_{\widehat{B1}} + 1]$;
          $i_a^{in}, i_{\widehat{B1}} \leftarrow increase\ 2$; $i_a^{out}, i_{\widehat{B2}} \leftarrow increase\ 1$;
        end if
      end while
      $i_s^{in} \leftarrow i_s^{in} + 2N[level] + k$;
      $\widehat{B1} \leftarrow \widehat{B2}$; $l_{\widehat{B1}} \leftarrow i_{\widehat{B2}}$; $\widehat{S1} \leftarrow \widehat{S2}$;
    end for
  end for
```

Table 4.2 Component Characteristics

	AAC	Adder	Multiplier
DSP48Es	–	$3(1.56\%)$	$10(5\%)$
LUTs	$5861(2\%)$	–	–
Clock Speed (MHz)	240	300	260
Pipeline Stages	48	14	15

otherwise, its output is a partial sum of a row.

The number of 1s remains unchanged during the IPV reduction. From the definition of the IPV, the i^{th} bit of a IPV is 1 if the i^{th} number is the last non-zero in a row. That implies that all the non-zeros of the i^{th} row have been transferred into the system. After the reduction of the register-adder tree with or without the accumulation of AACs, the sum of products of that row is ready at a output register. In other words, the number of valid outputs in a clock cycle is equal to the number of 1s in the corresponding IPV. Consequently, the IPV has a close relationship with the VOV. Nevertheless, the positions of the valid outputs in the output registers may not be the same as those indicated in IPV, due to the reduction in the register-adder tree and accumulators. Based on Algorithm 10, the products are sent to the left operation units in the architecture as often as possible except that the final product is sent to the right AAC. Therefore, one can expect that all the valid outputs are packed into the left side of the output register arrays except the last one. If we shift the output of the right AAC to the left position adjacent to a valid output, the Reduced IPV (RIPV) in the last level can be used as the VOV. In our architecture, we generate the VOV by extending the RIPV to k bits filled with 0s if the number of 1s is less than k. And the output of right AAC is sent to the $(i+1)^{th}$ output register if the number of 1s in the current IPV is i. In this way, there are k outputs from the output registers and a k-bit VOV in each clock cycle. If the i^{th} bit of the VOV is 1, the output of the i^{th} output register is a valid sum for a row. The reduce module in Figure 4.5 generates the VOV. Notice that the IPV reduction for VOV here is different from the IPV reduction in the map table generation. The former is performed during run-time for each IPV input while the latter is processed off-line in Alg 10.

Table 4.3 Implementation Characteristics

	k=5	k=7	k=10	k=12
AACs	2	2	2	2
Adders	4	6	9	11
Multipliers	5	7	10	12
Total DSP48Es	62(32%)	88(46%)	127(66%)	153(80%)
Map Table Dimension	16×28	64×40	512×68	2048×82
Map Table Resource Usage (LUTs)	2880(1%)	6800(3%)	9200(4.5%)	12980(6%)
Total LUTs	21184(10%)	27575(13%)	35081(16%)	40358(19%)
Clock Speed (MHz)	200	200	200	166
Bandwidth Requirement (Gb/s)	129	180	258	310
Total Pipeline Stages	106	106	120	120

4.5 Hardware Characteristic

4.5.1 Area

Our proposed SMVM architecture was implemented in VHDL and simulated and functionally verified in Mentor Graphics Modelsim. For our initial hardware resource usage experiments, we targeted a Xilinx Virtex-5 XC5VLX330 FPGA, which has 192 DSP48E slices (multiply-accumulate operators) and 207360 slice LUTs among which 3420Kbits distributed memory can be utilized. The Block Ram (BRAM) blocks of the device serve as 10368 Kbits block memory. We used the Xilinx Core Generator utility to customize floating-point multipliers and adders. These were configured to use the DSP48Es, which greatly increases the resulting chip's area and power efficiency. Although our work is independent of the data representation, we used IEEE 754 double-precision numbers in our implementation. The number of pipeline stages of the multipliers and adders are 15 and 14 respectively as shown in Table. 4.2. The number of pipeline stages of the register arrays in the register-adder tree is accordingly 14. The implementation of the AAC is from [140] which has 48 pipeline stages.

The implementation characteristics for some different values of k is shown in Table 4.3. The number of AACs remains unchanged. The numbers of multipliers and adders are k and $k-1$ respectively. Each multiplier is mapped to 10 DSP48E blocks. Each adder uses 3 DSP48E blocks. According to the discussion in Section 4.4.6, each map table has 2^{k-1} rows and $2 \times (k-1) + k \times \lceil log_2 k \rceil + k$ columns. In the case of k=10, the size of the map table is $2^{k-1} \times (2 \times (k-1) + k \times \lceil log_2 k \rceil + k) \times \lceil log_2 (k + \lfloor \frac{k}{2} \rfloor) \rceil \approx$

136K bits which is held in distributed memory. If the distributed memory is not large enough to hold the map table as k is increased, the map table could be placed into the BRAM blocks. In the case of k=10, the resource utilization of the map table and two AACs is no more than 9%. The other 16%-9%=7% LUTs are mainly used by the register arrays and control logic. The clock speed of the whole design is lower than that of any components. This is due to the increasing routing delay with larger resource utilization. The critical path lies in the AACs whose speed can be improved by using higher band architecture [140]. With the increased value of k, the clock speed is lower because of the extra routing delay in the critical path. The total pipeline stages of the architecture is $15 + 14 \times 4 + 48 + 1 = 120$ clock cycles when k is 10. Hence, the sum of products of a row will not be ready until 120 clock cycles after the last non-zero element of it has entered the system. Different values of k with the same number of levels have the same of number pipeline stages in the architecture.

4.5.2 Performance

FLoating point Operations Per Second (FLOPS) is a common measure of computing performance in high-performance computing environments. To compute $Y = AX$ where $A_{i,j} \in A, X_j \in X, Y_j \in Y, i, j \in [1, n]$, the operation $Y_i = Y_i + A_{i,j} \times X_j$ is performed for each non-zero $A_{i,j}$. In other words, each non-zero element of A requires two floating-point operations [172]. Thus, the total number of floating-point operations is $2n_z$ for $Y = AX$. The FLOPS are computed by the following equations:

$$FLOPS = \frac{n_{FP\ operations}}{t_{total}} = \frac{2n_z}{n_{clock\ cycles} \times t_{clock}} \tag{4.1}$$

In our design, k non-zeros from the sparse matrix are fed into the architecture in each clock cycle and the last 120 clock cycles without inputs are used to wait for the final results coming out of the pipelines. The total number of clock cycles is $\frac{n_z}{k} + 120$. Usually, $\frac{n_z}{k}$ is much larger than 120 thus $n_{clock\ cycles} \approx \frac{n_z}{k}$. The performance of our design in terms of FLOPS is consequently:

$$FLOPS_1 \approx \frac{2n_z}{\frac{n_z}{k} \times t_{clock}} = \frac{2k}{t_{clock}} = 2k \times f_{clock} \tag{4.2}$$

In the case of k=10 and clock frequency is 200MHz, the performance of our design is 4 GFLOPS. With higher value of k and enough FPGA hardware resources, higher FLOPS can be achieved. Table 4.4 lists the performance of our implementation with previous software implementations. The information for the other processors in the table is from [41, 153].

Table 4.4 Performance of SMVM on Various Platforms

Processor	**Release Year**	**MHz**	**Peak MFLOPS**	**SMVM MFLOPS**
Pentium 4	2000	1500	3000	425
Power 4	2001	1300	5200	805
Sun Ultra 3	2002	900	1800	108
Itanium	2001	800	3200	345
Itanium 2	2002	900	3600	1200
Virtex2 6000-4	2001	140	2240	1500
Virtex5 LX330	2006	200	$2kf_{clock}$	4000

We now compare the performance of our architecture with the architecture shown in Figure 4.2 in terms of FLOPS. Suppose both architectures have the same number of multipliers, I/O bandwidth and run at the same clock frequency. We also assume the I/O bandwidth is high enough to fully utilize the computation power of both architectures. Going back to Equation 4.1, the number of valid floating-point operations $2n_z$ and the clock period t_{clock} remain unchanged. The total number of clock cycles $n_{clock\ cycles}$ is around $\frac{n_z+n_p}{k}$ where $n_p = ov \times n_z$ is the total number of zero paddings introduced in definition 4. The performance of the design in Figure 4.2 in terms of FLOPS is:

$$FLOPS_2 \approx \frac{2n_z}{\frac{n_z+n_p}{k} \times t_{clock}} = \frac{2k \times f_{clock}}{1+ov} \tag{4.3}$$

$$\alpha = \frac{FLOPS_1}{FLOPS_2} = 1+ov \tag{4.4}$$

The speedup of our design over the design in Figure 4.2 is determined by the overhead ov. As shown in Section 4.3, $0 \leq ov \leq k-1$, thus $1 \leq \alpha \leq k$. Figure 4.8 shows the speedup based on the data from Figure 4.3 in the case of k=10. Independent of the IO bandwidth and data source, the comparison of different hardware SMVM architectures is shown in Table 4.5, where f is the clock frequency, α is the zero-padding overhead [172], and $u \in [0,1]$ is a utilization factor [51]. Note that the IPVs are generated only once for each sparse matrix given a specific k. In the scenario of computing $Y = A^m X$, for $m \gg 1$, the IPVs are generated once for matrix A. For each iteration, the IPVs are sent with the data to the FPGA. This is a one-time overhead, as compared to the other architectures in Table 4.5 for which the overhead recurs across all iterations.

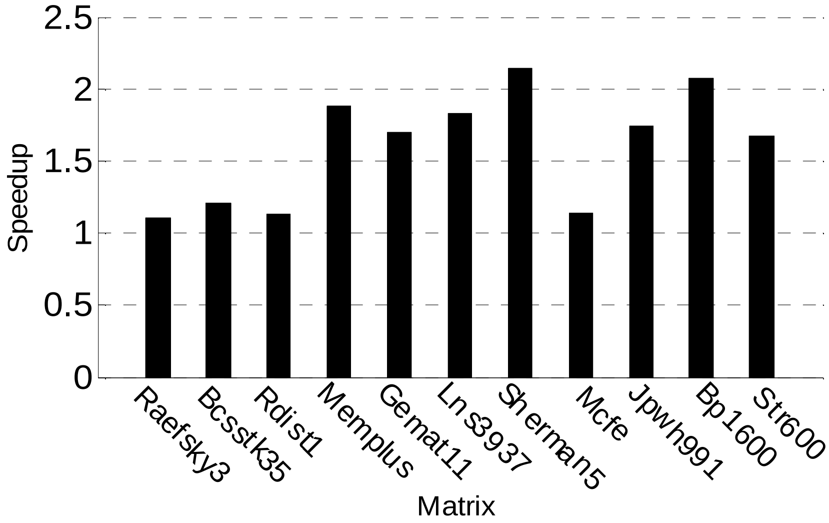


Figure 4.8 Speedup of Our Architecture Over the Architecture in [172]

Table 4.5 Comparison of SMVM Hardware Architectures

	Multiplier	**Adder**	**AAC**	**FLOPS**	**Required Overhead**
[172]	k	$k-1$	1	$2kf/(1+\alpha)$	Zero padding
[41]	k	k	0	$2kf \times 66\%$	Static data scheduling, zero padding
[137]	k	k	1	$< 2kf$	Pipeline stalling, zero padding
[51]	k	k	0	$2kf \times u$	Strip elements reordering
[168]	k	k	k	$< 2kf$	Data preprocessing, zero padding
This Work	k	$k-1$	2	$2kf$	IPV generation

4.6 Evaluation in Reconfigurable System

The performance of the computing platform as a whole system determines the speed of SMVM. SMVM is a typical data-intensive application where the processing requirement scales linearly with data size. IO bandwidth is a usually major performance constraint for data-intensive computing. However, the IO bandwidth and pin count requirement of the FPGA chip is practically determined by design strategy and where the original data is stored. For example, the computation power of our SMVM engine can be fully utilized and the required IO bandwidth is negligible if the data is stored in on-chip memory. The required number of pins can be greatly reduced if the SMVM module is wrapped in other modules.

Consequently, a memory-management methodology that would improve the effectiveness of our approach is to partition large sparse matrices into blocks of smaller size which can fit into the architec-

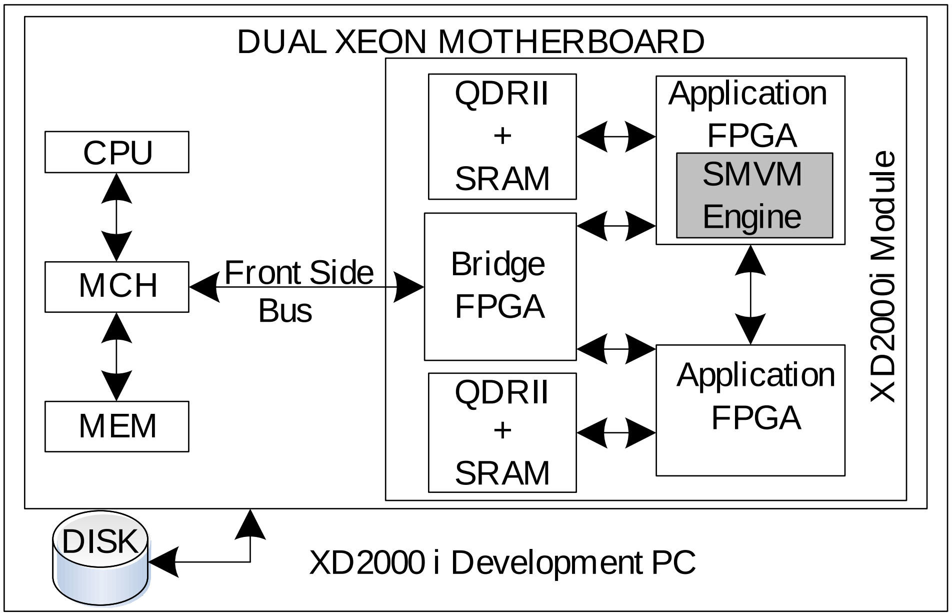


Figure 4.9 XtremeData XD2000i Architecture with SMVM Engine

ture. Similarly, the loading time cannot be ignored if the original data is stored in external storage (e.g. hard disk, network storage). Our SMVM architecture can be considered as an engine with k processing elements (PEs). Each PE processes one element from the input matrix in one clock cycle. If the whole matrix is partitioned into $\frac{n_z}{k}$ blocks, all elements in one block can be processed in one clock cycle, with the blocks loaded into the engine and processed sequentially. This is also referred to as the *locally parallel globally sequential(LPGS)* method in [91]. If the whole SMVM engine is considered as one PE and multiple FPGA chips are used, then *locally sequential globallly parallel (LSGP)* methods [91] can be used. In the LSGP approach, a large matrix can be divided into multiple blocks where each block contains a continuous sequence of rows. Our planned future work in porting this architecture to the Convey HC-1 platform [2] will adopt the LSGP method. A discussion of these and other partition schemes can be found in [172, 41].

4.6.1 Experimental Setup

The XD2000i from XtremeData [8] is an example of a cutting-edge system that provides a complete platform to deploy high-performance computing solutions with FPGAs. It greatly reduces the development effort of integrating the software and hardware components. The XD2000i consists of a Dual

Xeon motherboard with one Intel Xeon processor and two Altera Stratix III EP3SE260 FPGAs in the other socket as shown in Figure 4.9. The Xeon CPU has 4GB system memory and four cores running at 1.6GHz which communicate with the FPGAs through the Intel Memory Controller Hub (MCH). The MCH is connected to the bridge FPGA via the 1067M Front Side Bus (FSB) interface. The FSB with 8.5GBytes/s peak communication bandwidth is the highest performing, lowest latency bus in this generation of Intel platform [98]. The communication behavior between the MCH and FPGAs is nailed down to the cycle accuracy level.

The clock speed of the FPGAs is hardwired to 100MHz by the platform. We used the Altera MegaWizard Plug-In Manager to customize the IEEE 754 double-precision floating-point multipliers and adders in our implementation. The multipliers were configured to use dedicated multiplier circuitry (DSPs). Our SMVM engine is integrated into one of the application FPGAs with other hardware IP components provided by XtremeData. Specifically, the SMVM engine module is wrapped in an $afu\_loopback$ module which is again wrapped in an app_core module provided by the XtremeData XD2000i platform. These modules serve as the intermediator between the SMVM engine and the bridge FPGA. The design was simulated, synthesized, placed and routed using Altera Quartus II 8.1.

During the run-time, the CPU reads the entries in the input sparse matrices and vectors from disk and prepares a data buffer in system memory. The size of the buffer for transmission is no larger than 4MB, which is an XD2000i constraint. Once the data buffer is full, it is sent to the FPGA chip directly using a send-request command. Only one Send operation may be active at a time. The Receive command corresponding to the previous Send command must complete before the next Send command is submitted. However, the Send request may return immediately so that CPU can perform other operations before sending the Receive request. The computation in hardware is overlapped with the fetching data operation in software. A data buffer of known size is received back from the FPGA hardware and is stored in system memory [8]. We calculate the size of the receiving data buffer by counting the number of rows which have at least one nonzero in the sending data buffer. Based on our experimental results, the bandwidth available for sending data from CPU to FPGA and back to CPU during run-time is 1.035 GBytes/s in each direction, which is solely a function of the XtremeData architecture and drivers. All inputs are set to zeros to fill the idle clock cycles during which no data has arrived.

The performance of our system implemented on the XtremeData XD2000i platform is compared with SparseLib++ [122], which is a widely-used general-purpose software implementation and an object oriented C++ library for sparse matrix computations. It is designed for portability and high performance across a wide class of machine architectures [45]. The SparseLib++ is built upon the Sparse BLAS which provides high-performance sparse matrix-vector kernels that can be application or platform-specific [47].

4.6.2 Parameter Design

The design is characterized by several parameters which include the depth of the pipelines in the multipliers, adders and AACs, and the number of bands in the AAC. In this implementation, the depth of the pipelines in the multipliers, adders and AACs is 11, 14, and 48, respectively. Increasing these parameters would achieve a higher clock frequency at the expense of greater resource usage and initial latency. The number of multipliers, adders and register arrays, the width of IPV and the size of the map-table increase with k. Depending on the design strategy and data source, the value of k is usually determined by several factors including IO bandwidth, clock speed, pin count of the chip, and FPGA logic resources. However, the pin count of the FPGA chip has no impact on k since the SMVM engine must be wrapped in other hardware modules in the XD2000i platform. The on-chip memory such as BRAMs or distributed memory usually runs at the same clock frequency as the logic and the bit-widths are large enough. If the data is originally stored in the on-chip memory, the number of multipliers k is only limited by the logic resource in FPGAs, allowing the SMVM performance to grow linearly with the value of k. When the data source is off-chip, the IO bandwidth constrains the value of k in this platform. In our SMVM architecture, the number of input bits in each clock cycle is $2k \times 64 + k$ if double-precision floating-point data is used. The calculation of k based on peak IO bandwidth is consequently:

$$k \approx \frac{IO\ bandwidth}{129 \times f_{clock}} = \lceil \frac{8.5GBytes/s}{129bits \times 100MHz} \rceil = 6 \tag{4.5}$$

Based on the values of clock frequency and IO bandwidth shown in Section 4.6.1, $k = 6$ is enough to fully utilize the peak bandwidth of the FSB. Hence, using more multipliers will not provide higher performance given that communication bandwidth would become the bottleneck. Only one application FPGA is used since the two FPGA chips share the same communication channel. In Table 4.6, the computation time for FPGA ranges from k=3 to k=12 (the design complexity exceeded the capabilities

Table 4.6 SMVM Characteristics on Matrices

Name	Size	Nonzeros	t_{read}	t_{trans}	t_{ipv}
str_600	363×363	3279	6459	61	36
cdde2	961×961	4681	10741	87	44
bp_1600	822×822	4841	9387	90	63
jpwh_991	991×991	6027	9660	112	78
mcfe	765×765	24382	41630	452	171
ex25	848×848	24612	54897	457	173
tomography	500×500	28726	60897	533	200

of the Quartus FPGA synthesis tool for $k > 12$). The SMVM architecture uses 62% of the total logic in the Stratix III EP3SE260 FPGA when k=12. The components in the whole XD2000i system which incorporates the SMVM architecture into the FPGA consume 14% of the total available FPGA LUT resources.

4.6.3 Experimental Results

In order to have a fair comparison, the SparseLib++ software also runs on the XtremeData 2000i platform using one CPU of the Intel Xeon subsystem. Table 4.6 characterizes the sparse matrices from the University of Florida Sparse Matrix Collection [37]. The matrices are stored as MatrixMarket Coordinate format. These matrices have different sparsity patterns. The nonzeros in ex25, mcfe, jpwh_991, cdde2 are confined to a diagonal band. The sparsity patterns of tomography, bp_1600 and str_600 are not obvious. The values in the vectors are generated randomly and saved in a file on disk. The FPGA computation time for different benchmarks is shown in Figure 4.10. The computation speedup of FPGA over CPU (the running time ratio of CPU to FPGA) is depicted in Figure 4.11. Note that our design can run at higher clock frequencies in the Stratix III EP3SE260 FPGA, which would require linearly less time for computation.

The floating-point data is stored in scientific notation format. The t_{read} in Table 4.6 is the time for reading the matrix and vector from the disk into the system memory. The size of the file is determined by the number of digits in the mantissa which explains t_{read} for cdde2 is larger than that of bp_1600 even though cdde2 has less non-zeros. The t_{trans} is the time for transferring the data from the system memory to the FPGA chips. The IPV generation time by software is t_{ipv}. The time is measured in

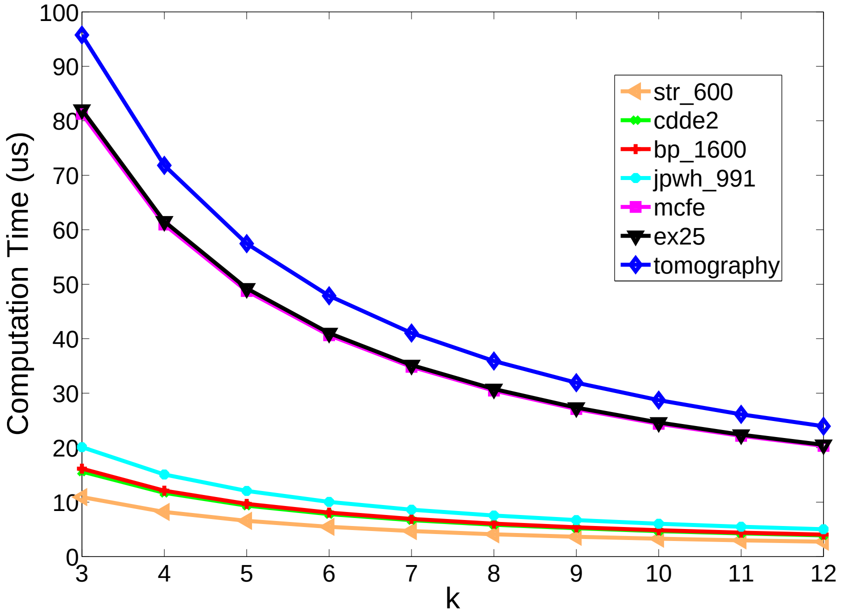


Figure 4.10   FPGA Computation Time as a Function of k

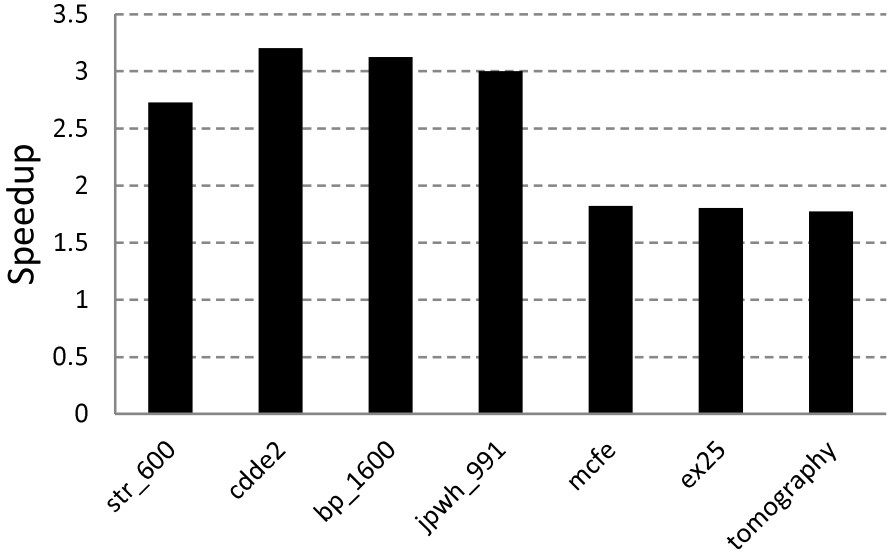


Figure 4.11   Speedup of FPGA Computation Over CPU (for k=6)

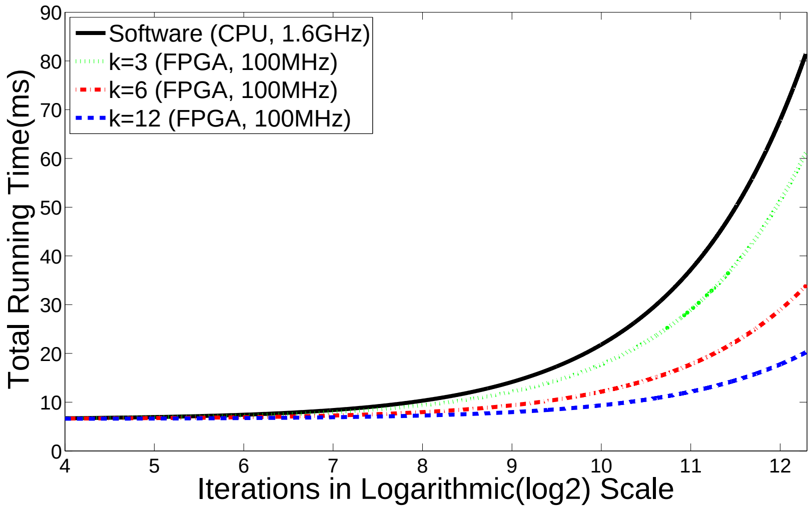


Figure 4.12 Total Running Time Comparison for `str_600`

microseconds. For a sparse matrix, SparseLib++ reads the matrix and vector from disk to the system memory before sending them to the CPU for computation. Therefore, the preprocessing time is t_{read}. To use the SMVM engine in FPGAs, the data in the system memory buffer along with the generated IPVs is sent to the FPGA chips. The preprocessing time is t_{read}+t_{trans}+t_{ipv}.

To cope with the deficiency in input communication bandwidth, a finite state machine (FSM) in the $afu\_loopback$ module implemented in VHDL is used to feed data into the SMVM engine. In each clock cycle, the FSM collects valid input data from the $write\_data$ bus and stores it into the input buffer of SMVM engine. If the buffer is full (with k pairs of data and an IPV), the data in it is sent to the SMVM engine. Otherwise, the inputs of SMVM engine are all zeros and the FSM stays in the data collecting state. From Figure 4.6, the computation time is much smaller than the preprocessing time in computing $Y = A^m X$ for $m = 1$. Figure 4.12 compares the total running time of CPU and FPGA increasing with m (the number of SMVM iterations) when to compute $Y = A^m X$. If m is small, the CPU system is even faster than the FPGA system because of the proprocessing overhead. Nevertheless, the running time is dominated by the computation time when SMVM is heavily used in practical computation.

It should be noted that assuming the input matrix is stored continuously on-chip, the scale of the problem (the number of non-zeros) is constrained by the available on-chip memory. For example, in Altera Stratix III EP3SE260, two M144K memory blocks with $4k \times 64$ (depth×width) are used to store the matrix and vector. When k=6, 40 M144K blocks are combined to serve as IO storage. Since there are in total 48 M144K blocks, the number of non-zeros in the matrix should be no more than $4k$. Partitioning large matrices for iterative computation in FPGAs is a planned aspect of our future work. In summary, the SMVM architecture as implemented on XD2000i platform has an advantage over the traditional approach in system level when the source data is stored in the on-chip memory of FPGAs or the SMVM is iteratively used (such as the power method for computing $Y = A^m X$, for $m \gg 1$).

4.7 Contribution

An expendable and high performance sparse matrix-vector multiplication architecture is proposed in this chapter. It can deal with sparse matrices with arbitrary size and sparsity pattern without impairing the performance. It also eliminates zero-paddings and pipeline stalls with the introduction of IPVs. With enough communication bandwidth and hardware resources, the performance of the architecture grows linearly with the number of multipliers, k, which is a configurable parameter in the architecture. The paths in which the data flows through the architecture are dynamically determined by the map table during runtime. The map table is generated by software off-line and hardcoded into the VHDL code.

We prototyped our architecture on both Xilinx (Virtex-5) and Altera (Stratix III) FPGAs. Based on the post place and route reports, we compare it with previous hardware architecture. The results show that our architecture outperforms existing state-of-the-art in FPGA-based SMVM. To compare it with the software implementation, we also implemented the architecture on the XtremeData reconfigurable platform. Our design is faster than an existing optimized software program.

CHAPTER 5. ACCELERATING K-NEAREST NEIGHBOR ALGORITHM IN TEXT CLASSIFICATION USING SMVM

We briefly introduced classification in section A.1. In this chapter, we will briefly examine an important type of classification–text classification(TC). Appendix C provides the detail information of TC. Text classification is one hot research areas which spans several disciplines, text mining which is a branch of data mining, machine learning, information retrieval, artificial intelligence and natural language processing. Compared with traditional analysis performed on structural database, text mining refers to a collection of methods to find patterns and extract knowledge from unstructured text data [56]. Text classification is also called text categorization, topic classification,or topic spotting [101], or **document classification**. Its definition is: Given a set of documents classes C and example documents for each class, construct a **document classifier** which, given a document d, find the class(es) to which d is most similar.

5.1 General Definition of TC

In text classification, each document has a description $d_i \in D$ where D is the document space and includes the descriptions of all documents. A given set of classes $C = \{c1, c2, \cdots, c_n\}$ are human defined based on the requirement of the applications. Each document d_i in a training set is assigned a label with a probability p_{ij}, that is, $< d_i, c_j, p_{ij} >\in D \times C \times P, P = [0, 1]$. The problem of TC is to construct a classifier $\widehat{\Gamma}$ which map documents to classes:

$$\widehat{\Gamma} : D \rightarrow C \times P$$

Note that, when $n = 2$ and $p_{ij} = 1/0$, Γ is a binary classifier(see Figure C.1(a)); when $n > 2$ and $p_{i,j=c} = 1, p_{i,j\neq c} = 0$ where c is a constant, Γ is a multiclass, single-label hard classification(see Figure C.1(b)); when $n > 2$ and $p_{i,j} = 1/0$, Γ is a multiclass, single-label hard classification(see

Figure C.1(c)); when $n > 2$ and $p_{i,j} \in [0, 1]$, Γ is a multiclass, single-label soft classification(see Figure C.1(d)).

5.2 Vector Space Model

Documents written in natural languages can not be fully understood by the classifier running in a computer. They must be uniformly applied to all documents while conserving the $semantics$. In recent years, the semantic technologies have an immense impact on search engines, thus on society and economy. Vector Space Model(VSM) is one of those new semantic technologies [149]. The VSM was developed for the SMART information retrieval system by Gerard Salton and his colleagues. The idea of the VSM is to represent a document as a point in a space. Points that are close together are semantically similar and points that are far apart are semantically distant. Most research engines use VSMs to measure the similarity between a query and a document [101, 42]. In addition to the success in information retrieval [9, 100], researchers in natural language processing applied VSM on multiple-choice synonym questions from Test of English as a Foreign Language(TOEFL) and achieved a score of 92.5% whereas the average human score was 64.5% [149]. VSM is also widely used in Artificial Intelligence(AI) and cognitive science. There are three types of VSMs: term-document, word-context and pair-pattern [149]. Our topics are focusing on the term-document VSMs. In this section, we will introduce the process to build a VSM from a document written in natural language and the operations of VSMs for text classification.

5.2.1 Linguistic Processing of VSM

5.2.1.1 Term Vocabulary

Usually the documents in a corpus have the same format. However, this is not the case for the documents available on the internet. For example, a document can be the plain English text in ASCII encoding, various single byte or multi-byte encoding schemes(such as UTF-8), Portable Document Format(PDF), Microsoft Word, XML, even a compressed format(such as ZIP, RAR). Therefore, the first step to build VSM is **decoding the byte sequence to a character sequence of natural language**.

The second step is tokenization shown in Figure 5.1. Given a character sequence, **tokenization**

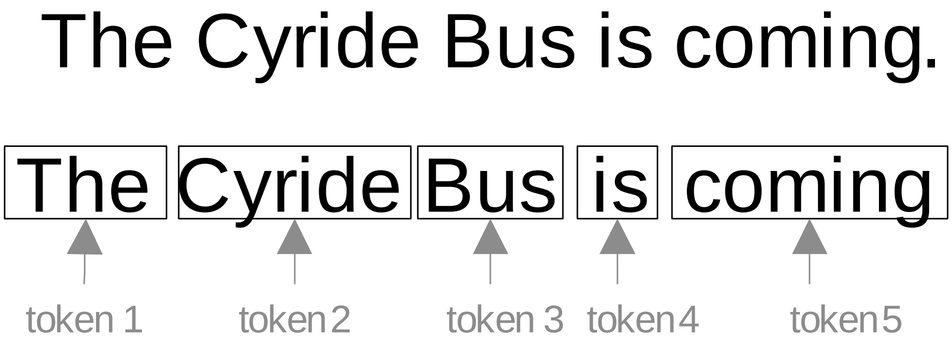


Figure 5.1 An example of Tokenization

chops it up into pieces, called tokens, at the same time throwing away certain characters, such as punctuation [101]. A **token** is an instance of a sequence of characters in some particular document that are group together as a useful semantic unit for processing. Extracting token is not trivial [101] and should deal with natural language identification, hyphenation and apostrophe processing, compound splitting, and word segmentation. More details can be found in [101].

5.2.1.2 Stop Words

Extremely common words is of little value in document representation, which should be excluded from the vocabulary entirely. These **stop words** stored as a **stop-list** which is often hand-filtered based on their semantic content relative to the domain of the document collection. Some frequently used stop words are ”a, an, and, are, as, be, is, the, these,...”. An effective stop-list can significantly reduce the size of the VSM.

5.2.1.3 Normalization and Stemming

Token normalization is the process of canonicalizing tokens so that matches occur despite superficial differences in the character sequences of the tokens. Normalization includes removing characters like hyphens(such as $anti-discriminatory$ to $antidiscriminatory$), dealing with synonyms(such as car and $automobile$), accents and diacritics, capitalization and case-folding [101]. Stemming transformed different forms of a word into a $term$ based on grammatical rules. For instance:

$$am, are, is \rightarrow be$$

$$eye, eyes, eye's, eyes' \rightarrow eye$$

The most common and empirically effective algorithm for stemming English is *Porter's algorithm* [101] which applies five steps of word reductions sequentially. Each step has its own rules. For example, the rules for suffix in the first step are

Rule
$SSES \longrightarrow SS$
$IES \longrightarrow I$
$SS \longrightarrow SS$
$S \longrightarrow$

Now we have three different concepts, *token, type and term*. A **type** is the class of all tokens containing the same character sequence. A **term** is a type that is included in the VSMs. For example, if a document has only one sentence:*to be or not to be, that is a question*, there are 10 tokens, 8 types. If we omit the stop words *that, is,a*, we have 5 terms: *to,be,or,not,question*.

5.2.2 Document Representation

The choice of a presentation of documents depends on what one regards as the meaningful units of text and the meaningful natural language rules for the domination of these units. In term-document approach, each document d_i are denoted as a vector of term weights $\overrightarrow{d_i}' = [w_{1,i}, w_{2,i}, \cdots, w_{n,j}]$ where n is the total number of distinct terms in the corpus. Each distinct term has an index in the vector.

There are different ways to define terms and term weights. In the case of considering words as terms, the document presentation is called *bag of words* where the order of the terms are arbitrary. Some authors have used *phrases* instead of words as terms to catch the semantic of the document more accurately. But the experiments showed that more sophisticated representations do not yield significantly better effectiveness [130]. The likely reason for the discouraging results is that, although indexing natural languages based on semantic phrases have superior semantic qualities, they have inferior statistical qualities, with respect to word-only approaches [130]. Therefore, in spite of the crudeness of term-document representation, it works surprisingly well; vector seem to capture an important aspect of semantics [149].

Each term in a document has a weight. As a simplest way, binary weights can be used where 1 denoting presence and 0 absence of the term in the document. However, this approach deems all words in the documents are equally important. Clearly this is not reasonable. For example, stop words do not contribute in any way to the semantic of the documents. The most popular weighting scheme is *tf-idf*. A term frequency tf_{t,d_i} denotes that the term *t* occurs tf_{t,d_i} times in document d_i. A document vector is viewed as a quantitive digest of that document. Using raw term frequency suffers from a critical problem: a term which occurs in all the documents in a collection has no or little power in discriminating the document from another even though its occurrence in any document is high. For example, a corpus on the university is likely have the term *student* in almost every document. We would like the few documents which contains a term *t* get a higher weight for the term *t* with respect to those documents. To this end, a new concept *document frequency* df_t is defined to be the number of documents in the corpus which contains the term *t*. The *inverse document frequency(idf)* of a term *t* is defined as:

$$idf_t = log\frac{N}{df_t}$$

where *N* is the total number of documents in a collection. Thus the idf a rare term is high, whereas the idf of a frequent term is low. The *tf-idf* weighting scheme assigns to term *t* a weight in document *d* is:

$$tfidf_{t,d} = tf_{t,d} \times idf_t = tf_{t,d} \times log\frac{N}{df_t}$$

That is, $tf\text{-}idf_{t,d}$ is high when *t* occurs many times within a small number of documents; lower if the term occurs fewer times in a document and occurs in many documents; lowest if the term occurs in all documents evenly. For each document vector, those terms do not occur in a document, the weights are zeros. Since the total number of terms in a corpus is usually much larger than the number of term occurring in a document, the vector for a single document is sparse. Some other weighting scheme other than *tf-idf* can be found in [101]. The $tf\text{-}idf$ of the example in section C.1 is shown in Table C.2. The vectors for the documents are shown in Table C.3. We can see that most of the elements in the vectors are zeros.

5.2.3 Similarity of VSM Vectors

The documents in a corpus can be represented as a set of vectors in a vector space, where there is one axis for each term. For a corpus with totally n distinct terms, these vectors can be deemed as the points in an n-dimension space. The choice of the similarity measure is an important consideration. Although various measures can be used to compute the distance between two point, the most desirable distance measure is one for which a smaller distance between two objects implies a greater likelihood of having the same class [160]. For example, if kNN is being applied to classify documents, it may be better to use the cosine measure rather than Euclidean distance [160]. One popular way to quantifying the similarity between two documents d_i and d_j is to defined as following:

$$cos(\theta) = cos(\overrightarrow{d_i}, \overrightarrow{d_j}) = \frac{\overrightarrow{d_i} \bullet \overrightarrow{d_j}}{|\overrightarrow{d_i}||\overrightarrow{d_j}|} = \frac{\overrightarrow{d_i}}{|\overrightarrow{d_i}|} \bullet \frac{\overrightarrow{d_j}}{|\overrightarrow{d_j}|} = \widetilde{d_i} \bullet \widetilde{d_j}$$

where $\bullet$ is the dot product between two vectors, $|\overrightarrow{d_i}| = \sqrt{\sum_{k=1}^{k=n} \overrightarrow{d_i}(k)^2}$, $\widetilde{d_i} = \frac{\overrightarrow{d_i}}{|\overrightarrow{d_j}|}$ is called a *normalized* vector. Normalizing vectors can make us focus on the direction of the vector without considering their lengths. In other words, the cosine of the angle θ between two vectors is the inner product of the two normalized vectors. If $\overrightarrow{d_i}$ and $\overrightarrow{d_i}$ are frequency vectors for terms, a long document will have a vector where each term has high frequency. Thus, for example, a long document $\overrightarrow{d_i}$ will have a vector with large length while a short document $\overrightarrow{d_j}$ with a short vector. The $\overrightarrow{d_i} \bullet \overrightarrow{d_k}$ will be much larger than $\overrightarrow{d_j} \bullet \overrightarrow{d_k}$ even if the similarity between $\overrightarrow{d_i}$ and $\overrightarrow{d_k}$ is almost the same as the similarity between $\overrightarrow{d_j}$ and $\overrightarrow{d_k}$. Cosine captures the idea that the length of the vectors is irrelevant and the angle between the vectors is critical [149].

The cosine value ranges from -1(θ is 180 degree) when the vectors are in opposite directions to +1(θ is 0 degree) when they point in the same direction. When the vectors are orthogonal(θ is 90 degree). With raw tfidf scheme, there will be no negative cosine values. However, with the feature selection discussed later negative cosine results can be generated when using truncated Singular Vector Decomposition(SVD). Without considering the negative results, the smaller the value of θ, the more similar is between the two normalized vectors. The similarities between the documents in the example of section C.1 are shown in Table C.4. We can see that similarity score between a vector and itself is 1. The testing document *t1* is most similar to the training document *d3*.

Table 5.1 A Example of Term-Document Matrices [80]

Collection	Dimension	Nonzeros	Density
Web Document	10000×255943	3.7M	0.15%
NSF Abstracts	94481×6366	7M	1.16%
Face Images	36000×2640	5.6M	5.86%

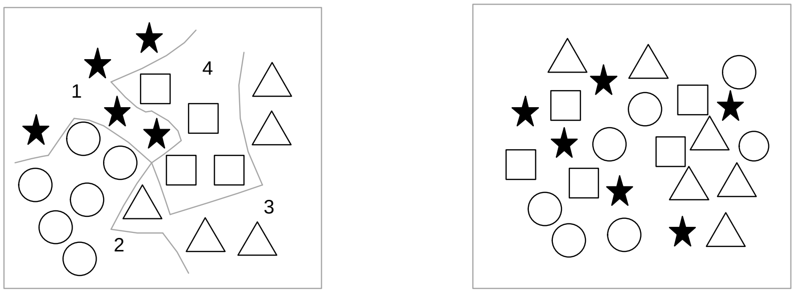
5.2.4 VSM and Sparse Matrix

Even after feature reduction, the dimensions of the vectors may still be in the order of thousands. Computing the similarity between all documents is a computation and data intensive problem. Suppose we have totally n documents and m terms in a document collection. The running time computing the similarity between all documents is $O(mn^2)$. However, the vectors are usually sparse when presenting documents(see section 5.2.2). If the vectors of all documents in a collection comprise a matrix, specific techniques taking advantage of the sparseness can be used to accelerate the computation speed. From section 5.2.3, we have:

$$cos(\overrightarrow{d_i}, \overrightarrow{d_j}) = \frac{\overrightarrow{d_i} \bullet \overrightarrow{d_j}}{|\overrightarrow{d_i}||\overrightarrow{d_j}|}$$

$\overrightarrow{d_i} \bullet \overrightarrow{d_j}$ is only calculated on the nonzeros elements which will significantly reduce the computation cost. Determining the nonzero elements can easily be achieved by building an inverted index for the coordinates. For large scalable computing, the sparse matrix can not fit into memory. Distributed implementation using MapReduce is implemented [149]. Mapreduce is proposed in [39] and provides an easy-to-understand programming model for designing scalable, distributed algorithms. MapReduce is a practical solution to large data problems today and distribute computations across multiple machines. In traditional parallel programming model, the developers shoulder the burden of explicitly managing concurrency and manage system-level details. MapReduce is composed two parts: mappers and reducers. Mappers take the input and generates the intermediate key-value pairs. Reducers aggregate the output values with the key. The developer can only focus on the mappers and reducers design. [95] described algorithms for computing pairwise similarity with MapReduce on document collections. Some other randomized techniques to improve the computational efficiency are discussed in [149].

The matrices arising in data mining applications are quite large, irregular and closer to a random pattern [80]. For example, the term-document matrices used in [80] are 10000×256000, 94481×6366



(a) Contiguity Hypothesis hold

(b) Contiguity Hypothesis not hold

Figure 5.2 Contiguity Hypothesis of two-dimension VSM

and 36000 × 2640 respectively as shown in Table 5.1. We will discuss to use SMVM to improve the performance of VSM later in this chapter. Sparse matrix-vector multiplication on data mining matrices has been used in text retrieval and face recognition for algorithms like Latent Semantic Indexing, Concept Decomposition, and Eigenface Approximation [80]

5.2.5 VSM and Classifier

The basic hypothesis in using the vector space model for classification is the *contiguity hypothesis* [101].

Definition 6 (**contiguity hypothesis**). *Documents in the same class form a contiguous region and regions of different classes do not overlap.*

Whether or not the contiguity hypothesis can hold depends on the choices for the document representation, the weighting scheme, stop-list, feature reduction techniques, and so on. If the document representation is not appropriate, the contiguity hypothesis will not hold and the classifiers based on the contiguity hypothesis will not be effective. In Figure 5.2, each two-dimension vector is represented by a block. We can draw boundaries that separate the 2-dimension vectors in Figure 5.2(a) while there is no cross among boundaries. We can not draw a bounding box which only contains the vectors of the same class in Figure 5.2(b). Therefore, the contiguity hypothesis does not hold for Figure 5.2(b).

The classifiers partition the vector space into regions separated by linear decision hyperplanes are called **linear classifiers**. More complex nonlinear models are not systematically better than linear

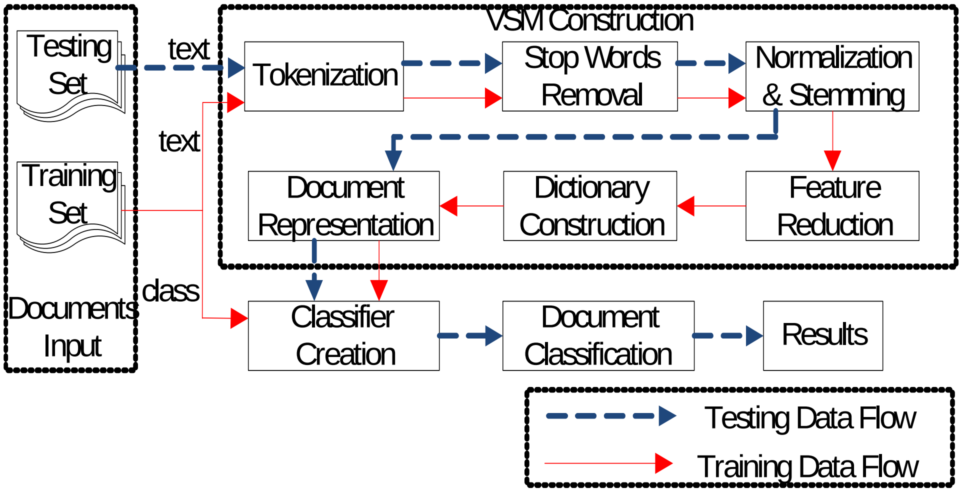


Figure 5.3 Architecture of Text Classification

classifiers. Nonlinear classifiers have more parameters to fit on a limited amount of training data and are more likely to make mistakes for small and noisy data sets [101].

5.3 Architecture of Text Classification

Text classification comprises three phases: vector space construction, classifier creation, and document classification. In vector space construction phase, the linguistic processing is performed on each document. Classifier creation is built upon the training set under the inductive learning hypothesis described in section C.2. Document classification classifies each test document using the classifier. The overview architecture of document classification is shown in Figure 5.3 where the solid arrows represent the training data flow and the dotted arrows represent the testing data flow.

5.4 K-Nearest Neighbor Algorithm

5.4.1 Classifier Design

The input of the classifier design is a training set of documents with known classes and the output is a classifier Γ defined in section 5.1. The existing classifier construction algorithms are based on the inductive learning hypothesis(see section C.2). Based on the approaches that tackle the inductive learning hypothesis, we have probabilistic classifiers, decision tree classifiers, decision rule classifiers, regres-

Algorithm 11 Classifier Construction of kNN Algorithm

```
**Input:** the documents and classes pairs in the training set $R$, $R \subseteq C \times D$
**Output:** VSM, $\widetilde{D}$; the value of $k$
  Construct the vector set $\widetilde{D}$ for $D$
  Select an appropriate value for $k$
```

sion methods, on-line methods, the Rocchio method, neural networks, example-based classifiers,support vector machines(SVM) which is one of the top 10 data mining algorithms [160], and classifier committees [130]. Naive Bayes classifier is a probabilistic classifier. Linear Least-Squares Fit(LLSF) is based on regression method. The k nearest neighbor(kNN) algorithm is a famous example-based classifier. Adaboost which is a boosting algorithm based on a committee is also one of the top 10 data mining algorithm [160]. Among these classifiers, boosting-based classifier committees, support vector machines, example-based methods, and regression methods deliver top-notch performance [130]. In this chapter, we focus on kNN classifier which is one of the state of the art TC algorithms. Details of SVM can be found in [71, 81].

5.4.2 Algorithm Description

kNN is one of the top 10 algorithms in data mining [160] and based on the contiguity hypothesis in that a test document has the same label as the training documents located in the local region surrounding it. kNN is a similarity-based learning algorithm [161]. Given a test document d_t represented as a vector of term weights and a chosen similarity metric, the kNN algorithm finds its k nearest neighbors by means of calculating the similarity of d_t to all documents in training set, which forms a neighborhood of d_t. Majority voting among the classes of document in the neighborhood is used to decide the class of d_t. For example, 1NN assigns each document to the class of its closest neighbor. However, 1NN is not robust. Experiments [161] showed that the value of k range from 30 to 50 can achieve the optimal results.

The training algorithm is shown in Algorithm 11 in which the vector space model is constructed and the value of k is determined. Compared with other TC algorithms, kNN uses a *lazy training* approach. The training phase is simple and does not have an off-line learning phase. The categorization algorithm

Algorithm 12 Classification Process of kNN Algorithm

```
Input: the document to be classified, $d_k$
Output: the class label of the testing document, $C_k$
  Construct vector $\widetilde{d_k}$ for the document, $d_k$
  for all $\widetilde{d_i} \in \widetilde{D}$ do
    $\cos(\theta_i) \longleftarrow \widetilde{d_k} \bullet \widetilde{d_j}$
  end for
  Construct $D' \subseteq D, |D'| = k$, where $d_i \in D'$,$d_j \in D - D'$, $cos(\theta_i) > cos(\theta_j)$
  for all $c_i \in C$ do
    $score_{c_i} \longleftarrow \sum_{d_j \in D', <d_j, c_i> \in R} cos(\widetilde{d_k}, \widetilde{d_j})$
  end for
  Classify $d_k$ as $C_k$ based on $score$
```

is shown in Figure 12. For each incoming document, a vector is built from it. The similarity between it with each document in the training set is computed. The closest k neighbors are collected. The score for each candidate class is computed based on the cosine values. Based on the type of classification, different policies can be used to determine the label of the incoming document. In the single label classification scenario, the incoming document is assigned to the class with the highest score. In the multi-label scenario, the classes whose score is higher than a threshold are assigned to the incoming document. [72] presents a Weight Adjusted k-Nearest Neighbor(WAKNN) classifier. Some other kNN variations can be found in [155, 167, 70, 134].

5.5 Motivation

Understanding and implementing kNN classifier is easier when compared with other classification techniques. Despite of its simplicity, its effectiveness is very high. Experimental result shows that the error of the nearest neighbor rule is bounded above by twice the Bayes error. kNN is particularly suitable for multi-label, multi-class text classification. Although most researchers believe that SVM is better than kNN and kNN better that Naive Bayes(NB), the ranking of classifiers ultimately depends on the class, the document collection, and the experimental setup [101]. A good feature selection method enables KNN to surpass SVM's performance [127]. For the assignment of functions to genes based on expression profiles, some researchers found that kNN outperformed SVM [160].

kNN is a *lazy learner*, but a *hard worker*. That is, building the model is cheap. However, classifying unknown objects is expensive. From the experimental result in [163], the testing time of kNN is 12 times larger than that of SVM though achieving better effectiveness. WAKNN [72] running time varied from a few hours on 300 to 500 training sample size to a few days on 1000 training examples. The computation cost increases with the increase of the size of training set. To employ kNN in a large scale and real-time classification task, its classification speed must be significantly improved. Current work on kNN algorithms mainly focus on improving the classification speed [26].

[109] described a Compressed Sparse Row(CSR) SMVM implementation on the Convey HC-1 reconfigurable computer. The architecture is optimized for CSR-formatted sparse matrix data. As far as we are aware, there is no previous work on accelerating kNN for text classification using FPGA hardware. An accelerator for k-th nearest neighbor thinning problem based on FPGA is proposed in [129]. Targeting at image retrieval and video object tracking, a fast k nearest neighbor search implementation using GPU is described in [57]. An FPGA implementation of kNN classifier used in the wavelet domain is proposed in [165].

5.6 Experimental Setup

The performance bottleneck of kNN is the high computation cost in the classification stage. To improve the efficiency of kNN, the classification component of kNN algorithm should be put into hardware. In Figure 5.3, all the modules are implemented in software except the part of the classifier. The software toolkit in [44, 43, 55] is used to create VSMs from text documents. The 20 news groups data set [92] is a collection of 18846 newsgroup documents, partitioned nearly evenly across 20 different newsgroups. The documents are sorted by date into training set(11314 files) and test(7532 files) set. During the training stage, the vectors created from the documents in the training set form a sparse matrix. During the classification stage, a testing document d_i is sent to the system which creates a vector $\tilde{d_i}$ for d_i. To compute the similarity between d_i and other documents in the training set, $\tilde{d_i}$ is sent to the SMVM engine as the vector input. The sparse matrix is sent to the SMVM engine as the matrix input. SMVM Engine in Figure 4.5 performs the Sparse Matrix-Vector Multiplication and returns the results to the software. The software finds the k nearest neighbors of d_i and assigns categories to it.

The implementation platform is convey HC-1 as shown in Figure 5.4. It is composed of one host

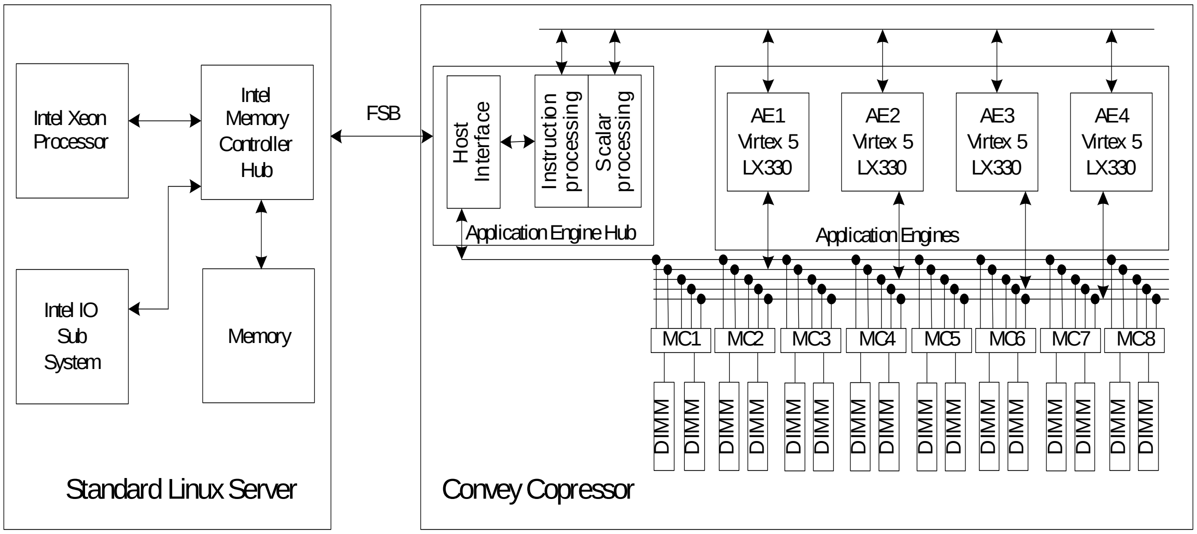


Figure 5.4 Convey HC-1 Architecture

X86 motherboard and one coprocessor board which are connected by a FSB mezzanine card. The motherboard includes a 2.13GHz Intel Xeon 5138 dual-core host CPU and standard Intel I/O chipset. The host CPU is equipped with 667MHz Fully-Buffered Dual In-line Memory Modules(FBDIMMs)(totally 24GB) providing a theoretical bandwidth of 8 GB/s. There are four Virtex-5 LX330 FPGA chips called Application Engines(AEs) and an Application Engine Hub(AEH) on the coprocessor board [2]. The AEH contains two FPGAs, one of which serves as the host interface to the host board. The other one is a soft-core processor called ”scalar processor” which invokes computations on the AEs. The scalar processor plays the role of co-processor of host processor. The Memory Controller(MC) Interface gives the AEs direct access to coprocessor memory. Each of the 4 AEs runs at 150 MHz and is connected to each of the 8 memory controllers through a DDR interface. The link between the AE FPGA and the MC runs at 300 MHz, but in order to ease timing in the FPGA, the 300 MHz interface is converted into two 150 MHz memory ports to/from the AE personality. Each MC physically connects to two Scatter-Gather-DIMMs(SGDIMMs). The communication bandwidth between each AE and MC connection is 2.5GB/s. Two 64-bit words are provided in each clock cycle for each connection. Each DIMM has a 5GB/s link to its MC. In the ideal scenario, the peak IO bandwidth of each PE is 20GB/s if there is no read conflict. The aggregate bandwidth of all PEs is 80GB/s. To fully utilize the memory bandwidth, a request must be made from every AE to every MC port on every cycle unless stalled by the MC

Table 5.2 Characteristics of VSM Matrices Built from Training Set and Testing Set

	Documents Number	**Terms Number**	**Matrix Dimension**	**NonZeros Number**	**Density**	**RunTime (seconds)**
Training	11314	94439	11314 × 94439	1033094	0.097%	14
Testing	7532		7532 × 94439	634413	0.089%	9

interface.

5.7 Implementation and Result

5.7.1 Dataset Processing

The software toolkit in [44, 43, 55] is used to translate a text document into a sparse vector. However, it does not differentiate the train dataset from the test dataset. We revised the toolkit so that different actions are taken in training and testing stages when a new word in test documents is encountered. In the training stage, each new term is inserted into the term dictionary. In the testing stage, each term is searched in the dictionary established in the training stage. If the term is found, the weight of the term is returned. Otherwise, the term is discarded. The characteristics of the VSM matrix built from the training and testing documents is listed in Table 5.2. Because all terms are created in the training stage, the number of terms in the testing set is equal to the number of terms in the training set. The toolkit translates each document into a normalized vector using *tf-idf* scheme. The vectors constitute the training matrix M_{train} and testing matrix M_{test}. Each row of the matrices corresponds to a document. The toolkit runs in the linux server(Figure 5.4) of the Convey HC-1 system. The runtime for training set includes reading training data from the hard disk, establishing the term dictionary, translating the training documents into sparse matrix and writing the matrix into hard disk in Compressed Column Storage(CCS) format. The runtime for testing set includes reading testing data from the hard disk, searching the term dictionary, translating the testing documents into sparse matrix and writing the matrix into hard disk in CCS format.

5.7.2 SMVM in Software

kNN(k=1) is used to the classify to testing documents. According to Algorithm 12, each row in the M_{test} is multiplied with all rows in the M_{train}. The label of the train document which has the maximum sum-of-product with the test document is set as the label of the test document. To label all test documents, $M_{test} \times M^{'}_{train}$ is computed where $M^{'}_{train}$ is the transpose of M_{train}. That is, the inner product of each row in M_{train} and each row in M_{test} is computed. The matrices are stored in CCS format. All non-zero values are stored in a value vector val. The algorithm to compute the sum-of-products is shown in Algorithm 13. The two matrices are firstly loaded into the memory. The first non-zero of each row i has an index $row\_start[i]$ in val and are computed afterwards. The product of two non-zeros(one is from the i^{th} row of M_{test}, the other is from the j^{th} row of M_{train}) is added to $Sum\_of\_Product(i, j)$ only when they have the same row and column index. The runtime to load the matrix data from the hard disk to memory is around 1.2 seconds. The runtime of the algorithm is around 39.8 seconds.

5.7.3 SMVM in Reconfigurable System

The on-chip memory in FPGAs is not large enough to store the matrix and vector data. To do the classification in FPGAs, the data is first loaded into co-processor DIMMs. Our goal is to align the data so that the memory throughput and SMVM computation performance on the Convey machine for text classification is maximized. Each AE takes two 64 bit values from each MC in one clock cycle. Therefore, each AE takes 16 64-bits values from all 8 MCs in each clock cycle. This determines that the number of multipliers in the SMVM engine is at most 8 and the number of SMVM engines in each AE is 1. The 16 double-precision floating-point numbers serving as the inputs of the SMVM engine are from the matrix and vectors in each clock cycle. To use SMVM engine, the data in DIMM is aligned in such a way that the successive values can be taken from all MCs in a single cycle. Because SMVM is a data-intensive computation, maximizing memory throughput will also maximize computation performance.

Multiple vectors are to be multiplied with the sparse matrix. However, we send each vector only to one SMVM engine. This will simplify the processing outputs from each engine. We let MC1-4 store the matrix data, IPVs, column indices and MC4-8 store the vector data, IPVs, column indices. Matrix data, Algorithm 13 Classification Process of kNN Algorithm

```
Input: $M_{test}$, $M_{train}$
Output: the class label of the testing documents
  Read $M_{train}$;
  Compute $row\_start$ of $M_{train}$;
  Read $M_{test}$;
  Compute $row\_start$ of $M_{test}$;
  for each row $i$ of $M_{test}$ do
    for each row $j$ of $M_{train}$ do
      $Sum\_of\_Product(i, j) \leftarrow 0$, $index_{test} \leftarrow 0$, $index_{train} \leftarrow 0$;
      while $test\_val[index_{test}].row\_index$ = $i$ and $train\_val[index_{train}.row\_index]$ == $i$ do
        if $test\_val[index_{test}].column\_index$ = $train\_val[index_{train}].column\_index$ then
          $Sum\_of\_Product(i, j)$ += $test\_val[index_{test}] \times train\_val[index_{train}]$;
          $index_{test}$ ++;
          $index_{train}$ ++;
        else if $test\_val[index_{test}].column\_index < train\_val[index_{train}].column\_index$ then
          $index_{test}$ ++;
        else
          $index_{train}$ ++;
        end if
      end while
    end for
  end for
```

vector data, IPVs, column indice have continuous addresses in DIMMs. Each type of data has a start address. The data storage pattern in MCs is shown in Figure 5.5. Note that each vector is duplicated four times. Each non-zero is 64 bits and has a column index which is 32 bits. The total number of non-zero is 1033094. The total number of IPVs is $\frac{1033094}{8} = 129137$. The IPVs and column indices are evenly distributed in MCs while the non-zeros are evenly stored in MC1 to MC4. Each vector has a copy in MC5 to MC8, thus duplicated four times. Each vector has 94439 columns. Each element of it is 64 bits. Different DIMM memory spaces are allocated for non-zeros in the sparse matrix, vector data, column indices and IPVs. 8 Mega-Bytes of memory is allocated for sparse matrix, 4 Mega-Bytes for column indices and 256 Kilo-Bytes for IPVs. Because zero values in vector data need be stored in DIMMs, 1 Giga-Bytes of memory are allocated in each MC(from MC5 to MC8). That is, each MC has 1GB memory(128M 64-bit words) to store the vector data. 128K words of memory is allocated for each vector. Therefore, 1K(1024) vectors are to be stored in the 1GB memory. There are totally 7532

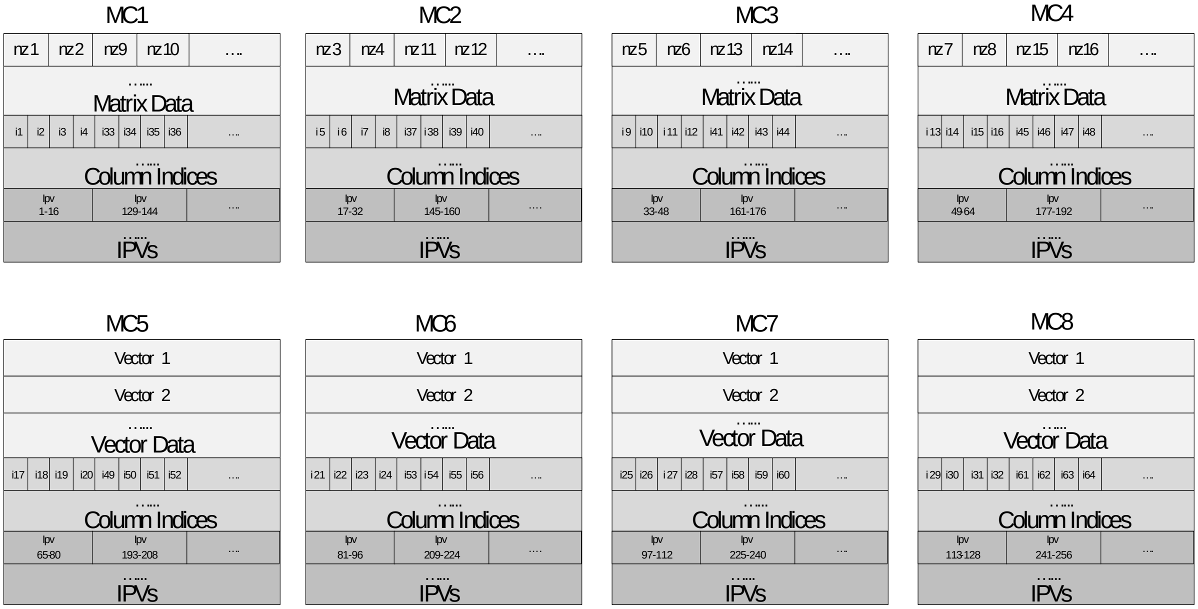


Figure 5.5 Data Storage Pattern in MCs

vectors. We need 8 rounds to load the data into DIMMs and perform the computation. It will take long time to set 1GB memory in each MCs. Note that most of values in a vector are zeros. We initialize all words to be zero when the memory is allocated. In each round, only the non-zeros in the vectors are loaded into the specified memory cell based on their column indices. After the computation, only the memory cells which have a non-zero value will be set back to zeros. In this way, the time of loading vector data is greatly reduced. The time to load the sparse matrix, column indices, vector data and IPVs is around 6.9 seconds not including the time to allocate and initialize the memory space.

Each AE has an input state machine which retrieves data from DIMMs and sends it to the SMVM engine. The state machine has the start address of Matrix data, IPVs and column indices. Each matrix number is 64 bits. Each IPV is 8 bits because the number of multipliers in the SMVM engine is 8. Each column index is 32 bits which means the number of columns in the matrix and vector is at most 2^{32}. Each non-zero in the sparse matrix has its column index in the DIMMs and multiplied with a vector element which has the same column index in the vector. Note that the corresponding vector element may be zero because the vector in text classification is sparse. In order to ease the memory access, we store all vector values including zeros into the DIMMs. The input state machine has three

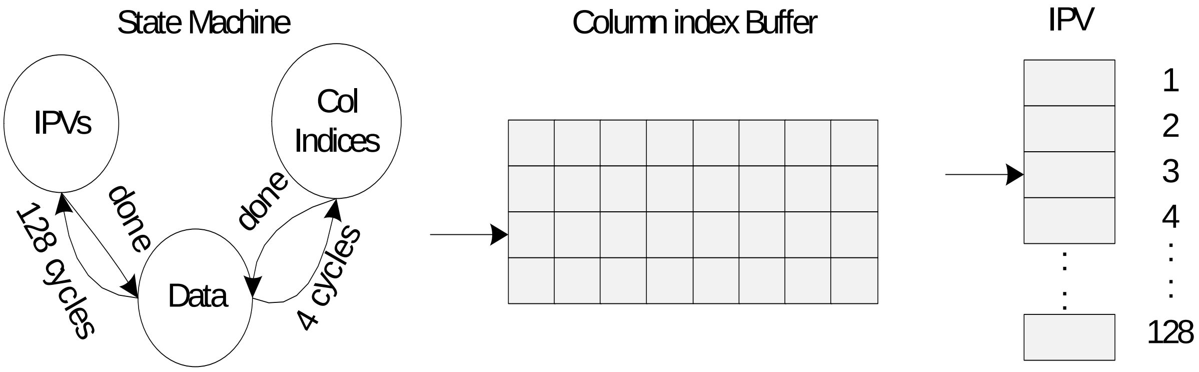


Figure 5.6 Input State Machine

states: reading IPVs, reading matrix and vector data and reading column indices. The state transition is shown in Figure 5.6. In one clock cycle, each AE will receive $\frac{64\times16}{8} = 128$ IPVs in the *reading IPVs* stage, $\frac{64\times16}{32} = 32$ column indices in the *reading column indices* stage, 8 matrix numbers and 8 vector numbers in the *reading data* stage. In one clock cycle, eight column indices are used in the *reading data* stage. Therefore, the machine need read the column indices every four clock cycles. In the input state machine, there is an IPV storage buffer, a column index buffer and two counters. The counters point to the data currently used. In some clock cycles, the input state machine sends 8 matrix numbers, 8 vector numbers and an IPV to the SMVM engine. However, this will not happen in each clock cycle because the input state machine need read IPVs every 128 clock cycle, and read column index every 4 clock cycles. Considering in 644 consecutive clock cycles, 128 of them are in *reading column indices* stage, 4 of them are in the *reading IPVs* stage, and 512 of them are in the *reading data* stage. That is, the SMVM engine has valid input data in 512 clock cycles. The computation resource usage is $\frac{512}{644} = 79.5\%$. The computation performance of each AE in terms of FLOPs(see chapter 4) is $2k \times f_{clock} \times 79.5\% = 2 \times 8 \times 150MHz \times 79.5\% = 1.908$ GFLOPs. Since we have four AEs, the total computation performance is 7.632 GFLOPs. The computation time of the data in Table 5.2 is around $\frac{7532\times1033094\times2}{7.632\times10^9} = 2.039$ seconds(Each number has two floating-point operations). The computation time is $\frac{39.8}{2.039} = 19.519 \approx 20$ times less than the software. The memory bandwidth usage is almost 100%.

We compare the text classification runtime in software with that of in Convey in Figure 5.7. Both

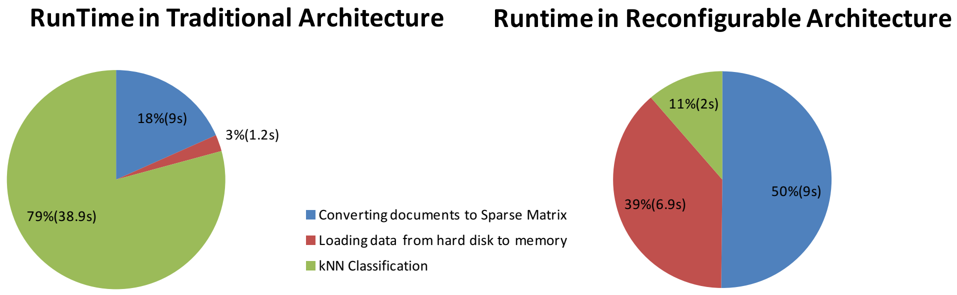


Figure 5.7 RunTime Comparison of Text Classification

approaches uses the same toolkit to translate the documents into sparse vectors. In the software approach, the data is loaded into the main memory. In the reconfigurable approach, the data is loaded into the coprocessor memory in a pre-designed pattern which takes longer time than the software approach. The computation of FPGAs is much faster than the software approach. Putting them in all, the reconfigurable approach is around 2.8 times faster than the software approach.

5.8 Contribution

We deployed the SMVM engines into multiple FPGA chips of Convey HC-1. Text documents are represented as large sparse matrices using Vector Space Model(VSM). The k-nearest neighbor algorithm uses SMVM to perform classification on multiple FPGAs. We interleaved the data in the DIMMs to maximize the memory throughput, thus maximized the computation performance. Our experiment shows that the classification time using FPGAs is greatly reduced compared with the traditional computing architecture.

CHAPTER 6. HPRC ARCHITECTURE FOR DATA-INTENSIVE APPLICATIONS

MapReduce is a widely used program model for data-intensive application which simplifies the design of large distributed computing system. FPGA is notorious for the programming productivity and has been out of reach for many domain experts. The programmer using hardware language such as VHDL, Verilog usually spent months to develop a module. The same module can sometimes be accomplished in several lines of C/C++ code. Even though the performance of FPGA-based design is much higher, the shortcoming mentioned above is a great hindrance to the popularity of FPGAs. There are some C-to-HDL tools like ROCCC, Impulse C or Handle-C available, however, the viability and translation efficiency for practical scenarios still need be further explored [156]. We propose a pipelined and high performance framework for FPGA design based on the MapReduce model. Our goal is to lessen the FPGA programmer burden while minimizing performance degradation.

6.1 Introduction

MapReduce is a programming model proposed by Google for processing data-intensive applications [96]. The original implementation [39] of the model simplifies the development of larger-scale applications on clustering machines(PCs). Since many practical problems can be expressed in this model, MapReduce is widely used in industry. For example, hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs run on thousands of machines in Google. Tens of companies, such as Yahoo!, Facebook and Adobe use Hadoop [1] which is an open-source implementation of MapReduce.

In essence, MapReduce divides a job with massive amount of data into many discrete tasks that are executed in parallel on clusters of servers. The MapReduce work-flow on a cluster of commodity machines is depicted in Figure 6.1. The commodity machines are connected together by high speed

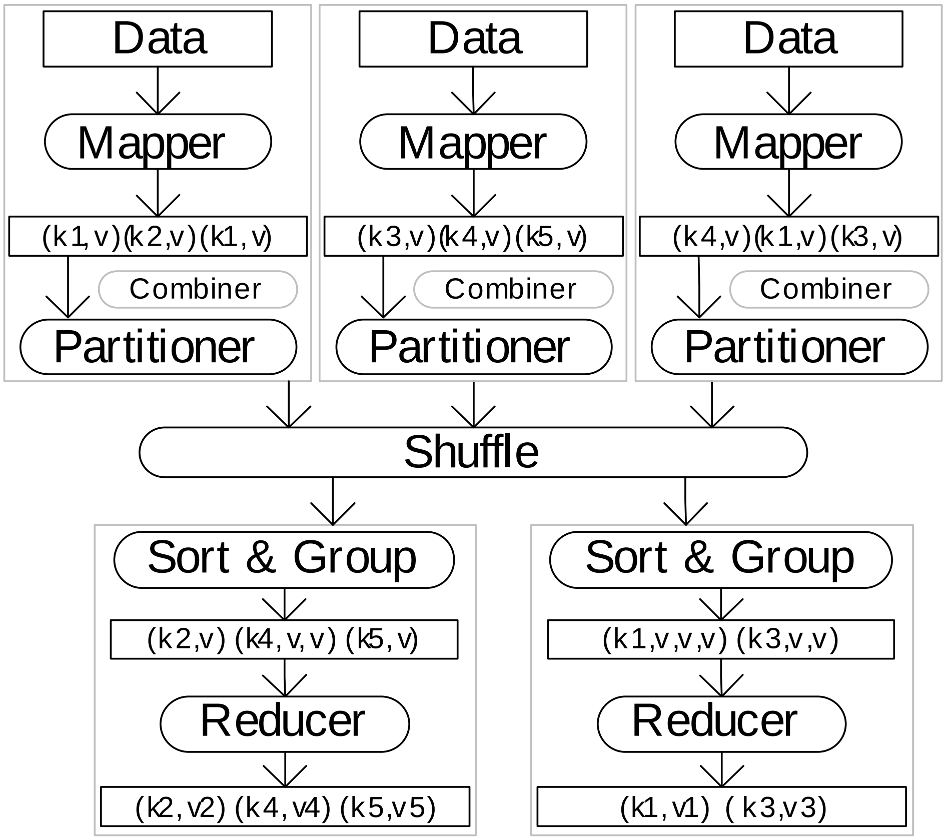


Figure 6.1 An Example of MapReduce Work Flow

network link,such as Ethernet. The MapReduce Framework is mainly composed of map and reduce stages. Each distinct section of input data is located closely to mappers. The mappers translate the data into a massive amount of intermediate (key, value) pairs. These pairs are usually stored in local disk and reduced by an optional combiner before they are sent to the reducers in the shuffle stage. In this way, the number of pairs sent on the network is greatly diminished. Otherwise, the network bandwidth will be devoured and become the performance bottleneck of the Framework. The pairs with the same key are sent to the same reducer. Each reducer works in a distinct key space. Partitioners are responsible for dividing up the intermediate key space and assigning the (key, value) pairs to reducers. When a reduce worker has read the intermediate pairs, it sorts and groups those pairs by keys. The reduce worker then iterates on each pair. For each unique key encountered, the set of intermediate values are sent to the reducer. Both the mapper and reducer code are written by the user. Note that a commodity machine may have multiple mappers and reducers. Algorithm 14 is the pseudo-code to count the number of

Algorithm 14 MapReduce Program Example:counting the words in a corpus [39]

```
Map(String key, String value)
//key: document name
//value: document contents
for each word w in value do
    EmitIntermediate(w, 1);
end for

Reduce(String key, Iterator values)
//key: a word
//values: a list of counts
int result = 0;
for each v in values do
    result += v;
end for
Emit(key,result)
```

occurrences of each word in a large corpus [39].

In MapReduce model, the programmers only need write the mapper and reducer code in the predefined style. The MapReduce system handles the underlying operations. For those applications running on clusters of commodity machines, the run-time system is responsible for partitioning the data, creating the MapReduce processes to machines, scheduling and synchronizing the different tasks, managing the machine failures and data redundancy(fault tolerance), alleviating the problem of stragglers, and handling the communication among machines. In this way, the technical details of large-scale parallel and distributed systems are hidden from the programmers. The development efficiency is highly improved.

The architecture of a Hadoop cluster [96] is shown in Figure 6.2. The *namenode* is the HDFS(Hadoop Distributed File System) master. The *job submission node* runs the jobtracker. The jobtracker monitors the running progress of MapReduce jobs and is responsible for coordinating the execution of mappers and reducers. The two master nodes typically run on two machines. Often times they are co-located. Each slave node runs a tasktracker which is responsible for running user code. The datanode daemon is for serving HDFS data. The tasktrackers periodically send heartbeat messages to the jobtracker. If a tasktracker is available to run tasks the return acknowledgment of the tasktracker heartbeat contains task allocation information. In scheduling map tasks, the jobtracker tries to take advantage of data locality, if possible, map tasks are scheduled on the slave node that holds the input split, so that the mapper will be processing local data.

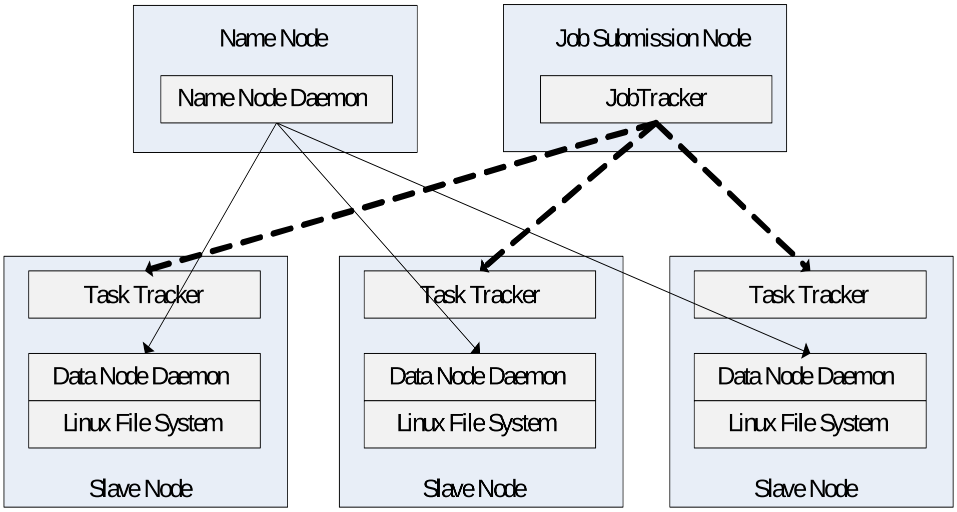


Figure 6.2 Architecture of Hadoop Cluster [96]

In this paper, we will adopt the MapReduce model into FPGA design. Our vision is to improve the efficiency of application development in FPGA. The parallelism in an FPGA chip can be used to mimic the Mapreduce model in a cluster of commodity machines.

6.2 Related Work

Since the MapReduce model simplifies the designing complexity of data-intensive applications, it has been adapted in various platforms. Google proposed the MapReduce model and implemented it in clusters with thousands of machines [39]. Apache Hadoop MapReduce is a programming model and software framework for writing applications that rapidly process vast amounts of data in parallel on large clusters of compute nodes [1]. An implementation of MapReduce for shared-memory systems, called Phoenix, is introduced in [126]. To harness GPU's power for MapReduce, a Framework called Mars is developed on an NVIDIA G80 GPU [74]. A heterogeneous system which uses parallel FPGAs and GPUs boards as MapReduce processing units is presented in [166]. There are lots of MapReduce implementations on other platforms, to name a few. An on-chip scheduled MapReduce Framework on FPGAs is proposed in [132]. The Mappers and Reducers communicate through on-chip RAMs. However, this framwork does not mention how the intermediate pairs are sorted and grouped together.

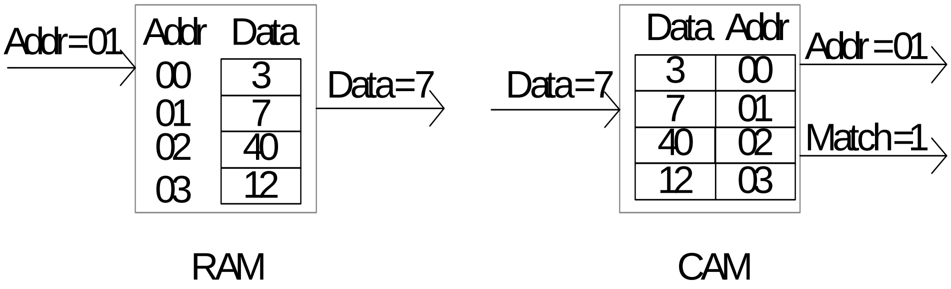


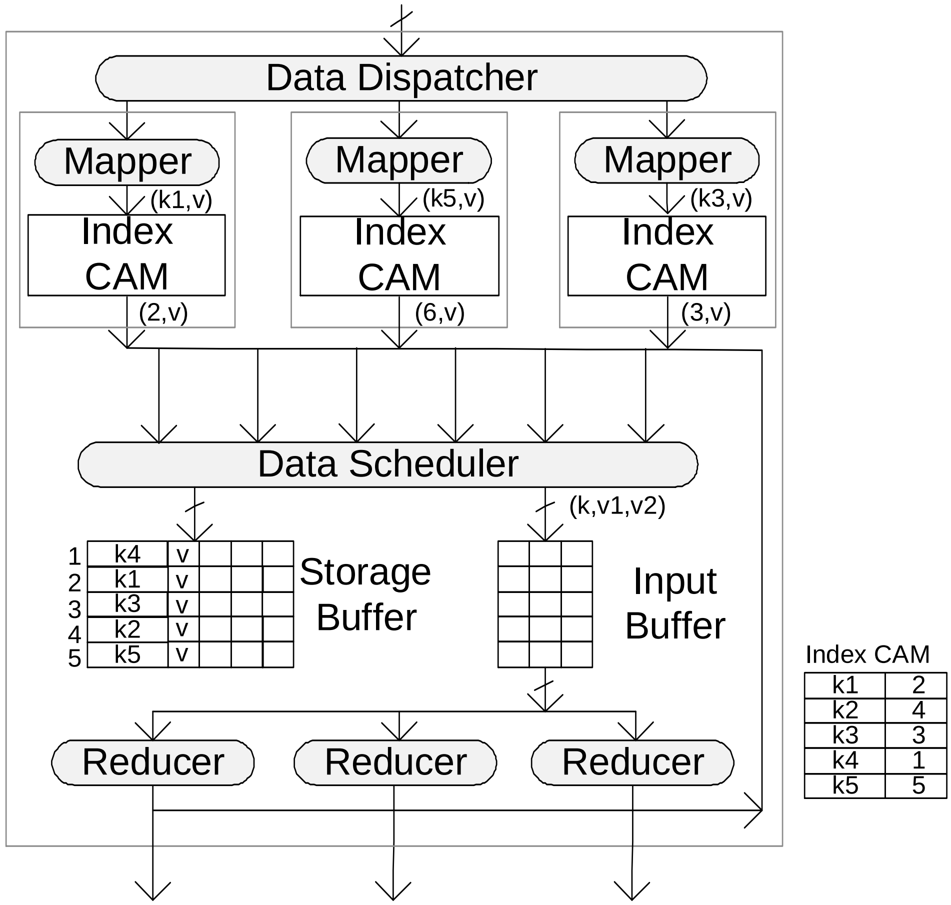
Figure 6.3 Read Mode in RAM and CAM

6.3 Content Addressable Memory

Content Addressable Memory(CAM) offers increased data search speed [7] since it allows a concurrent search of input data into the memory. The CAM outputs the corresponding address and a match signal when a match is found. The write mode in CAM is comparable to a Random Access Memory(RAM). The content of CAM can be initialized by a disk file or written dynamically during run-time. In reading RAM, an address is supplied as an index to a memory block. The data stored in that address is sent to the output after some clock cycles. In comparison, the input to the CAM is data. Once the data is found in the memory, the CAM outputs the address location in the array and the match signal is set high as shown in Figure 6.3. The CAM hardware has been available for decades. Its use can be found in cache controller for searching cache tags, network router for classifying network packets. Implementing CAM in FPGAs is not trivial. Xilinx Inc. published several papers on this topic [78, 68, 7]

6.4 Pipelined On-chip MapReduce Framework

There are several hurdles to design MapReduce framework in FPGA Chips compared with doing that in clusters. First, the on-chip memory is scarce in FPGAs and can not be used to hold a large amount of intermediate pairs. Consequently, the intermediate pairs must be reduced in time. That is, the stage barrier between the Map and Reduce stages must be removed in FPGA design. Even though the stage barrier provides programming and implementation simplicity, careful design of barrier-less MapReduce framework has equivalent generality and ease of programming and achieve higher



k1	2
k2	4
k3	3
k4	1
k5	5

Figure 6.4 MapReduce Framework on FPGAs(m=r=3)

performance [152]. The validity of barrier-less MapReduce model can be found in [152]. The proof of the equality of original and pipelined MapReduce models can be found in [35]. Second, there is no the same shared media in FPGA as the Ethernet network in clusters. And each component in the FPGA design runs in pipeline and parallel. Therefore, the shuffle of the intermediate pairs is not trivial because of the potential collisions among the mapper outputs. Distributed memory can be used for this purpose. Compared with block RAM and external memory, distributed memory is extremely fast and the cells in it can be accessed simultaneously. In contrast, only one word per clock cycle can be read or written into each port of the block RAM and external memory. However, distributed memory is usually small in size and not suitable for applications with large key space. Multiple block RAMs for each key can partially fulfilled the requirement of large key space. Third, the sort operation in FPGA is usually very expensive. Aggregating values by keys is done by sorting the values in reducers. This is infeasible when the number of keys is large. In this paper, we use Content Address Memory(CAM) to serve this purpose.

The on-chip MapReduce Framework on FPGAs is shown in Figure 6.4. The data dispatcher receives data from off-chip memory or high speed communication interface such as Front Side Bus(FSB), PCI express. It distributes the data to the mappers which are available. The Mappers process the input and generate the intermediate pairs. Each Mapper has an identical copy of index CAM which stores the memory address of each key. The pairs are sent to the index CAMs which convert the key to the storage index for each pair. In each clock cycle, the Data scheduler accepts the outputs from all the Mappers and stores the intermediate results into the on-chip memory based on the storage index in each pair. Because the on-chip memory is scarce, the intermediate results in the on-chip memory must be reduced in time. The data scheduler monitors the number of values for each key and schedules the reducers to reduce those values. The outputs of the reducers are sent back to the on-chip memory for further reduction. In the on-chip memory, each key has a buffer which stores all the intermediate values corresponding to the key. These intermediate values come from the outputs of the Mappers or Reducers.

6.4.1 Parameter Design

Let the number of mappers and reducers are m and r respectively. In each clock cycle, the mappers generate m pairs of (key, value). The data scheduler stores these pairs into the distributed memory and sends them to the reducers. To eliminate pipeline stalls, the number of pairs reduced in each clock cycle should be no less than m. On the other hand, reducing more than m pairs will lead to idle computation resources in FPGAs. Therefore, the number of pairs reduced in each clock cycle should be equal to m in the ideal case. Suppose the number of inputs of each reducer is x. The number of pairs reduced by each reducer is $x - 1$ because the inputs are merged into one (key,value) pair. Hence we have:

$$m = r(x-1) \Longrightarrow x = \frac{m}{r} + 1 \tag{6.1}$$

We can draw several conclusions from equation 6.1. First, the number of mappers should be equal to or multiple of the number of reducers. Second, the number of input values x of each reducer is at least 2. The value of x is closely related to the buffer size for each key in the distributed memory and the data scheduling policy. A larger value of x requires larger buffer size b for each key to fill up the inputs in a

Algorithm 15 Data Scheduling Algorithm

```
for each output $(k_i, v_i)$ from the mappers and reducers do
    if $\exists j$, such that $(k_i, v_j)$ is in the storage buffer then
        send $(k_i, v_i)$, $(k_i, v_j)$ to the input buffer
        remove $(k_i, v_j)$ from the storage buffer
    else
        store $(k_i, v_i)$ into the storage buffer
    end if
end for
```

reducer. To minimize the buffer size, x is set to 2 in our architecture which implies $r = m$.

6.4.2 Data Scheduling

In each clock cycle, $m + r = 2m = 2r$ outputs from the mappers and reducers are sent to the data scheduler. The $2m$ outputs together with the values in the storage buffers are scheduled as the $2r = 2m$ inputs of the reducers. The two input values of a reducer have the same key in each clock cycle. Based on the data scheduling algorithm shown in Algorithm 15, the number of values for each key in the storage buffer is at most one if there is enough number of reducers in each clock cycle. Our architecture has only $r = m$ reducers. Suppose there are totally k unique keys. Consider the worst scenario where each key has one value in the storage buffer and the input keys to the data scheduler are non-repetitive in $c = \lfloor \frac{k}{m} \rfloor$ consecutive clock cycles. In each clock cycle, there are $2m$ pairs of values which should be sent to the m reducers. Obviously, a queue should be used as an input buffer of the reducers if pipeline stalls are not allowed in the architecture. Each entry in the input buffer is composed of a key and two values which have the same key. As discussed in section 6.4.1, an architecture with equal number of mappers and reducers is free of pipeline stalls. Therefore, the size of the input buffer is not unlimited and bounded by the sized used in the worst scenario.

Lemma 10. *There are no pipeline stalls when the size of the input buffer is k, k is the number of unique keys.*

Proof We prove that the lemma is true in the worst scenario. In the first scenario, $k = n \times m$, where n is a natural number. There are $2m$ pairs of values to be reduced in each clock cycle. Half of them are sent to the $r = m$ reducers. The other m pairs are sent to the input buffer. This happens in $\frac{k}{m}$ consecutive clock cycles. The size of the input buffer is $\frac{k}{m} \times m = k$. In the second scenario, $k = n \times m + t, t < m$.

In the first n clock cycles, the number of pairs sent to the input buffer is $n \times m$. In the $n+1^{th}$ clock cycle, there are $2m$ values sent to the data scheduler. Among them, t values pairing with the values in the storage buffer are sent to the input buffer. There are $2m - t$ values left to be sent to the input buffer and $m - t$ reducers. Because of $2m - t > 2(m - t)$, more input buffer entries are needed. Suppose $2m - t$ is an even number and all of the $2m - t$ values are ready for reducer. That is, the value can be divided into $\frac{2m-t}{2}$ pairs and the values in each pair have the same key. In this worst case, the number of entries required in the input buffer is $\frac{2m-t}{2} - (m - t) = \frac{t}{2}$. Therefore, the size of buffer we need is $n \times m + \frac{t}{2} < n \times m + t = k$. In the case of $2m - t$ is an odd number, we can store a value into the storage buffer. The required size of buffer will be smaller than k. In summary, the input buffer with k entries is enough to build a pipeline stall architecture. □

6.4.3 Processing Partitioning

Saving the values of all keys in the on-chip memory will easily exhaust the FPGA resources. In our architecture, each stored value of a key in the memory is involved in scheduling operation. The resource usage increases dramatically with the number of keys. In the case that there is no enough resources to store all the keys in on-chip memory, partitioning the data processing is necessary. Two strategies are used to partition the data processing depending on the input data characteristics. If the keys of the input data are temporal-coherent, one instance of the architecture can be used to process the whole key space K by dividing it into multiple subsets $K_1 \cup K_2 \cdots K_s$, where $K = K_1 \cup K_2 \cup \cdots K_s, K_1 \cap K_2 \cap \cdots K_s = \Phi$. In each time interval $[t_i, t_{i+1}]$, the data with key in $K_{i \in [1,s]}$ are sent to the architecture. The storage and input buffers must be cleared by a reset signal during the switching between different key subsets. We call this approach as temporal partitioning. SMVM is an example application suitable for temporal partitioning when the matrix data is sent in row-major or column-major order. If the keys of the input data are spatial-coherent, multiple instances of the architectures can be used to process the whole space K independently. We call this approach as spatial partitioning. The *distributed grep* problem mentioned in [39] can be used in spatial partitioning. Some problems can be revised to fit into this kind of partitioning, for example, the problem of counting the number of occurrences of each word in a large collection of documents. One instance of the architecture is only responsible for the words starting with letter ’a’; another one is only for the words starting with ’b’

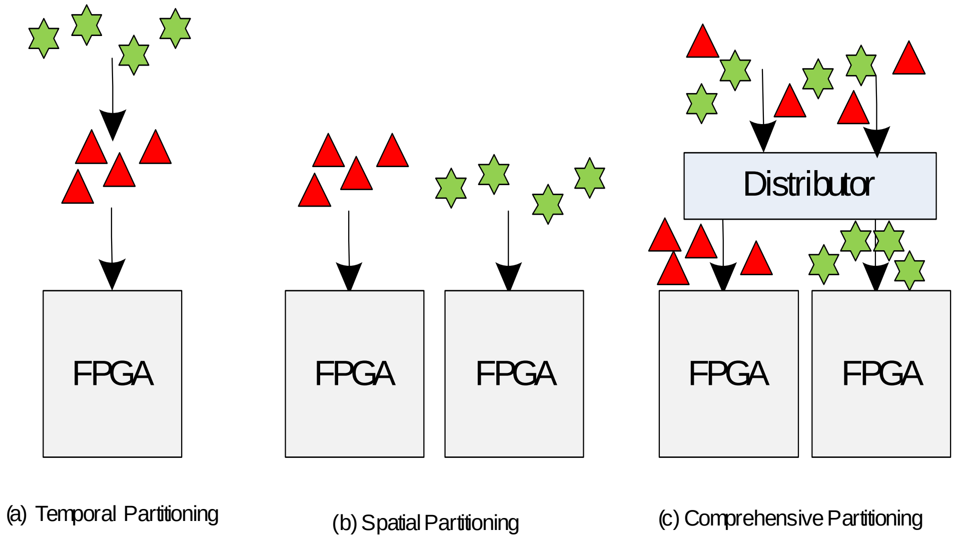


Figure 6.5 Different Partitioning Approaches

and so on. The different partitioning approaches are illustrated in Figure 6.5. When the application is neither temporal nor spatial-coherent, a distributor is used to partition data into different architecture instances. This approach can be called comprehensive partitioning. The application fitting in temporal partitioning can also be spatially partitioned if there are multiple data sources. Similarly, the application fitting in spatial partitioning can also be temporal partitioned if there is enough buffer to store the source data.

6.5 Example: SMVM Implementation Using MapReduce Framework

Sparse Matrix-Vector Multiplication (SMVM) plays a paramount role in many scientific and engineering applications. In solving large linear systems $y = Ax$ (where A is an $n \times n$ sparse matrix with n_z non-zero entries, and x and y are $1 \times n$ matrices) and eigenvalue (λ) problems $Ax = \lambda x$ using iterative methods, SMVM can be performed hundreds or even thousands of times on the same matrix. Our previous work in [138] presents a scalable and efficient FPGA-based SMVM architecture which can handle arbitrary matrix sparsity patterns without excessive preprocessing or zero padding and can be dynamically expanded based on the available I/O bandwidth. In this section, we will use MapReduce framework to implement SMVM without much coding effort while achieving comparable performance.

6.5.1 Mapper and Reducer Design

Each element y_i in the vector y is given by $y_i = \sum_{j=1}^{n} a_{ij}x_j = \sum y_{ij}$. The inputs and outputs of a mappers are (i, a_{ij}, x_j) and (i, $a_{ij}x_j$) respectively. That is, the main component of a mapper is a multiplier. All terms of the sum that make up the component x_i of the matrix-vector product will get the same key(row number). The number of input values for each key in a reducer is two. Each clock cycle, a reducer merges two values with the same key. The inputs and outputs of a reducer are (i, y_{ij}, $y_{ij'}$) and (i, $y_{ij} + y_{ij'}$) respectively. That is, the main component of a reducer is an adder.

6.5.2 Dispatcher Design

The dispatcher acts as a translater between the format of input data and inputs of the mappers, thus is application-dependent. In order to make a comparison with the architecture in [138], the input to the dispatcher is the same as the input in [138]. In each clock cycle, m pairs of numbers from the sparse matrix and vector together with an m-bit input pattern vector(IPV) are are fed into the inputs of the dispatcher. The i^{th} bit of a m-bit IPV is 1 if the i^{th} number in a segment is the last non-zero of a row; otherwise, the i^{th} bit of a m-bit IPV is 0. The dispatcher has a local variable *key-index* which is initialized to 0. The IPV is checked bit by bit from left to right. If the i^{th} bit is equal to 0, *key-index* remains unchanged; Otherwise, it is increased by 1. In both cases, the value of *key-index* is sent as key with the i^{th} pair of data to the i^{th} mapper.

6.5.3 CAM Design

CAM serves to aggregate values in our architecture. If the keys are used as indices of the buffers, the CAMs can be eliminated. In SMVM, if every row in the sparse matrix has a non-zero value, the keys generated in the dispatcher can be used as indices of the buffers. However, the rows which have only zero values are very common in sparse matrices. Many entries in the buffer will be empty if the row numbers are used as indices of them. The storage buffers will be unnecessarily large. One option is to use CAMs as map tables between the keys and the row numbers. The other option is to move the map tables out of the architecture and eliminate the CAMs. We implemented the SMVM in this architecture using the latter option. The map tables are built by software using arrays while the data are sent to the FPGAs. Note that this will not impact the performance too much since accessing the array takes constant time in software.

6.5.4 Data Scheduler Design

The storage buffers and input buffer are implemented as distributed memory in FPGA. The intermediate and final reduced pairs are put into the storage buffers. According to the Algorithm 15, the intermediate results are stored in the storage buffers after all the data is fed into the architecture. The structure of the storages buffer is closely related to the output stage design. For example, one approach is to use a two-dimension array which has k rows and $m + r$ columns as shown in Figure 6.7. Each element in the array has a flag bit and a memory cell. In each clock cycle, the intermediate product with key k_i from the i^{th} mapper is sent to the k_i^{th} row and the i^{th} column of the two dimension storage buffer if the corresponding flag bit is 0. If the flag bit is 1, the intermediate product from the mapper is sent to the reducer along with the value in the storage buffer as shown in Algorithm 15. Another approach is to use two storage buffers as shown in Figure 6.6. In each clock cycle, the data scheduler reads data from the left storage buffer, operates the data using Algorithm 15 and writes the data back to the right storage buffer. At the same time, the previous right storage buffer is pipelined into the left storage buffer. In the first approach, each memory cell is accessed only one time in each clock cycle. In the second approach, the value in each memory cell is written and read multiple times in each clock cycle.[1] A register in hardware can only be accessed one time in each clock cycle. Hence, the second approach needs much more logic to handle the temporary values and introduces more delay in circuit.

The way to schedule data from input buffer to the reducers has great impact to the performance and area of the architecture. One way is to use two pointers associated with the input buffer. One is increased by data scheduler when there are values to be stored into the buffer. The other one is increased when the values are sent to the reducers. Initially the two pointers, say $p1, p2$, are both equal to 0. When there are n values to be removed from the buffer, the new value of $p2$ is $(p2 + n) \ mod \ k$. The modulo operation ensures that the input buffer acts as a circle queue. The queue is empty when the two pointers are equal. In each clock cycle, at most r pairs of values are sent to the reducers. If there is less than r

[1] This will introduce the read/write hazard in hardware. Therefore, two storage buffers are used in the architecture to avoid this hazard.

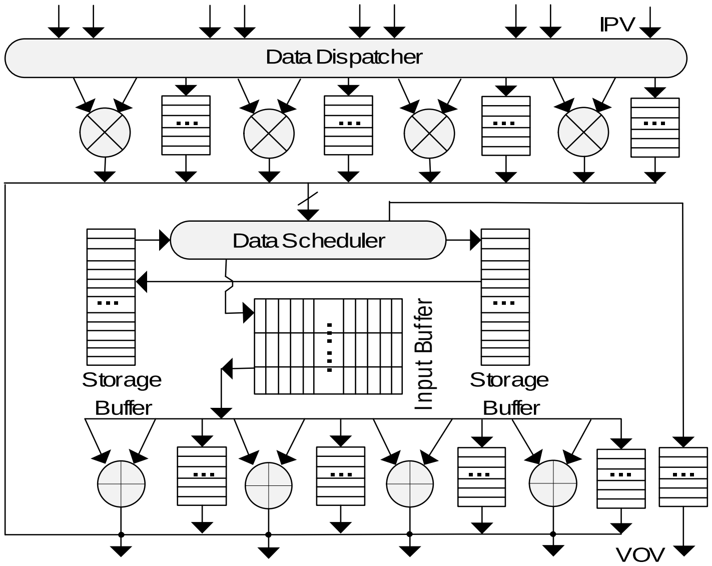


Figure 6.6   SMVM Architecture on MapReduce Framework with two Storage Buffers

pairs of values in the input queue, some reducers have zeros as inputs. The other way is to use only one pointer p associated with the input buffer. The first $m = r$ values in the buffer are sent to the reducers. In each clock cycle, all the values in the input buffer are shift $m = r$ positions toward the reducers. At the same time, the scheduled s pairs of values from the storage buffer, mappers and reducers are sent to the input buffer. The new value of p pointer is set as $max(0, p - r + s)$.

6.5.5 Output Stage Design

The output stage design is closely related to the structure of the storage buffer. In the approach (Figure 6.6) which has two storage buffers, each entry in the storage buffers has at most one intermediate value. The two storage buffers have the same key sequences. Starting from the head of the storage buffers, the data scheduler sends the two values of each key to a reducer. If there are no values in both buffers for a key, the inputs of the reducer are set to zeros. If there is only one value in two buffers for a key, one of the input of the reducer is set to 0. After $\lceil \frac{k}{r} \rceil$ clock cycles, the output stage are done. In the approach which has a large storage buffer in Figure 6.7, the $m + r = 2r$ values in one row are sent to the r reducers in each clock cycle. Therefore, all intermediate values in the storage buffer are reduced after k clock cycles in the output stage.

The data scheduler must have a way to decide the start of output stage. It enters into the output

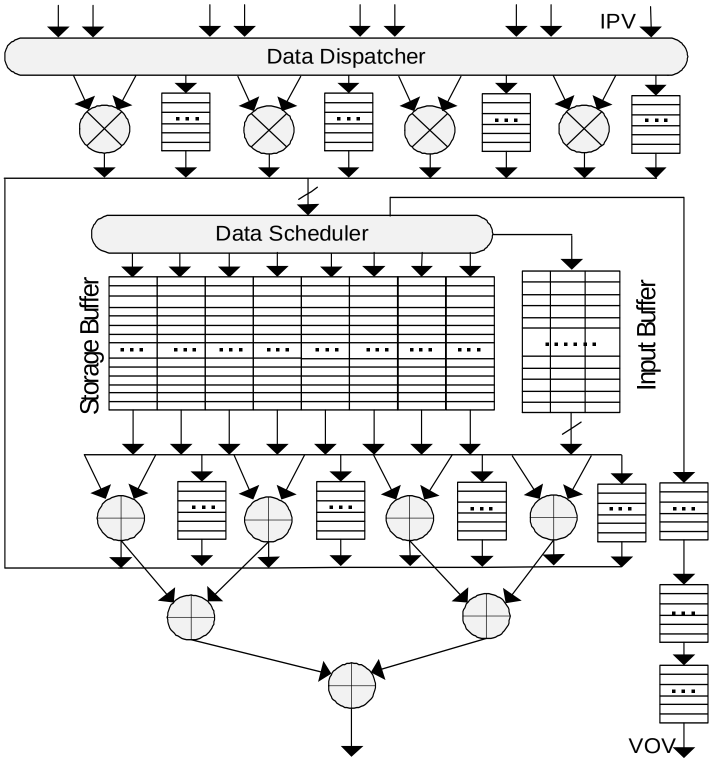


Figure 6.7 SMVM Architecture on MapReduce Framework with One Large Storage Buffer

stage when all of the following conditions are satisfied: (1)there is no more inputs. (2)the input buffer is empty; (3)there is no intermediate data flowing through the mappers and reducers. That is, there are no valid data operations in the architecture. The mechanism to start the output stage decision is application-dependent. For example, a *done* signal controlled by the software can be used as an input to the architecture. In SMVM, the values from the matrix will not be zeros when there is more data available. When any value from the matrix is zero, the *done* signal is set to 1; otherwise, it is zero. In our hardware architecture, we only check the first value and set the *done* signal accordingly in order to avoid high fan-out *or* gate and high delay in the critical path. To make sure there is no intermediate data flowing through the architecture is architecture-dependent. For example, in the architecture shown in Figure 6.7, the data scheduler enters the output stage S_r clock cycles after (1) and (2) are satisfied, where S_r is the number of pipeline stages in the reducers. In the architecture shown in Figure 6.6, the data scheduler enters the output stage $3 \times S_r$ clock cycles after (1) and (2) are satisfied, S_r clock cycles for the case where input data is from the input buffer, $2 \times S_r$ clock cycles for the case where intput data

Table 6.1 Implementation Characteristics of SMVM Hardware Architectures(Stratix IV EP4SE820F4313)

Architecture	**1**	**2**	**3**	**4**	**5**
Description	in 4.5	in 6.6	in 6.6	in 6.7	in 6.7
Multipliers(m)	4	4	4	4	4
Adders(r)	3	4	4	7	7
Storage Buffers	–	2	2	1	1
Storage Buffers Dimension	–	$k \times 1$	$k \times 1$	$k \times (m + r)$	$k \times (m + r)$
Pointers in Input Buffer	–	2	1	2	1
Key Numbers(k)	–	128	128	128	128
Logic Utilization	4%	38%	15%	29%	11%
Maximum Clock Frequency	230MHz	35MHz	61MHz	55MHz	142MHz
MLOC	4300	650	600	650	600

is from the right and left storage buffers.

6.5.6 Architecture Comparison

We compare the architectures drawn in section 6.5.4(Figure 6.7 and Figure 6.6) with the customized architecture proposed in chapter 4(Figure 4.5). Figure 4.5 is the architecture designed specifically for SMVM [138]. This architecture has a unit called Adder Accumulator(AAC) which is specifically designed for SMVM. The details of AAC can be found in [140].

All SMVM architectures were implemented manually in VHDL and simulated and functionally verified in Mentor Graphics Modelsim. We used the Altera MegaWizard Plug-In Manager to customize the IEEE 754 double-precision floating-point multipliers and adders in our implementation. The multipliers were configured to use dedicated multiplier circuitry. The FPGA application developers only need focus on the architectures of the mapper and reducer when using our proposed MapReduce framework. This will save lots of effort on designing the whole architecture thus shorten the product designing time. Furthermore, our MapReduce framework will greatly reduce the coding effort. The lines of HDL code written manually(MLOC) is a good measurement of the effort to implement the architecture. The data scheduler is the largest component of the SMVM architecture based on MapReduce framework. From Table 6.1, the MLOC of the architecture in this work is much smaller than the architecture in [138]. We targeted at Altera Stratix IV EP4SE820F4313 FPGA, which has 650,440 ALUTs and 2,368,5120 memory bits. The architectures were synthesized, placed and routed in 64-bit Altera Quartus II 10.0. The

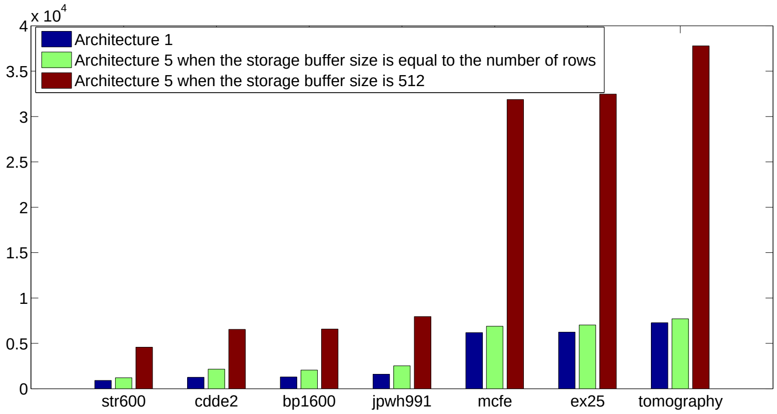


Figure 6.8 Runtime Comparison of SMVM Architecture in Figure 4.5 with the SMVM Architecture in Figure 6.7

size of the storage buffers and the number of mappers and reducers are parameters in the VHDL code. We tuned the parameters and mapped the whole design into the FPGA chip using Quartus software in a reasonable time and memory usage. The number of keys is chosen based on the resource usage and just for illustration purpose. The maximum clock frequency is reported by TimeQuest Time Analyzer in Quartus II. The implementation characteristics of different SMVM architectures is shown in Table 6.1

We simulate the hardware performance on the matrix benchmarks in Table 4.6. Even though different architectures have difference synthesized maximum clock frequencies, they must run in the clock frequency hardwired in a FPGA chip. Hence we choose the runtime as the number of clock cycles. The simulated runtime(based on clock cycles) is shown in Figure 6.8[2]. For the SMVM architectures on MapReduce Framework, two cases are considered: (1)there is enough hardware resources in a single FPGA chip to store all rows in the matrix; (2)the matrix data is temporal partitioned and processed. Note that the runtime of the MapReduce SMVM architecture with spatial partitioned input data is no larger than case (1). That is, the runtime in multi-FPGA system is smaller than case(1). From the figures, the runtime of the SMVM architecture on MapReduce framework is close to that of customized architecture if there is enough hardware resources in a single chip. That is, the MapReduce Architecture achieves similar runtime when it is used in a multi-chip system and the rows are partitioned into multiple FPGA chips. The runtime of the temporal partitioning is much larger because of the extra time

[2] We select architecture 5 because it has higher performance and less resource usage compared with other architectures. Architecture 5 has 36% logic utilization in Stratix IV EP4SE820F4313 when k is equal to 512.

to clean up the intermediate data in the storage buffers in each partition.

6.6 Contribution

We propose an FPGA architecture based on MapReduce program model which allow application developer to focus on the mapper and reducer design. This will greatly simplify the design effort and increase the production efficiency. However, only the applications which can be expressed in MapReduce model are suitable to be implemented in FPGAs using the proposed architecture. When the number of keys is larger than the storage buffer size, multi-FPGA system will be used in order to achieve similar runtime. When the temporal partition is used, the runtime will dramatically increase because of the runtime of output stage. The storage buffers are initialized for each partition in one clock cycle. However, the initialization time will be much larger if the CAM is used in the architecture. Hence, the temporal partition approach should be avoided in the scenarios where the runtime is critical. The design scheduler is the most complicated component of the architecture. We have proposed and implemented several data scheduler designs in this paper. To further reduce the resource and usage and improve the performance of data scheduler will be in our future work.

CHAPTER 7. DISCUSSION AND FUTURE WORK

Since the computing performance of data mining algorithm becomes a problem with the increasing data volume nowadays, hardware acceleration of existing algorithm is imperative. FPGA find its way to accelerate those algorithms and act as an appropriate platform to prototype the hardware architecture. We have designed reconfigurable architectures for a data-intensive application and a data-compute intensive application respectively. We implemented the architecture in Xtremedata and Convey high performance reconfigurable platforms which are state-of-the-art while this thesis is undergoing construction. The implementations are compared with other hardware architecture and software approaches. The performance was analyzed and experimental results were compared against previous state-of-the-art approaches. For data-intensive applications running in the HPRC systems, the performance is usually constrained by the IO bandwidth. This claim is also true in traditional microprocessor(CPU) based computing system. MapReduce is a programming model proposed by Google for processing data-intensive applications using traditional commercial personal computers(PC). The original implementation of the model simplifies the development of larger-scale applications on clustering machines. Since many practical problems can be expressed in this model, MapReduce is widely used in industry. In the end of the thesis, we adopted the Map-Reduce computing model and propose a new programming model(we can call it R-MapReduce) for reconfigurable design which greatly simplifies the design effort in FPGAs. While MapReduce model is used in clustering machines which usually include thousands of machine, R-MapReduce model is an architecture used in a single FPGA-chip. To achieve the same computing objective, the number of lines of manually written VHDL code is significantly reduced while slightly degrading the performance.

To cope with the rapid data growth and compute need in data mining, the breakthrough technologies will come from collaborative efforts of several disciplines [65]. In reconfigurable compute architecture area, some potential future examples are listed below.

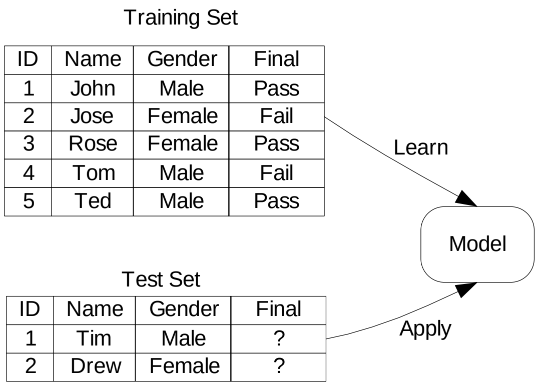
Architecture Design in Multi-chip FPGA System. In our current research, the hardware architecture is usually identically copied to multiple FPGA chips. The data is divided into multiple unrelated parts and sent to the FPGA chips. However, designing heterogeneous compute cores in different FPGA chips to fully utilize the IO bandwidth is an open problem.

Design Software Algorithms scaling to process large data sets for reconfigurable system. The data-fed mechanism is critical to data-intensive computation. In HPRC platform, the software usually has the role of data management which includes sending and receiving data, storing the data in an external disk. The balance of computation and communication overload between software and hardware modules impacts the performance of the system. The methodology to design software algorithms for HPRC platform is an interesting research topic.

APPENDIX A. INTRODUCTION TO DATA MINING

As the flood of data puffs up, extracting useful knowledge manually is infeasible. Consequently, there is an increasing need for automated and intelligent data analysis tools. *Data mining*, or data-driven knowledge discovery, is a powerful technology that converts raw data into an understandable and actionable form, which then can be used to predict future trends or provide meaning to historical events. Data Mining is among the DIC applications. It draws ideas from machine learning, pattern recognition, statistics and database systems [141]. Originally limited to scientific research and medical diagnosis, these techniques are becoming central to a variety of fields including marketing and business intelligence, biotechnology, multimedia and security. Data mining applications can be broadly classified into classification, regression, deviation detection, clustering and association rule discovery, among others. Each domain contains unique algorithmic features. We briefly discuss the problems in classification and clustering in this chapter to give readers some intuition of data mining.

A.1 Classification



Training Set

ID	Name	Gender	Final
1	John	Male	Pass
2	Jose	Female	Fail
3	Rose	Female	Pass
4	Tom	Male	Fail
5	Ted	Male	Pass

Test Set

ID	Name	Gender	Final
1	Tim	Male	?
2	Drew	Female	?

Figure A.1 Building a Classification Model

Given an object, assigning it to one of predefined categories is called *classification*. The input for a classification task is a collection of records(training set). Each record contains a set of attributes, one of the attribute is the *class*. The goal of classification is to find a model for class attribute as a function of the values of other attributes and classify previous unseen records as accurate as possible. Fig. A.1 shows a general approach for building a classification model [141]. The training set is used to build up the model using some sophisticated learning algorithm. The built-up model is used to classify the records in the test set. In this example, the model predicts a student's final performance based on the name and gender. Among existing solutions, Decision Tree Classification(DTC) is a popular method that yields high accuracy while handling large datasets.

A.2 Clustering

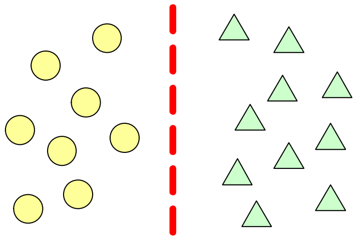


Figure A.2 Clustering Points

Clustering divides data into meaningful groups. In information retrieval, for example, the search engine clusters billions of web pages into different groups, such as news, reviews, videos, audios and so on. One straightforward example of clustering problem is to divide points into different groups as shown in Fig. A.2. One prominent algorithm of clustering technique is K-means [141].

APPENDIX B. INTRODUCTION TO FPGAS

B.1 FPGA Design Flow

Hardware architecture design on FPGAs can be schematic based, HDL based or combination of both. Schematic based entry gives designers much more visibility into the hardware, while HDLs represent a level of abstraction that can isolate the designers from the details of the hardware implementation. Hence, when the design is complex or in an algorithmic way, HDL is a better choice. A simplified FPGA design flow is shown in Fig. B.1.

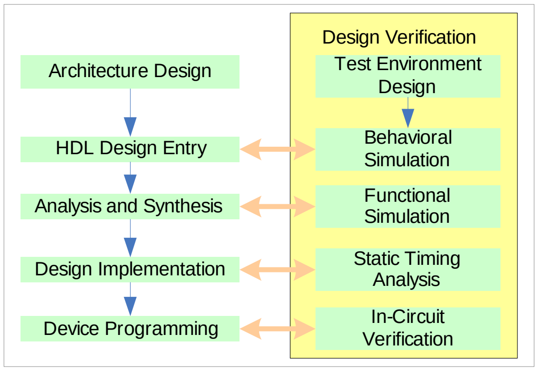


Figure B.1 FPGA Design Flow

Architecture Design The project requirements are analysis and further decomposed into smaller problems. The output of this step is a document which describes the architecture, structural blocks with function and interface description.

HDL Design Entry The architecture is implemented using HDL such as Verilog or VHDL.

Analysis and Synthesis The synthesizer checks the syntax of HDL code and translates it from Register Transfer Level(RTL) to Gate level netlist. The logic network is optimized using device independent and device dependent technique.

Design Implementation This process consists of a sequence of steps: mapping, placement and routing. In the mapping process, the synthesized netlist are the whole circuit into sub blocks. The placement process places the sub blocks from the mapping process into logic blocks according to the constraints. The connection between logic blocks are routed by programming the switch blocks.

Device Programming The configuration file generated by the design is loaded onto FPGAs.

Test Environment Design Usually the test bench code written in HDL for simulation.

Behavioral Simulation Behavioral simulation is performed before synthesis to check syntax and to verify function of the RTL code. Since the design is not yet synthesized to gate level, the timing and resource usage properties are still unknown.

Functional Simulation Different from behavioral simulation, functional simulation is performed after synthesis.

Static Timing Analysis It is performed after placement and routing the design. Post placement and routing timing report incorporates timing delay information of circuit paths to provide a comprehensive timing summary of the design.

In-Circuit Verification Verify the function and behavior the system by checking the Input/Output data of the FPGA chips.

B.2 FPGA as Accelerator

Hardware acceleration is designing optimized hardware architecture most suitable for a given problem, rather than intended for general-purpose use. One example is Graphical Processing Units(GPUs); Another well-known example can be found in the world of Digital Signal Processing(DSP), which spawned an entirely new segment of the computing industry centered around specialized application-specific architectures and software development processes for this class of computation. More recently, embedded processors to accelerate common multimedia, network and security algorithms have been introduced as an attractive alternative to general-purpose microprocessors for meeting real-time performance constraints and shrinking power budgets.

Some prior research on hardware implementations of data intensive application has appeared in the reconfigurable computing literature. In [46], the author proposed an efficient interprocessor communication than a conventional symmetric multiprocessing approach coupled with auto-tuning technologies to improve kernels computational efficiency.s In [54], K-Means clustering is implemented using reconfigurable hardware. In [16, 15], FPGAs are used to implement and accelerate the Apriori algorithm [10], a popular association rule mining technique. A hardware accelerator for DTC in [110] chose to accelerate the Giniscore computation process. Poor scalability with increasingly large and complex datasets, as well as the existence of concise, well-defined kernels of execution make DTC a suitable candidate for re-characterization using hardware accelerators. The DTC platform was prototyped on a Xilinx Virtex-II Pro FPGA, and its performance was compared to an optimized software-only implementation. They achieved a speedup of $5.58\times$ over the software implementation when 16 Gini units were used. The experimental results also strongly suggest that the prototype is scalable and that it would be possible to achieve higher speedup using larger-capacity FPGAs.

APPENDIX C. INTRODUCTION TO TEXT CLASSIFICATION

Let's introduce the concept of Text Classification(TC) by revising an example used in [63]. Suppose we have five documents, each of which has only one sentence as shown in table C.1. Note that this is rare in practice. Here is just an example for illustration purpose. Each of the five documents is assigned a class label. Based on the characteristics of the five training documents, the task of classification is to construct an **document classifier** which can predict and assign a class label to upcoming documents(which is also called testing documents). In our example, one or more class labels can assigned to the testing document $t1$. Table C.2, C.3, C.4 will be explained later.

C.1 An Example of TC

Table C.1 Text Classification Example

	Document ID	**Document Content**	**Document Class**
Training Set	d1	apple apple eve eve	c1
	d2	eve adam eve adam	c2
	d3	apple portable computer	c3
	d4	big apple new york	c4
	d5	fast computer	c3
Testing Set	t1	apple computer	?

C.2 TC as Machine Learning

The definition of TC is very similar to the definition of machine learning. In [71], **Machine Learning** is defined as below:

Definition 7 (Machine Learning). *Given:*

- *a data universe X*

Table C.2 Document Frequency(df) and Inverse Document Frequency(idf) of the Example

Term ID	**Term**	df	$idf = log_{10}\frac{5}{df}$
1	apple	3	0.22
2	eve	2	0.4
3	adam	1	0.7
4	portable	1	0.7
5	computer	2	0.4
6	big	1	0.7
7	new	1	0.7
8	york	1	0.7
9	fast	1	0.7

Table C.3 Vector Representation of the Example

	Term	**apple**	**eve**	**adam**	**portable**	**computer**	**big**	**new**	**york**	**fast**
Document	**Coordinate**	1	2	3	4	5	6	7	8	9
d1	1	0.44	0.8	0	0	0	0	0	0	0
d2	2	0	0.8	1.4	0	0	0	0	0	0
d3	3	0.22	0	0	0.7	0.4	0	0	0	0
d4	4	0.22	0	0	0	0	0.7	0.7	0.7	0
d5	5	0	0	0	0	0.4	0	0	0	0.7
t1		0.22	0	0	0	0.4	0	0	0	0

- *a sample set S, where* $S \subset X$

- *some target function(labeling process)* $f : X \rightarrow \{true, false\}$

- *a labeled training set D, where* $D = \{(x, y)|x \in S, y = f(x)\}$

Compute a function $\widehat{f} : X \rightarrow$*true, false using D such that* $\widehat{f}(x) \cong f(x)$ *for all* $x \in X$

The training set S acts as a representative of the data universe in order to make the process of building the classifier tractable [71]. The function f is the process to assign labels. Here we assumed that f is able to provide a suitable label for any element in X. The above definition formally states that learning can be viewed as computing the function $\widehat{f}$ as an approximation to or a model of the original process f based on the training set S. Machine learning with training set is called **supervised learning**, otherwise, it is called **unsupervised learning**. Document classification is a kind of supervised learning while document clustering is called unsupervised learning. In machine learning, we try to build

Table C.4 Similarities($cos(\theta) = \widetilde{d_i} \bullet \widetilde{d_j}$) in the Example

	d1	**d2**	**d3**	**d4**	**d5**	**t1**
d1	1	0.435	0.127	0.086	0	0.232
d2	0.435	1	0	0	0	0
d3	0.127	0	1	0.047	0.238	0.546
d4	0.086	0	0.047	1	0	0.086
d5	0	0	0.238	0	1	0.435
t1	0.232	0	0.546	0.086	0.435	1

a function/model to approximate the labeling process over the entire data space based on the limited amount of data in the training set. The assumptions underlying are also called **Inductive Learning Hypothesis** [71] described as:

1. the training set is a representative of the whole data universe X

2. the function/model $\widehat{f}$ built to approximate f well over the training set also approximates $\widehat{f}$ well over the entire data universe X

As we will see later, inductive learning hypothesis is critical to design TC algorithms. In the above definition, there are only two labels: $true$ and $false$. But in practical, there may be multiple labels.

C.3 Text Classification Types

Text classification can be divided into two types based on the target text. Classification on web pages is known as **web page categorization/classification**; otherwise, it is called **traditional text classification**. Web page classification can be further divided into more specific categories: subject classification, functional classification, sentiment classification, and other types of classification [124]. **Subject classification** is concerned about the subject or topic of a web page. For example, some web pages are about ”computer”, ”novel”, or ”sport”. **Functional classification** cares about the role of the web pages. For example, some web pages are used for ”tuition submission”, ”Campus Housing” or ”Course Registering”. **Sentiment classification** focuses on the opinion of the authors who write on a web page. For example, the attitude towards the performance of England during the 2010 FIFA World Cup. In this thesis, we only cares about topic classification.

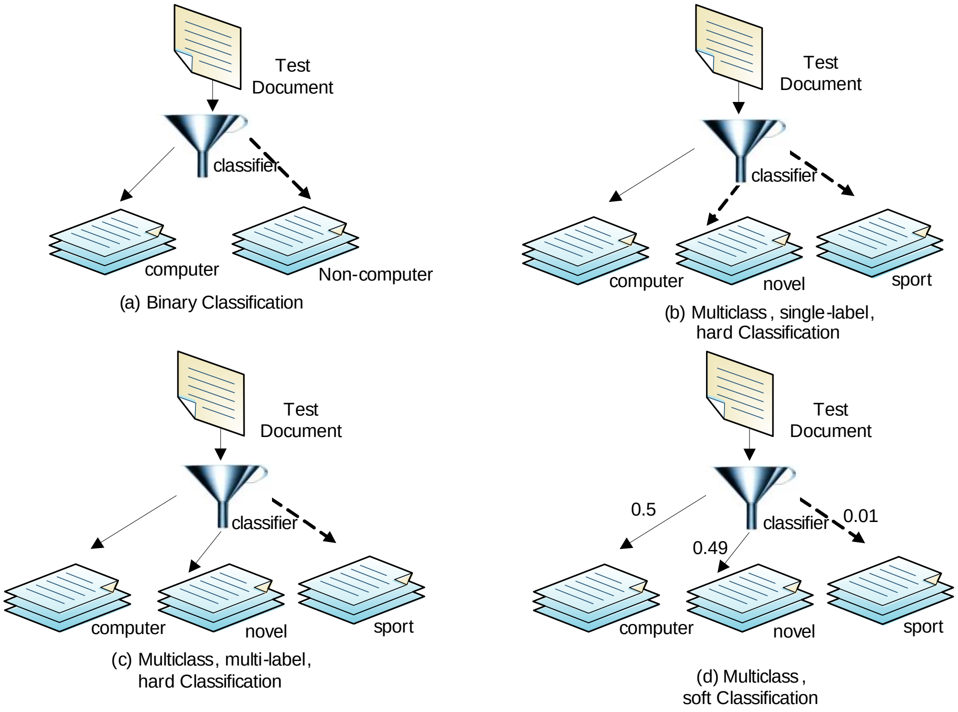


Figure C.1 Types of Classification

The document classification can be divided into **binary classification** and **multiclass classification** depending on the number of classes in the problem. Based on the number of classes that can be assigned to a document, we have **single-label classification** and **multi-label classification**. In single-label classification, one and only one label can be assigned to one document. In multi-label classification, a document can be assigned one or two, even all classes. Classification can also be split into **hard classification** and **soft classification**. In hard classification, a document is either or not in a class, without intermediate state, while in soft classification, a document can be in a class with some probability. The combinations of these categories are illustrated in Fig. C.1. In Fig. C.1(a), the test document is classified as a computer article(shown as solid line) instead of a non-computer one(shown as dotted line). In Fig. C.1(b), the test document is classified as a computer article, instead of novel or sport. In this case, the test document can only be assigned to one label even though there are multiple classes available. In Fig. C.1(c), the test document is assigned to computer and novel class. One can deem the test document as a computer-history related book. In Fig. C.1(d), the test document is a computer article with probability of 0.5, a novel article with probability of 0.49, and a sport with a probability of 0.01.

The binary classification can be used to construct multi-label classification. One just need to transform the problem of multiclass classification under $C = \{c1, c2, \cdots, c_n\}$ into $|n|$ independent problems of binary classification under $\{c_i, \overline{c_i}\}$, for $i = 1, 2, \cdots, n$. However, this requires that different categories be stochastically independent of each other [130]. However, an algorithm for multi-label classification may not be used for binary or single label classification.

C.4 Applications of TC

TC is widely used in medical organizations, industry and academia, such as speech categorization, multimedia document categorization, author identification for literary texts of unknown or disputed authorship, language identification for texts of unknown language, automated identification of text genre, disease categorization for medical text MEDLINE, and automated essay grading [130]. Some researchers have proposed to use TC in classifying confidential files [26]. In the security classification scenario, the security label of a document is not solely determined by the topic since the topic of the document may have some bearing on its sensitivity but is by no means sufficient to determine its ultimate classification level. For example, a document discussing specific deployment details of Canadian light armored vehicles may be sensitive and need to be kept confidential; however a newspaper article discussing the use of Canadian light armored vehicles in Afghanistan is public knowledge and is clearly not classified [26]. Automated security classification is still an open research problem. Web page classification is also essential to many information retrieval(IR) tasks. Since TC has been used in so many different kinds of applications, we here briefly review the most important scenarios in more detail. The borders between the different applications listed here is fuzzy, some of them can be included into others.

C.4.1 Document Organization

For personal purpose or structuring of a corporate documents, TC can be used to classify the incoming files. For example, the incoming advertisement of a newspaper must be classified as a specific category before it is published.

C.4.2 Constructing WebPage Directories

Automatically categorizing web pages, or sites, under the hierarchical or flat catalogues hosted by Internet Portals such as Yahoo!, and the dmoz Open Directory Project(ODP). These directories are mainly constructed and maintained manually. This requires a huge amount of human effort and money. As of February 2008, it was reported that there were78940 editor involved in the dmoz ODP [124].

C.4.3 Vertical Search Engine

The search which restricts itself on a particular topic is called **vertical search** or **topic-specific** search. For example, the query ”matrix” on a vertical search engine for the topic of ”mathematics” will return the matrix theories instead of the ”matrix” movies. This will improve the efficiency and effectiveness of the query.

C.4.4 Email Processing

Systems for categorizing personal’s incoming emails into different directories, especially weeding out spam emails. For example, in gmail box the incoming emails is classified as regular emails or spam.

C.4.5 The Difference between Webpage Classification and Traditional Text Classification

Compared with traditional text classification, classification of web content is different in the following aspects [124]. First, web pages are are semistructured documents in HTML while traditional text are written in plaintext. Second, web documents exist within a hypertext with connections to and from other documents. This feature is critical to the web mining, and is not present in typical text classification problems. For example, Google’s PageRank is a link-based ranking algorithm which calculates the authoritativeness of web pages based on a graph constructed by web pages and their hyperlinks, without considering the topic of each page. Some web-ranking algorithm consider the topics of webpages, such Topic-Sensitive Pagerank proposed by Haveliwala.

Due to the special characteristics of web pages, some techniques which take advantages of the on-page features are used in web content classification. For example [124], textual tags, visual analysis, features of neighbor pages(parents, children, siblings, couple, grandparents), anchor text, and so on. The classifier algorithms used in web page categorization can be inherited from the algorithms of traditional text classifiers. In this chapter, we focus on the traditional text classification.

Large scale topical categorization on the web is an active research area in recent years. In contextual advertising, ad matching platform acts as an intermediary between an advertiser and the publisher of web pages. Whenever a user views a page that is being served by the ad matching system, the ads are selected from a corpus in real time and displayed on a publisher's page. The ads are matched to a webpage based on the content of the pages. A large scale ad matching platform processes a few tens of millions of pages at any point of time thereby manually filtering pages is impossible. Hence, there is a need to categorize the web pages automatically and efficiently. A successfully deployed scalable classification system for web page is proposed in [125]. [67] describes a fast Class-Feature-Centroid(CFC) classifier for multiclass, single-label webpage classifier. There are huge amount of short segments of text on the web, such as search snippets, forum and chat messages, blog and news feeds, product reviews, and book or movie summaries. Classifying these short and sparse text is a challenging task addressed in [118]. Compared with traditional text classification, the web pages data is usually multi-class and multi-label [142]. For instance, a read-world query category problem consists of roughly 1.5 million queries in more than 6000 categories which requires a scalable and efficient multi-label classifier. In this chapter, we will focus on traditional text classification. WebPage classification can be included in our future work.

C.5 Evaluation of Text Classification

The central notion of text classification is inherently hard to be formalized. Thus, the evaluation of text classification is usually conducted *experimentally*, rather than *analytically* [130]. We use two terms to measure the text classification, *effectiveness* and *efficiency*. *Effectiveness* evaluates the quality of classification results. *Efficiency* refers to computational cost and running time of classification algorithms.

Table C.5 The Contingency Table for Class c_i

For Class		True Classification	
c_i		c_i	$\overline{c_i}$
Classifier Result	c_i	T_i^+	F_i^+
	$\overline{c_i}$	F_i^-	T_i^-

C.5.1 Effectiveness of Text Classification

The measure for TC can be borrowed from Information Retrieval(IR) system. A document is *relevant* if it contains valuable information to the user *query*. To an IR system, the *Precision* is the fraction of the relevant and returned results in the whole returned results; the *Recall* is the fraction of the relevant and returned documents in the whole relevant corpus.

C.5.1.1 From the Document Perspective

In the multi-class,soft classification scenario, given a classifier whose input is a document and whose output is ranked list of categories assigned to that document. The recall and precision for a document d_i are defined as [162]:

$$precision_i = \pi_i = \frac{\text{categories found and correct}}{\text{total categories found}}$$

$$recall_i = \rho_i = \frac{\text{categories found and correct}}{\text{total categories correct}}$$

The 11-point interpolated average precision(11-pt AVGP) can be computed based on the precision and recall of documents [162].

C.5.1.2 From the Category Perspective

The precision with respect to a class c_i is denoted as π_i. The recall with respect to a class c_i is denoted as ρ_i. They are defined below.

- π_i: if a random document d_i is classified as c_i using classifier $\widehat{\Gamma}$, the probability that the decision is correct is π_i. That is, $\pi_i = P(\Gamma(d_i) = c_i|\widehat{\Gamma}(d_i) = c_i)$
- ρ_i: if a random document d_i belongs to c_i, the probability that $\widehat{\Gamma}$ classifies it as c_i is ρ_i. That is, $\rho_i = P(\widehat{\Gamma}(d_i) = c_i|\Gamma(d_i) = c_i)$.

The precision and recall with respect to a class c_i are usually described by the contingency table as shown in Table C.5 [130].

- T_i^+: the number of test documents which belong to c_i are correctly classified by $\widehat{\Gamma}$ as c_i. That is, $F_i^- = |\{d_i|\widehat{\Gamma}(d_i) = c_i \bigwedge \Gamma(d_i) = c_i\}|$.
- T_i^-: the number of test documents which belong to $\overline{c_i}$(do not belong to c_i) are correctly classified by $\widehat{\Gamma}$ as $\overline{c_i}$. That is, $F_i^- = |\{d_i|\widehat{\Gamma}(d_i) = \overline{c_i} \bigwedge \Gamma(d_i) = \overline{c_i}\}|$.
- F_i^+: the number of test documents which belong to $\overline{c_i}$ are incorrectly classified by $\widehat{\Gamma}$ as c_i. That is, $F_i^+ = |\{d_i|\Gamma(d_i) = \overline{c_i} \bigwedge \widehat{\Gamma}(d_i) = c_i\}|$.
- F_i^-: the number of test documents which belong to c_i are incorrectly by $\widehat{\Gamma}$ as $\overline{c_i}$. That is, $F_i^- = |\{d_i|\widehat{\Gamma}(d_i) = \overline{c_i} \bigwedge \Gamma(d_i) = c_i\}|$.

π_i and ρ_i can be estimated by T_i^+, T_i^-, F_i^+ and F_i^- as following:

$$\pi_i = P(\Gamma(d_i) = c_i|\widehat{\Gamma}(d_i) = c_i) = \frac{P(\Gamma(d_i) = c_i \bigwedge \widehat{\Gamma}(d_i) = c_i)}{\widehat{\Gamma}(d_i) = c_i} \Longrightarrow \widehat{\pi}_i = \frac{T_i^+}{T_i^+ + F_i^+}$$

$$\rho_i = P(\widehat{\Gamma}(d_i) = c_i|\Gamma(d_i) = c_i) = \frac{P(\widehat{\Gamma}(d_i) = c_i \bigwedge \Gamma(d_i) = c_i)}{\Gamma(d_i) = c_i} \Longrightarrow \widehat{\rho}_i = \frac{T_i^+}{T_i^+ + F_i^-}$$

Table C.6 The Global Contingency Table

For Class		**True Classification**	
$C = \{c_1, c_2, \cdots, c_n\}$		$\sum c_i$	$\sum \overline{c_i}$
Classifier Result	$\sum c_i$	$T^+ = \sum_{i=1}^n T_i^+$	$F^+ = \sum_{i=1}^n F_i^+$
	$\sum \overline{c_i}$	$F^- = \sum_{i=1}^n F_i^-$	$T^- \sum_{i=1}^n T_i^-$

Based on the local contingency table in Table C.5, the Global Contingency Table is shown in Table C.6. There are two approaches to obtain the estimates of π and ρ for the classifier $\widehat{\Gamma}$ as listed below:

- Micro Average:

$$\widehat{\pi}^\mu = \frac{T^+}{T^+ + F^+} = \frac{\sum_{i=1}^n T_i^+}{\sum_{i=1}^n (T_i^+ + F_i^+)}$$

$$\widehat{\rho}^\mu = \frac{T^+}{T^+ + F^-} = \frac{\sum_{i=1}^n T_i^+}{\sum_{i=1}^n (T_i^+ + F_i^-)}$$

- Macro Average:

$$\widehat{\pi}^M = \frac{\sum_{i=1}^{n} \widehat{\pi}_i}{n}$$
$$\widehat{\rho}^M = \frac{\sum_{i=1}^{n} \widehat{\rho}_i}{n}$$

These two approaches may give quite different results. Whether one or the other should be used totally depends on the application scenarios. Neither precision nor recall makes sense in isolation from each other. For example, a simple classifier which assigns a document to all classes obviously has $\rho = 1$. In practice, the F_β function is usually used.

$$F_\beta = \frac{(\beta^2 + 1)\pi\rho}{\beta^2\pi + \rho}$$

where β is a real number. Obviously, $F_0 = \pi$ and $F_\infty = \rho$. In practice, $\beta = 1$ is used.

$$F_1 = \frac{2\pi\rho}{\pi + \rho}$$

There are also some other measures used. For example, *accuracy* estimates of a classifier is defined as $\hat{A} = \dfrac{T^+ + T^-}{T^+ + T^- + F^+ + F^-}$. More measures can be found in [130, 162, 101, 147].

C.6 Efficiency of Text Classification

Table C.7 The Time Complexity of Classifiers

Classifier	**Method**	**Time Complexity**	
		Training	**Testing**
Naive Bayes		$\Theta(\lvert D\rvert L_{ave} + \lvert C\rvert\lvert V\rvert)$	$\Theta(\lvert C\rvert M_a)$
Rocchio		$\Theta(\lvert D\rvert L_{ave} + \lvert C\rvert\lvert V\rvert)$	$\Theta(\lvert C\rvert M_a)$
kNN		$\Theta(\lvert D\rvert L_{ave})$	$\Theta(\lvert D\rvert M_{ave} M_a)$
SVM	conventional	$O(\lvert C\rvert\lvert D\rvert^3 M_{ave}); \approx O(\lvert C\rvert\lvert D\rvert^{1.7} M_{ave})$	
SVM	cutting planes	$O(\lvert C\rvert\lvert D\rvert M_{ave})$	$O(\lvert C\rvert M_a)$

The efficiency of a classifier is usually measure in terms of the time and space complexity. The time complexity of training and testing of some classifiers are listed in Table C.7 which is from [101]. In this table, the training time is the time taken to build a classifier over the documents D in the training set; the testing time is the time to classify a document; $|V|$ is the number of terms in the dictionary; $|C|$ is the number of classes; L_{ave} is the average number of tokens per document; M_{ave} is the average number of non-zero features(terms) of a document; L_a and M_a are the number of tokens and types, respectively, in the test documents.

As we have stated in Alg. 11, training a kNN classifier is composed of determining the value of k and building VSM of training documents. The testing time of kNN grows linearly with the number of testing documents since the testing document is compared with each training document. Testing time is independent of the number of classes. Hence kNN is suitable for the document classification problem with a lot of classes.

C.7 Feature Reduction in VSM

The terms in representing a document are also called **features**. Based on Encyclopedia Americana(Volume 10. Grolier, 1999), there are more than 600,000 words in English language, not to say the increasing number of scientific words. In practical collections, the number of terms is extremely large. For instance, in the Reuters collection 3, the training set vocabulary has 24,858 unique words [162]. Even though most modern classifiers can scale to such large dimensions, the computation cost is prohibitive with nowadays computation power.

Feature reduction is also beneficial since it tends to reduce *overfitting*. **Overfitting** is a phenomenon by which the classifier is tuned not only to the *constitutive* characteristics of the categories but also the *contingent* characteristics of the training data. In other words, a classifier with *overfitting* tends to consider those terms which occur in a document category but do not contribute to that document class/category. Those terms are also called **outliers** with respect to some classes. Classifiers that overfit the training data are good at reclassifying the documents they have been trained on, but much worse at classifying previously unseen documents. Removing outliers is a challenge task and must be done with care in case deleting some useful terms. The feature reduction can be performed locally or globally [130]. Based on the nature of the resulting terms, the global feature reduction techniques are drawn into two distinct groups, **Term Selection** and **Term Extraction**.

- Term Selection

  Given the set of terms in a document collection as Ψ, selecting the set of terms Φ where $|\Phi| << |\Psi|$ while yielding the same or better effectiveness. The simplest term selection approach is based on the document frequency, where only the terms with the highest number of documents are retained. Some other information-theoretic terms selection functions are information gain, χ^2,mutual information, NGL coefficient, relevancy score, GSS coefficient [130]. A controlled study on a large number of terms selection methods using four well-known classifiers is reported in [127]. They found that χ^2 statistics consistently outperformed other methods. Experiments have shown that it is possible to reduce the number of features by a factor of 10 without effectiveness loss. For example, χ^2 based feature selection reduce the vocabulary from 24,858 unique words to 2,485 [162]. Other term selection methods can be found in [130, 101] with more detail.

- Term Extraction

  Term extraction does not consist of a subset of original features but ”synthesizes” the original terms and creates artificial terms based on the newly synthesized ones. Two approaches are usually used: *term clustering* and *latent semantic indexing*. Term clustering groups term with a high degree semantic relatedness, so that a representative of them may be used in the vector space. Term clustering is different from term selection in that it tends to address polysemy, homonymy and synonymy. A lot of algorithms are proposed for term clustering [130]. The latent semantic indexing(LSI) [40] uses Singular Value Decomposition(SVD) to identify patterns in the terms relationships and reduces the dimension of the vectors. It is based on the assumption that words that are used in the same contexts tend to have the same meanings. The obtained terms, unlike the term selection and term clustering are not interpretable. The original work of LSI is in [40].

BIBLIOGRAPHY

[1] Apache hadoop. hadoop.apache.org.

[2] Convey Computer Corporation. www.conveycomputer.com.

[3] Cray Inc. www.cray.com.

[4] General-purpose computation on graphigcs hardware. www.gpgpu.org.

[5] SRC Computers, LLC. www.srccomp.com.

[6] Wikipedia. available at www.wikipedia.org.

[7] Xilinx coregen. www.xilinx.com.

[8] Xtremedata inc. xd2000i development system user handbook. www.xtremedata.com.

[9] C. Aasheim and G. J. Koehler. Scanning world wide web documents with the vector space model. *Decision Support Systems*, 42(2):690–699, 2006.

[10] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[11] S. R. Alam, P. K. Agarwal, M. C. Smith, J. S. Vetter, and D. Caliga. Using fpga devices to accelerate biomolecular simulations. *Computer*, 40:66–73, 2007.

[12] J. I. Aliaga and V. Hernandez. Symmetric sparse matrix-vector product on distributed memory multiprocessors. *Parallel Computing and Transputer Applications*, pages 109–120, 1992.

[13] Altera. Accelerating high-performance computing with fpgas. white paper, Altera, October 2007.

[14] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Ucb/eecs-2006-183, EECS Department, University of California, Berkeley, December 2006.

[15] Z. Baker and V. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. In *Proceedings of the IEEE symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 3–12, April 2005.

[16] Z. Baker and V. Prasanna. An architecture for efficient hardware data mining using reconfigurable computing systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 67–75, April 2006.

[17] R. Baxter, S. Booth, M. Bull, G. Cawood, K. D'Mellow, X. Guo, M. Parsons, J. Perry, A. Simpson, and A. Trew. High-performance reconfigurable computing - the view from edinburgh. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 373–279, 2007.

[18] R. Bayardo, B. Goethals, and M. Zaki, editors. *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*. Nov. 2004.

[19] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report nvr-2008-004, NVIDIA Corporation, December 2008.

[20] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of Supercomputing 2009*, August 2009.

[21] D. Bennett. Is high-performance, reconfigurable computing the next supercomputing paradigm? In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages xv –xv, 2006.

[22] J. Bentley and H. Kung. A tree machine for searching problems. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 265–266, 1979.

[23] R. H. Bisseling and W. Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005.

[24] C. Borgelt. An implementation of the fp-growth algorithm. In *Proceedings of the International Workshop on Open Source Data Mining (ODSM)*, pages 1–5, 2005.

[25] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, pages 1–33, 2010.

[26] J. Brown. Developing an automatic document classification system. Technical report, Defence R & D Canada, January 2010.

[27] D. Buell, T. E. ghazawi George, and K. Gaj. High-performance reconfigurable computing. *IEEE Computer*, 40:23–27, 2007.

[28] H. Calderon and S. Vassiliadis. Reconfigurable fixed point dense and sparse matrix-vector multiply/add unit. In *In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 311–316, August 2006.

[29] J. Carter and K. Rajamani. Designing energy-efficient servers and data centers. *IEEE Computer*, 43(7):76–78, 2010.

[30] U. Catalyurek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 75–86, 1996.

[31] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Computing*, 10:673–693, 1999.

[32] S. Chakrabarti. *Data Mining: know it all*. Morgan Kaufmann, 2008.

[33] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Symposium on Application Specific Processors*, pages 101 –107, 2008.

[34] N. CHREC. Nsf center for high-performance reconfigurable computing. chrec.ufl.edu/.

[35] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[36] I. Corporation. Sparse matrix-vector multiplication toolkit for graphics processing units. www.alphaworks.ibm.com/tech/spmv4gpu, 2009.

[37] T. Davis and Y. Hu. The university of florida sparse matrix collection. www.cise.ufl.edu/research/sparse/matrices.

[38] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *International Conference on Field-Programmable Technology (FPT)*, pages 33–40, 2008.

[39] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, pages 137–150, 2004.

[40] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[41] M. deLorimier and A. Dehon. Floating-point sparse matrix-vector multiply for fpgas. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, February 2005.

[42] G. Demartini, J. Gaugaz, and W. Nejdl. A vector space model for ranking entities and its application to expert search. In *Proceedings of the 31th European Conference on IR Research on Advances in Information Retrieval*, pages 189–201. Springer-Verlag, 2009.

[43] I. S. Dhillon, J. Fan, and Y. Guan. Efficient clustering of very large document collections. In V. K. R. Grossman, C. Kamath and R. Namburu, editors, *Data Mining for Scientific and Engineering Applications*. Kluwer Academic Publishers, 2001. Invited Book Chapter.

[44] I. S. Dhillon and D. S. Modha. Concept decompositions for large sparse text data using clustering. *Machine Learning*, 1(2):143–175, 2001.

[45] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington. Sparse matrix libraries in C++ for high performance architectures, 1994.

[46] D. Donofrio, L. Oliker, J. Shalf, M. F. Wehner, C. Rowen, J. Krueger, S. Kamil, and M. Mohiyuddin. Energy-efficient computing for extreme-scale science. *IEEE Computer*, 42(11):62–71, 2009.

[47] I. S. Duff, M. A. Heroux, and R. Pozo. The Sparse BLAS. Technical Report TR/PA/01/24, CERFACS, September 2001.

[48] K. D.Underwood. Challenges for reconfigurable computing in hpc. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages xvI –xvI, 2006.

[49] K. D.Underwood, K. S. Hemmert, and C. Ulmer. Architectures and APIs: assessing requirements for delivering fpga performance to applications. In *Proceedings of Supercomputing (SC)*, 2006.

[50] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell. The promise of high-performance reconfigurable computing. *IEEE Computer*, 41(2):69–76, 2008.

[51] Y. El-Kurdi, D. Giannacopoulos, and W. J. Gross. Hardware acceleration for finite-element electromagnetics: Efficient sparse matrix floating-point computations with FPGAs. *IEEE Transactions on Magnetics*, 43(4):1525–1528, 2007.

[52] J. Erhel. Sparse matrix multiplication on vector computers. Technical Report RR-1101, INRIA, 1989.

[53] M. Estlick, M. Leeser, and J. Theiler. Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 103–110, 2001.

[54] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski. Algorithmic transformations in the implementation of k- means clustering on reconfigurable hardware. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 103–110, 2001.

[55] J. Fan. Mc: A toolkit for creating vector models from text documents. www.cs.utexas.edu/users/dml/software/mc/.

[56] L. A. Francis. Taming text: An introduction to text mining, 2006.

[57] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6, 2008.

[58] Garland and Michael. Sparse matrix computations on manycore gpu's. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 2–6, 2008.

[59] A. George. Reconfigurable computing research pushes forward. www.anandtech.com/show/2549/2.

[60] R. Geus and S. Rollin. Towards a fast parallel sparse symmetric matrix-vector multiplication. *Parallel Computing*, 27(7):883–896, 2001.

[61] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 577–588, 2005.

[62] B. Goethals. Fp-growth implementation. adrem.ua.ac.be/ goethals/software.

[63] N. Goharian, A. Jain, and Q. Sun. Comparative analysis of sparse matrix algorithms for information retrieval. *Systemics, Cybernetics and Informatics*, 1(1):38–46, 2002.

[64] M. Gokhale, J. Cohen, A. Yoo, W. M. Miller, A. Jacob, C. Ulmer, and R. Pearce. Hardware technologies for high-performance data-intensive computing. *Computer*, 41(4):60–68, 2008.

[65] I. Gorton, P. Greenfield, A. Szalay, and R. Williams. Data-intensive computing in the 21st century. *Computer*, 41(4):30–32, 2008.

[66] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *Journal of Supercomputing*, 50(1):36–77, 2009.

[67] H. Guan, J. Zhou, and M. Guo. A class-feature-centroid classifier for text categorization. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 201–210, 2009.

[68] S. A. Guccione, D. Levi, and D. Downs. A reconfigurable content addressable memory. In *In 7th Reconfigurable Architectures Workshop(RAW 2000)*, 2000.

[69] K. Gulati and S. P. Khatri. *Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs*. Springer, 2010.

[70] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer. Using knn model for automatic text categorization. *Soft Computing*, pages 423–430, 2006.

[71] L. H. Hamel. *Knowledge Discovery with Support Vector Machines*. Wiley-Interscience, New York, NY, USA, 2009.

[72] E.-H. Han, G. Karypis, and V. Kumar. Text categorization using weight adjusted k-nearest neighbor classification. In *PAKDD '01: Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 53–65. Springer-Verlag, 2001.

[73] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, January 2004.

[74] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, 2008.

[75] C. He, G. Qin, M. Lu, and W. Zhao. Group-alignment based accurate floating-point summation on FPGAs. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 136–142, 2006.

[76] D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Modeling data locality for the sparse matrix-vector product using distance measures. *Parallel Computing*, 27(7):897–912, 2001.

[77] E.-J. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *International Conference on Computational Science*, 2001.

[78] P. B. James-Roxby and D. J. Downs. An efficient content-addressable memory implementation using dynamic routing. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 81–90, 2001.

[79] E. jin Im and K. Yelick. Optimizing sparse matrix vector multiplication on smps. In *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[80] E. jin Im and K. Yelick. Optimization of sparse matrix kernels for data mining. In *First SIAM Conf. on Data Mining*, 2000.

[81] T. Joachims. Text categorization with support vector machines: learning with many relevant features. In *Proceedings of ECML-98, 10th European Conference on Machine Learning*, pages 137–142. Springer, 1998.

[82] F. B. Jocelyne, F. Bodin, J. Erhel, and T. Priol. Parallel sparse matrix vector multiplication using a shared virtual memory environment. In *Proceeeding of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 421–428, 1993.

[83] N. H. Kapadia. *A SIMD SPARSE MATRIX-VECTOR MULTIPLICATION ALGORITHM FOR COMPUTATIONAL ELECTROMAGNETICS AND SCATTERING MATRIX MODELS*. PhD dissertation, Purdue University, West Lafayette, Indiana, 1994.

[84] S. K.Moore. Multicore is bad news for supercomputers. *IEEE Spectrum*, 45(11):15–15, 2008.

[85] P. M. Kogge. *The Architecture of Pipelined Computers*. Hemisphere Publishing Corporation, 1981.

[86] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000. www.ecn.purdue.edu/KDDCUP.

[87] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda, and A. Nishida. Performance evaluation of parallel sparse matrix vector products on sgi altix3700. *Springer Science*, pages 153–163, 2008.

[88] K. Kourtis, G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th conference on Computing frontiers*, pages 87–96, 2008.

[89] R. T. Kouzes, G. A. Anderson, S. T. Elbert, I. Gorton, and D. K. Gracio. The changing paradigm of data-intensive computing. *Computer*, 42(1):26–34, 2009.

[90] U. Kulisch. The fifth floating-point operation for top-performance computers. Technical report, Universitat Karlsruhe, April 1997.

[91] S. Y. Kung, editor. *VLSI Array Processors*. Prentice Hall, 1988.

[92] K. Lang. the 20 newsgroups data set. people.csail.mit.edu/jrennie/20Newsgroups/.

[93] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *Proceedings of the 2004 International Conference on Parallel Processing*, pages 169–176, 2004.

[94] S. Lee and R. Eigenmann. Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 195–204, 2008.

[95] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *In Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 155–162, 2009.

[96] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[97] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.

[98] L. Ling, N. Oliver, C. Bhushan, W. Qigang, A. Chen, S. Wenbo, Y. Zhihong, A. Sheiman, I. McCallum, J. Grecco, H. Mitchel, L. Dong, and P. Gupta. High-performance, energy-efficient platforms using in-socket FPGA accelerators. In *Proceeding of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 261–264, 2009.

[99] C. Lucchese, S. Orlando, and R. Perego. kDCI: on using direct count up to the third iteration. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*, 2004.

[100] J. Magalhaes, S. Overell, and S. Ruger. A semantic vector space for query by image example. In *ACM SIGIR Conf. on research and development in information retrieval, Multimedia Information Retrieval Workshop*, 2007.

[101] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[102] G. Manzini. Lower bounds for sparse matrix vector multiplication on hypercubic networks. In *Discrete Mathematics and Theoretical Computer Science*, pages 35–47, 1998.

[103] S. McGettrick, D. Geraghty, and C. McElroy. An FPGA architecture for the PageRank eigenvector problem. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 523–526, 2008.

[104] R. T. McLay, S. Swift, and G. F. Carey. Maximizing sparse matrix-vector product performance on risc based mimd computers. *Journal of parallel and distributed computing*, 37(2):146–158, 1996.

[105] L. M.Ni and K. Hwang. Vector-reduction techniques for arithmetic pipelines. *IEEE Transactions on Computers*, c-34(5):404–411, 1985.

[106] A. Monakov and A. Avetisyan. Implementing blocked sparse matrix-vector multiplication on nvidia gpus. *Springer Berlin / Heidelberg*, 2009.

[107] N. Moore, A. Conti, M. Leeser, and L. S. King. Vforce: An extensible framework for reconfigurable supercomputing. *Computer*, 40:39–49, 2007.

[108] muthu Manikandan Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus. Technical Report RC24704, IBM Research Report, April 2009.

[109] K. K. Nagar and J. D.Bakos. A sparse matrix personality for the convey hc-1. In *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–8, 2011.

[110] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno. An FPGA implementation of decision tree classification. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 189–194, April 2007.

[111] S. G. Nasteaa, O. Friederb, and T. El-Ghazawi. Load-balanced sparse matrix vector multiplication on parallel computers. *Journal of Parallel and Distributed Computing*, 46(2):180–193, 1997.

[112] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Appl. Algebra Eng., Commun. Comput*, 18(3):297–311, 2007.

[113] A. N.Langville and C. D.Meyer, editors. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, New Jersy, 2006.

[114] Nvidia. Cudpp:cuda data parallel primitives library. gpgpu.org/developer/cudpp.

[115] A. T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, 14:519–530, 1993.

[116] OpenCores. *Double Precision Floating Point Core*, 2009.

[117] D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.

[118] X.-H. Phan, L.-M. Nguyen, and S. Horiguchi. Learning to classify short and sparse text & web with hidden topics from large-scale data collections. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 91–100, 2008.

[119] J. Pichel, D. Heras, J. Cabaleiro, and F. Rivera. Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 66–71, 2004.

[120] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999.

[121] J. Pisharath and A. Choudhary. Design of a hardware accelerator for density based clustering applications. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 101–106, July 2005.

[122] R. Pozo, K. Remington, and A. Lumsdaine. Sparselib++. math.nist.gov/sparselib++/.

[123] I. Pramudiono and M. Kitsuregawa. Parallel FP-growth on PC cluster. In *Proceedings of the Pacifica-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2003.

[124] X. Qi and B. D. Davison. Web page classification: Features and algorithms. *ACM Comput. Surv.*, 41(2), 2009.

[125] S. Rajan, D. Yankov, S. J. Gaffney, and A. Ratnaparkhi. A large-scale active learning system for topical categorization on the web. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 791–800, 2010.

[126] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 13–24, 2007.

[127] M. Rogati and Y. Yang. High-performing feature selection for text classification. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 659–661, 2002.

[128] L. F. Romero and E. L. Zapata. Data distributions for sparse matrix vector multiplication. *Parallel Computing*, 21(4):583–605, 1995.

[129] T. Schumacher, C. Plessl, and M. Platzner. An accelerator for k-th nearest neighbor thinning based on the imorc infrastructure. In *19th International Conference on Field Programmable Logic and Applications(FPL)*, pages 338–344, 2009.

[130] F. Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, 2002.

[131] J. Shalf. The new landscape of parallel computer architecture. *journal of Physics*, 78, 2007.

[132] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. Fpmr: Mapreduce framework on fpga. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 93–102, August 2010.

[133] A. L. Shimpi and D. Wilson. Building nvidiaś gt200. www.anandtech.com/show/2549/2.

[134] P. Soucy and G. W. Mineau. A simple knn algorithm for text categorization. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 647–648. IEEE Computer Society, 2001.

[135] E. d. Sturler and D. Loher. Implementing iterative solvers for irregular sparse matrix problems in high performance fortran. In *Proceedings of the International Symposium on High Performance Computing*, pages 293–304, 1997.

[136] J. Sun, G. Peterson, and O. Storaasli. Mapping sparse matrix-vector multiplication on fpgas. In *Proceedings of Supercomputing 99*, 1999.

[137] J. Sun, G. Peterson, and O. Storaasli. Sparse matrix-vector multiplication design on FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines*, April 2007.

[138] S. Sun, M. Monga, P. Jones, and J. Zambreno. An i/o bandwidth-sensitive sparse matrix-vector multiplication engine on fpgas. *IEEE Transactions on Circuits and Systems-I (TCAS-I), to appear*, 2011.

[139] S. Sun and J. Zambreno. Mining association rules with systolic trees. Proceedings of the International Conference on Field-Programmable Logic and its Applications (FPL), September 2008.

[140] S. Sun and J. Zambreno. A floating-point accumulator on fpgas. In *International Conference on Field-Programmable Technology (FPT'09)*, December 2009.

[141] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[142] L. Tang, S. Rajan, and V. K. Narayanan. Large scale multi-label classification via metalabeler. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 211–220, 2009.

[143] T.Claburn. Yahoo claims record with petabyte database. *Information Week Magazine*, May 2008.

[144] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *IBM Journal of Research and Development*, 1997.

[145] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–725, 1997.

[146] J. L. Tripp, M. B. Gokhale, and K. D. Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40:28–37, 2007.

[147] G. Tsoumakas and I. Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining*, pages 1–13, 2007.

[148] R. S. TUMINARO and J. N. S. S. A. HUTCHINSON. Parallel sparse matrix vector multiply software for matrices with data locality. *Concurrency: practice and experience*, 10(3):229–247, 1998.

[149] P. D. Turney and P. Pantel. From frequency to meaning: Vector space models of semantics. *Journal of Artificial and Intelligence Research*, pages 141–188, 2010.

[150] T. Uno, M. Kiyomi, and H. Arimura. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*, 2004.

[151] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[152] A. Verma, N. Zea, B. Cho, I. Gupta, and R. H. Campbell. Breaking the mapreduce stage barrier. In *IEEE International Conference on Cluster Computing*, pages 235–244, 2010.

[153] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing (SC)*, November 2002.

[154] R. W. Vuduc and H. J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communcations*, volume 3726 of *Lecture Notes in Computer Science*, pages 807–816. Springer Berlin Heidelberg, 2005.

[155] Y. Wang and Z.-O. Wang. A fast knn algorithm for text categorization. In *International Conference on Machine Learning and Cybernetics*, pages 3436 –3441, 2007.

[156] W.Augustin, V. Heuveline, and J.P.Weiss. Convey hc-1: The potential of fpgas in numerical simulation. www.emcl.kit.edu/preprints/emcl-preprint-2010-07.pdf, 2010.

[157] G. Wellein, G. Hager, A. Basermann, and H. Fehske. Fast sparse matrix-vector multiplication for teraflops computers. In *Proceedings of the 5th international conference on High performance computing for computational science*, pages 287–301, 2003.

[158] Y.-H. Wen, J.-W. Huang, and M.-S. Chen. Hardware-enhanced association rule mining with hashing and pipelining. *IEEE Transactions on Knowledge and Data Engineering*, 20(6):784–795, 2008.

[159] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix vector multiplication on emerging multicore platforms. In *Proceedings of Supercomputing (SC)*, 2007.

[160] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowl. Inf. Syst.*, 14(1):1–37, 2007.

[161] Y. Yang. Expert network: effective and efficient learning from human decisions in text categorization and retrieval. In *SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 13–22. Springer-Verlag New York, Inc., 1994.

[162] Y. Yang. An evaluation of statistical approaches to text categorization. *Information Retrieval*, 1(2):69–90, 1999.

[163] Y. Yang, J. Zhang, and B. Kisiel. A scalability analysis of classifiers in text categorization. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 96–103, 2003.

[164] Y. Ye and C.-C. Chiang. A parallel Apriori algorithm for frequent itemsets mining. In *Proceedings of the International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 87–94, 2006.

[165] Y.-J. Yeh, H.-Y. Li, W.-J. Hwang, and C.-Y. Fang. Fpga implementation of knn classifier based on wavelet transform and partial distance search. In *Proceedings of the 15th Scandinavian conference on Image analysis*, pages 512–521, 2007.

[166] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong. Map-reduce as a programming model for custom computing machines. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 149–159, 2008.

[167] H. Zhang, A. C. Berg, M. Maire, and J. Malik. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2126–2136, 2006.

[168] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar, and J. Bakos. FPGA vs. GPU for sparse matrix vector multiply. In *International Conference on Field-Programmable Technology (FPT)*, pages 255–262, 2009.

[169] L. Zhuo, G. Morris, and V. Prasanna. Designing scalable FPGA-based reduction circuits using pipelined floating-point cores. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[170] L. Zhuo, G. Morris, and V. Prasanna. High-performance reduction circuits using deeply pipelined operators on FPGAs. *IEEE Trans. Parallel Distrib. Syst.*, 18(10):1377–1392, 2007.

[171] L. Zhuo and V. Prasanna. High-performance and area-efficient reduction circuits on FPGAs. In *Proceedings of the International Symposium on Computer Architecture on High Performance Computing (SBAC-PAD)*, pages 52–59, 2005.

[172] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, February 2005.

[173] P. Zicari, S. Perri, P. Corsonello, and G. Cocorullo. An optimized adder accumulator for high speed macs. In *Proceedings of the International Conference on ASIC (ASICON)*, pages 757–760, October 2005.