AUTOMATIC SOFTWARE PERFORMANCE OPTIMIZATION ON MODERN ARCHITECTURES

BY

CHANGHAO JIANG

B.S., Tsinghua University, 1999
M.S., Tsinghua University, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

As computer architectures become more complex, the task of writing efficient program to best utilize the underlying architecture's power increasingly becomes an extremely difficult and expensive process. Traditional approach of expert manual tuning of software performance becomes infeasible as both software and hardware complexity grow. To make things even worse, the relative cost of man labor compared with that of machine computation increases rapidly. One approach to attacking the problem is automatic library generation via empirical evaluation. The essential idea is to have a meta-program automatically generate other high performance program via empirical evaluation and intelligent search. The methodology has been successfully applied in several application domains, such as numerical computing, signal processing, sorting, etc.

This dissertation extends the automatic library generation methodology to emerging untraditional computer architectures and to a more complex application domain. Specifically, it consists of two parts of work: First, it studies and implements an automatic matrix multiply library generator for graphics hardware – a specialized architecture with enormous computing power for graphics applications; Second, it uses machine learning techniques to automatically select the best algorithm for frequent pattern mining problems according to input characteristics.

In order to utilize the tremendous computing power of graphics hardware and to automatically adapt to the fast and frequent changes in its architecture and performance characteristics, this dissertation implements an automatic tuning system to generate high-performance matrix-multiplication implementation on graphics hardware. The automatic tuning system uses a parameterized code generator to generate multiple versions of matrix multiplication, whose performances are empirically evaluated by actual execution on the target platform. An ad-hoc search engine is employed to search over the implementation space for the version that yields the best performance. In contrast to similar systems on CPUs, which utilize cache blocking, register tiling, instruction scheduling tuning strategies, it identifies and exploits several tuning strategies that are unique for graphics hardware. These tuning strategies include optimizing for multiple-render-targets, SIMD instructions with data packing, overcoming limitations on instruction count and dynamic branch instruction. The generated implementations have comparable performance with expert manually tuned version in spite of the significant overhead incurred due to the use of the high-level BrookGPU language.

Frequent pattern mining is a fundamental problem in data mining and a large number of distinct algorithms have

been proposed to solve it efficiently. However, no single algorithm outperforms all the others since their relative performance highly depends on the characteristics of the input data. In the dissertation, we present a machine learning based approach to select the best frequent pattern mining algorithm based on the input characteristics. Three of the fastest publicly available algorithms, FP_Growth, LCM and Eclat, were extensively evaluated using synthetic data sets. The results of these evaluations were used to train a support-vector machine (SVM) prediction system, which is then used at runtime to predict the best mining algorithm for real-world data sets. Our experiments show that the runtime prediction overhead is negligible and that the trained SVM prediction system usually identifies the best algorithm. In case of misprediction, the selected algorithm is still competitive in performance. Thus, our approach generates an adaptive, SVM-based algorithm whose performance is close to optimal and is significantly better than that of any of the single algorithms.

To Minglong and my parents.

Acknowledgments

First of all, I feel so lucky to have found professor Marc Snir to be my Ph.D. advisor. I clearly remember when I first started my Ph.D. study at UIUC, I almost knew nothing about computer architecture. Marc brought me into the fascinating world of computer architecture and performance tuning. He taught me how to break down a big complex task into smaller manageable concrete goals and always identify the most essential part of the problem to work on. Often times his mathematician's instinct helped me understand complex engineering problems from simple but beautiful mathematical angles. His strict attitude toward scientific truth not only teaches me how to do research but also how to be a righteous researcher. I really cherish the enjoyable experience of working with a nice and smart advisor like him. I would also like to thank Karen Stahl and Jennifer Reynolds for their consistent help throughout my working with professor Marc Snir.

I am also greatly indebted to professor Maria Garzaran and professor David Padua. I started working with Maria and David in the last two years of my Ph.D. study. It was professor David Padua who helped me identify the interesting and challenging problem of algorithm selection for frequent pattern mining. In the initial stage of the project, David and Maria both had a lot of fruitful discussions with me. Their insights and experiences enlightened me and helped me quickly get on track. I learned a lot from David and Maria about how to clearly define a goal in research and present it to others. I am so thankful to Maria that she also served on my thesis committee even after she just gave birth to a baby.

I would like to thank professor Jiawei Han and professor Dan Roth for serving on my thesis committee. Professor Jiawei Han is a pioneering researcher in data mining who first invented the FP_Growth algorithm which I studied extensively in my thesis. Professor Jiawei Han always kindly replies to my request very promptly, even in short notice. Professor Dan Roth is an expert researcher in machine learning. He is also a great teacher in transferring his knowledge and experience in machine learning to his students. It is actually from his class of "Introduction to machine learning" where I first came up with the idea of using Support Vector Machine to solve the algorithm prediction problem.

UIUC has one of the best computer science programs in the world. I was fortunate to have taken classes and learned from many of the elite professors: Sarita Adve (CS321: Computer System Organization), Vikram Adve

(CS321: Programming Languages and Compilers), Sariel Har-Peled (CS373: Combinatorial Algorithms), David Padua (CS320: Introduction to Parallel Programming), Roy Campbell (CS423: Advanced Operating System), Josep Torrelas (CS533: Advanced Parallel Architecture), Mathew Frank (ECE511: Computer Architecture). I also am in debted to professor John Hart who generously lent me a few graphics cards for my research on automatic tuning on GPU.

I met many friends at UIUC who had helped me a lot and made my stay at the corn field much enjoyable, they are: Mingliang Wei, Jing Yu, Eun-Gyu Kim (now at Yahoo!), Sara Sadeghi Baghsorkhi, Jay H. Byun, Danny Dig, Geraud P Krawezik, Qing Wu, Yue Zhou, Hong Cheng, Jing Jiang, Chao Liu, Chih-Wei Hsu, Chun-Cheng Chen, Shan Lu, Yi-Ting Chou, Pei-Hsi Chen, Yuen-tien Lee, Bin Tan, Zheng Shao (now at Yahoo!), Jianqing Zhang, Han Liu (now at CMU), Lin Tan, Weihang Jiang, Xiao Ma, Tian Xia, Pin Zhou (now at IBM), Wei Liu (now at Intel), Wanmin Wu, Zhenmin Li, Min Wu (now at Morgan Stanley), Qixin Wang, Hui Fang, Xiaoming Li (now at Delaware), Gang Ren (now at Google), Shengnan Cong (now at Intel), Chengkai Li, Liqian Luo, Chengdu Huang, Qiaozhu Mei, Xuanhui Wang, Nathan Carr (now at Adobe Systems), Shipeng Jiang, Zhen Sun (now at Microsoft), Jin Liang.

During my three months' Summer internship at the IBM Almaden research center in 2005, I met a few very nice and unforgettable people: Lan Huang, Shauchi Ong, Windsor Hsu, Ying Chen.

Last but not least, I can't thank enough my family members, my dear wife – Minglong and my great parents. Without their consistent support and encouragement, I would not have been able to complete the long journey of my Ph.D. study. This dissertation is dedicated to them for the love and support.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ATLAS	Automatically Tuned Linear Algebra Software.
BLAS	Basic Linear Algebra Subroutines.
DFT	Discrete Fourier Transform.
DSP	Digital Signal Processing.
FFT	Fast-Fourier Transform.
GPU	Graphics Processing Unit.
GPGPU	General Purpose computation on GPUs
LAPACK	Linear Algebra PACKage.
LCM	Linear time Closed itemset Miner.
MMM	Matrix-matrix multiplication.
PHiPAC	Portable High Performance ANSI C.
RGBA	Red, Green, Blue, Alpha.

Chapter 1

Introduction

1.1 Motivation

The pursuit of higher performance has been one of the persistent themes of computer system research. In recent years, the fast evolution, increasing complexity, and increasing diversity of computing platforms poses a major challenge to developers of high-performance applications: software development has become an interdisciplinary task that requires the programmer to have specialized knowledge in algorithms, programming languages, and computer architectures. Furthermore, tuning even simple programs tends to be expensive because it requires an intense and sustained effort—which can stretch over a period of weeks or months, if not years—from the technically best expert programmers. But the manual tuning or adaptation of software implementations to a particular platform also leads to a vicious cycle: the code developer invests tremendous energy tinkering with the implementation to exploit in the best way the computing resources available, simply to realize that the hardware infrastructure has become obsolete in the interim as an effect of the relentless technological advances reflecting Moore's Law. This has led to large repositories of applications that were once well adapted to the existing computers of the time, but continue to persist because the cost involved in updating them to the current technology is simply too large to warrant the effort. To break successfully this cycle, it is necessary to rethink the process of designing and optimizing software in order to deliver to the user the full power of optimized implementations on the available hardware resources.

Traditionally, for portability reasons, tuning is usually confined to a library of a small number of heavily-used computational kernels. For any given architecture, a considerable investment in time and effort is needed to optimize performance for it. The process of customizing a numerical kernel's source code to optimize performance requires a comprehensive understanding of the exploitable hardware resources of that architecture. This primarily includes the memory hierarchy and how it can be utilized to maximize data-reuse, as well as the functional units and registers and how these hardware components can be programmed to generate the correct operands at the correct time. Clearly, the size of the various cache levels, the latency of floating point instructions, the number of floating point units and other hardware constants are essential parameters that must be taken into consideration. To realize high performance on the complicated architectures for even the simplest of operations often requires tedious, hand-coded, programming efforts.

It would be ideal if compilers are capable of performing the optimization needed automatically. However, compiler technology is far from mature enough to perform these optimizations automatically. Since this time-consuming customization process must be repeated whenever a slightly different target architecture is available, or even when a new version of the compiler is released, the relentless pace of hardware innovation makes the tuning of numerical libraries a constant burden.

The difficult search for high performance software is compounded by the complexities of porting and tuning numerical libraries to run on specialized architectures, which dramatically differ from traditional general purpose architectures and potentially outperform significantly traditional ones for software that fits in their special niches. Given the fact that the performance of common computing platforms has increased exponentially in the past few years, especially some specialized processors, scientists and engineers have acquired legitimate expectations about being able to immediately exploit these available resources at their highest capabilities. Fast, accurate, and robust software have to be encoded in libraries that are highly portable and optimizable across a wide range of systems in order to be exploited to their fullest potential.

In addition to the problem of optimizing kernels across architectures, often times there are several formulations of the same operation that can be chosen. The variations can be the choice of data structures or the choice of basic algorithms. The variation in the best algorithm usually depends on input characteristics, where offline tuning cannot help. Intelligent software adaptation at runtime that takes into account input features is required.

The ever increasingly complex computer architectures and the large space of variations in data structures and algorithms make it an extremely difficult research problem to find the best form of software for achieving the highest performance. Therefore, new methodology of software performance optimization is needed.

1.2 Empirical search based performance tuning

To attack the aforementioned challenges, a new software performance tuning methodology, *empirical search based performance tuning*, has been proposed and applied to a few application domains. The idea is to represent the performance tuning problem as the problem of finding the best version out of multiple versions of implementations capable of solving the same computing task. The different versions of code usually come from applying various compiler transformations, software restructuring techniques, or choosing from different data structures, different algorithms etc. The best version refers to the one that optimizes some targeted metric, that can be any meaningful measure of efficiency, such as CPU cycles, power consumption, throughput, load-balance, etc. Usually minimizing CPU cycles is the main concern of software performance tuning.

To find the best version, traditionally software performance tuning is carried out by experts who have intimate

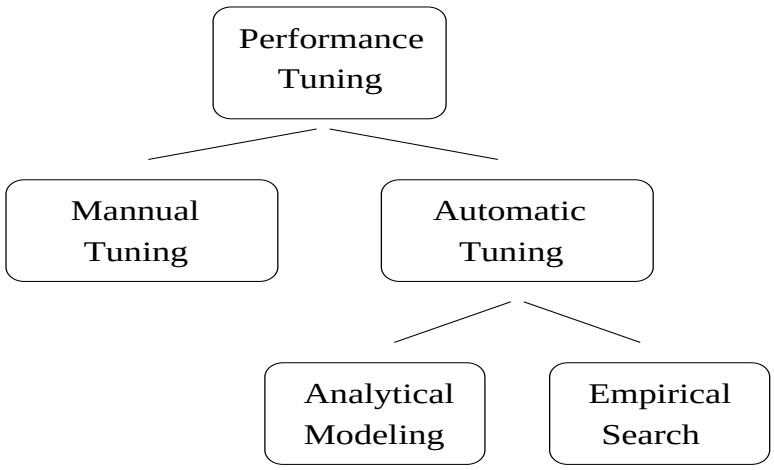


Figure 1.1: Types of performance tuning

knowledge of the underlying architecture and the software domains. As the relative cost of expert man-power increases and hardware evolution speeds up, automatic software performance tuning attracts more and more interests. There are two alternative ways of automating software performance tuning. One is through analytical modeling, the other is through trial-and-search or empirical search. Analytical modeling method only applies when the interactions between affecting factors are well understood and can be mathematically modeled as a constrained optimization problem. However, this prerequisite is increasingly difficult to satisfy as the hardware become more complex and dynamic. The empirical search method however relies on empirical evaluations of multiple code versions and an intelligent search strategy to efficiently prune and explore the space of code versions to find the best. The empirical search based tuning has gained a great deal of popularity in recent few years. Previous research [YLR+03] has shown that sometimes analytical modeling can be used together with empirical search to speed up its search process. Figure 1.1 depicts the categorization of performance tuning.

Several research projects that applied the empirical search based automatic tuning have been proved to be successful. As we will further describe in Chapter 2, the previous research in the area focuses on general purpose processors and some classic well-studied scientific computing tasks. In recently years, specialized architectures such as graphics hardware (GPUs), network processors, multimedia processors have enjoyed a faster evolution pace than general purpose processor in both the peak performance and architecture innovation. There is growing interest to perform conventional high performance computation tasks on them. How to best utilize the fast and dynamic architecture poses new challenges. Empirical search based automatic performance exhibits great potential due to its capability to automatically adapt to fast changing architectures.

According to the optimization target, performance tuning can be coarsely divided into two categories: architecture oriented tuning and input oriented tuning. The first kind typically involves tuning a set of kernels offline, where the

efficiency of computation is mostly determined by the interaction between the software and the underlying architecture. The second kind requires online adaptation that takes into account the input characteristics and chooses the best variations in algorithms and/or data structures or problem formulations. In this thesis, we refer to the first kind of tuning as the kernel tuning problem, refer to the second kind as the algorithm selection problem. Note that the algorithm selection problem is usually harder than the kernel tuning problem because it has to occur at run-time and must have a low overhead. In reality, complex computing tasks might require performance tuning in both aspects: each candidate algorithm is a tuned kernel whose performance is determined by the interaction of the software implementation and the underlying architecture; algorithm selection is performed at run-time to choose the best tuned kernel for each given input.

1.2.1 Automatic kernel tuning

Kernels usually refer to heavily-used code snippets that form the building blocks of higher level computations. For example, Discrete Fourier Transforms can be decomposed into a set of smaller transformations. Large linear algebra routines are carried out by applying lower level primitive operations, such as vector-vector, matrix-vector and matrix-matrix operations corresponding to BLAS (Basic Linear Algebra Subprogram) level I, II and III respectively. The potential of achieving higher performance comes from restructuring the code to cater to the underlying architecture features. For general purpose processors, it usually involves optimizing the usage of memory hierarchy, scheduling hardware resources such as functional units, registers. Using the compiler optimization at its best, optimizing the operations to account for many parameters such as blocking factors, loop unrolling depths, software pipelining strategies, loop ordering, register allocations and instruction scheduling, etc. are crucial for machine-specific factors affecting performance. The hardware parameters such as cache size, memory latency, instruction latency, number of registers play critical roles in affecting the performance.

1.2.2 Automatic algorithm selection problem

When no algorithm dominates, one is faced with the problem of deciding which algorithm to use; in 1976 Rice named this the *algorithm selection problem* [Ric76]. The basic idea is that the algorithms provide a classification of inputs, with each class consisting of the set of inputs for which a particular algorithm is better. One wishes to find a classifier that associates (with high probability) each input to the correct class. One approach that has been studied by several authors [LBNA$^+$03, LGP05, TTT$^+$05] is to use machine learning algorithms for this purpose: A training set of inputs are selected; the algorithms are run on these inputs and each input is labeled with the best algorithm; the classifier is trained to identify the labels; the resulting classifier is then used during execution to select which algorithm to run.

1.3 Contribution of the thesis

Automatic tuning matrix multiplication on GPUs

In the past decade, graphics hardware, a.k.a graphics processing unit (GPU), has enjoyed a faster growth than what Moore's law dictates. By utilizing extensive parallel and vector processing units, modern graphics hardware dramatically outperforms the most advanced CPU. As a result, a growing interest has been raised in performing general purpose GPU computation, namely GPGPU. GPGPU's ultimate goal is to enable the GPU as a powerful coprocessor to CPU and offload computationally intensive tasks. GPU algorithms for dense matrix multiplication [AM03, HCH03, LM01], FFT [MA03], database operations [GLW+04], sparse matrix conjugate gradient solver [BFGS03], ray tracing [CHH02, PBMH02], etc. have been studied and demonstrated to work on graphics hardware.

In order for general purpose computing to fully utilize the power of graphics hardware, it is critical to tune software to cater to the underlying architecture. Tuning software performance for a particular hardware architecture usually requires detailed knowledge of that architecture. However this requirement is difficult to meet for graphics hardware because: First, most GPU vendors do not release their products' architectural internal details, such as cache organization, rasterization algorithm. Second, programmers have only indirect control of the hardware through a vendor supplied driver, which dynamically loads, compiles, optimizes and executes the application supplied programs. The optimizations done by the dynamic compiler inside the driver are transparent to the programmer. Third, graphics hardware evolves fast. Every six months, GPU vendors introduce a new generation of graphics cards. A well tuned program for a particular generation of architecture may turn out to perform badly on its successor generation due to changes in the underlying architecture.

The difficulty in tuning performance for a fast-evolving hardware architecture makes self-adaptive software desirable. Automatic software tuning for general purpose processors has been studied for some years. Previous research in this field centered around automatic generation of high-performance numerical routines, such as dense and sparse matrix operations [BAwCD97, IYV04, WPD01], sorting [LGP05], FFT [FJ05], signal processing [PSX+04], by improving software's spatial/temporal locality, instruction scheduling and dynamic algorithm selection to cater to modern processors' deep memory hierarchy and pipeline design.

Automatic tuning for graphics hardware presents new challenges. First, graphics hardware uses a non-traditional programming model that complicates the mapping of algorithms to the hardware. Second, as graphics processors are vastly different from general purpose processors, new tuning strategies are needed. Common strategies for CPUs, such as cache blocking, loop unrolling, software pipelining rarely directly work on GPUs. Third, since graphics hardware's architectural details and machine parameters are usually withheld by vendors, the use of performance models either to prune search space of automatic tuning or to replace search is more difficult to realize on graphics hardware.

The work done in this thesis is the first attempt to implement an automatic tuning system to generate numerical

libraries for graphics hardware. More specifically, it studies automatic generation of high-performance matrix multiplication on graphics hardware, as matrix multiplication is the most important building block for a variety of numerical libraries. In contrast to ATLAS [WPD01], which utilizes register tiling, cache blocking and instruction scheduling to achieve high performance on pipelined processor with deep memory hierarchy, our approach automatically tunes matrix multiplication to graphics hardware's unconventional architecture features, such as SIMD instructions with swizzling and smearing, multiple-render-targets, limited instruction count, limitation on branch instruction, varying shader models, etc. Our automatic tuning system is capable to generate matrix multiplication implementations with comparable performance to expert manually tuned version despite the significant overhead incurred due to the use of a poorly supported high level language.

Automatic algorithm selection for frequent pattern mining

Frequent pattern mining is the problem of finding subsets of items that occur frequently in a set of transactions. It has many applications such as finding association rules, correlations, or causality. Since the introduction by Agrawal et al. [AIS93], a large number of algorithms have been proposed. During the FIMI (Frequent Itemset Mining Implementations) workshops [GZ03a, JGZ04] different implementations of well-known pattern mining algorithms were submitted and their relative performances were evaluated on a few datasets.

In the thesis, we present an SVM (support vector machine) [Vap95] based learning method to train a classification system that selects a frequent pattern mining algorithm based on input characteristics. Our main contribution is to demonstrate the viability of this machine learning approach for frequent pattern mining algorithm selection. In particular (1) we identify a set of input features that can guide the selection and (2) we show how to generate synthetic data sets that are representative of the real-world data sets and that can be used to train the input classifier. As a result of our contribution, we obtain a hybrid algorithm that is better that any of the three selected algorithms (LCM, FP_Growth, and Eclat). Our experiments show that the performance of the hybrid algorithm is on the average only 12.5% worse than that of the optimal algorithm, and 65.3% better than LCM that obtained the best average performance for all the tested inputs. We believe that the approach will extend to other choices of basic frequent pattern mining algorithms.

1.4 Organization of the thesis

The remainder of the thesis is organized as follows: in Chapter 2 we introduce some related research projects to the thesis. Chapter 3 describes the streaming programming model and the architectures for graphics hardware. Chapter 4 elaborates the algorithms for performing matrix multiplication on graphics hardware. Chapter 5 presents our automatic matrix multiplication tuning system for graphics hardware and its experimental results. In Chapter 6, we introduce

the frequent pattern mining problem, the algorithm selection problem for it and the support vector machine tool. In Chapter 7, we present our automatic algorithm selection system for frequent pattern mining using support vector machine. Chapter 8 concludes the thesis with some potential future research directions.

Chapter 2

Related Work

Automatic library generation via intelligent empirical search has been studied in many projects. In this Chapter, we will briefly introduce some of the most relevant and representative projects. They are categorized into “kernel tuning” and “algorithm selection” types.

2.1 Kernel Tuning

2.1.1 ATLAS

ATLAS (Automatically Tuned Linear Algebra Software) [WPD01] is a tool for the automatic generation of optimized numerical software for modern computer architectures and compilers. This tool has initially focused on level three BLAS operations (matrix-matrix multiplications) and also a few routines from LAPACK (Linear Algebra PACKage) that have high potential for optimization. Traditionally, the optimization of these routines has been a tedious, architecture dependent, hand coding process. Codes automatically generated by ATLAS have been able to meet and even exceed the performance of the vendor supplied, hand-optimized BLAS, on a range of platforms.

It uses a parameterized code generators that can generate multiple versions of matrix multiplication code according to input tuning parameter values. These tuning parameters control different software transformations that affect L1-cache blocking, register tiling, instruction scheduling. The generated code’s performance is empirically evaluated by actual execution. A search engine is then used to search over the implementation space for the version that yields the best performance. An alternative approach to empirical-search based tuning is to use analytical model to determine the best tuning parameter values [YLR+03].

2.1.2 PHiPAC

The PHiPAC (Portable High Performance ANSI C) [BAwCD97] is another automatically tuned library for BLAS III compatible fast matrix matrix multiply. First, rather than code by hand, it uses parameterized code generators whose parameters are germane to the resulting machine performance. Second, the generated code follows the PHiPAC

(Portable High Performance Ansi C) coding suggestions that include manual loop unrolling, explicit removal of unnecessary dependencies in code blocks (if not removed, C semantics would prohibit many optimizations), and use of machine friendly C constructs. Third, it takes advantage of search scripts that, for a given code generator, find the best set of parameters for a given architecture/compiler.

2.1.3 Sparsity

The SPARSITY [IYV04] system targets at the problem of sparse matrix-vector multiplication, which is an important computational kernel that performs poorly on most modern processors due to a low compute-to-memory ratio and irregular memory access patterns. Optimization is difficult because of the complexity of cache-based memory systems and because performance is highly dependent on the non-zero structure of the matrix. SPARSITY addresses these problems by allowing users to automatically build sparse matrix kernels that are tuned to their matrices and machines. SPARSITY combines traditional techniques such as loop transformations with data structure transformations and optimization heuristics that are specific to sparse matrices. It provides a novel framework for selecting optimization parameters, such as block size, using a combination of performance models and search.

2.1.4 FFTW

The FFTW library [Fri99, FJ05] for computing the discrete Fourier transform (DFT) has gained a wide acceptance in both academia and industry, because it provides excellent performance on a variety of machines (even competitive with or faster than equivalent libraries supplied by vendors). In FFTW, most of the performance-critical code was generated automatically by a special-purpose compiler, called genfft, that outputs C code. Written in Objective Caml, genfft can produce DFT programs for any input length, and it can specialize the DFT program for the common case where the input data are real instead of complex. Unexpectedly, genfft "discovered" algorithms that were previously unknown, and it was able to reduce the arithmetic complexity of some other existing algorithms. FFTW employs a high-level description of execution plan for decomposing large Fourier transform into smaller specially optimized kernels, named "codelet". A dynamic programming based search process is performed at runtime, when input transform size is known, to find the best execution plan.

The innovation in FFTW consists in having a variety of composable solvers, representing different FFT algorithms and implementation strategies, whose combination into a particular plan for a given size can be determined at runtime according to the characteristics of your machine/compiler. This peculiar software architecture allows FFTW to adapt itself to almost any machine. The main reasons for FFTW's superior performance are the following: 1. FFTW uses a variety of FFT algorithms and implementation styles that can be arbitrarily composed to adapt itself to a machine. 2. FFTW uses a code generator to produce highly-optimized routines for computing small transforms. 3. FFTW uses

explicit divide-and-conquer to take advantage of the memory hierarchy.

2.1.5 SPIRAL

The Spiral [PSX+04] system extends FFTW's idea to more general signal processing with high-level tensor notations and genetic algorithms based search. SPIRAL is a generator of libraries for fast software implementations of signal processing transforms. These libraries are adapted to the computing platform and can be re-optimized as the hardware is upgraded or replaced. The main components of SPIRAL include: a mathematical framework that concisely describes signal transforms and their fast algorithms, a formula generator that captures at the algorithmic level the degrees of freedom in expressing a particular signal processing transform; a formula translator that encapsulates the compilation degrees of freedom when translating a specific algorithm into an actual code implementation; and, finally, an intelligent search engine that finds within the large space of alternative formulas and implementations the "best" match to the given computing platform.

For a specified transform, SPIRAL automatically generates high performance code that is tuned to the given platform. SPIRAL formulates the tuning as an optimization problem, and exploits the domain-specific mathematical structure of transform algorithms to implement a feedback-driven optimizer. Similar to a human expert, for a specified transform, SPIRAL "intelligently" generates and explores algorithmic and implementation choices to find the best match to the computer's micro-architecture. The "intelligence" is provided by search and learning techniques that exploit the structure of the algorithm and implementation space to guide the exploration and optimization. SPIRAL generates high performance code for a broad set of DSP transforms including the discrete Fourier transform, other trigonometric transforms, filter transforms, and discrete wavelet transforms. Experimental results show that the code generated by SPIRAL competes with, and sometimes outperforms, the best available human tuned transform library code.

2.2 Algorithm Selection

2.2.1 Dynamic Sorting

Sorting is an example of applications which need to be tuned not only to the architecture but also to input data's characteristics. There are numerous efficient sorting algorithms favoring different kinds of input data. In the dynamic sorting library, Li et al. [LGP04, LGP05] study machine learning techniques that extend empirical search to the generation of algorithms whose performance depends on both the input characteristics and the architecture of the target machine.

They observe that various sorting algorithms perform differently depending on input characteristics. They first

study if it is possible to predict and select the best sorting algorithm for a specific input. They develop a machine-learning based technique to find the mapping from architectural features and input characteristics to the selection of best algorithm. The mapping is used at runtime to make selection of sorting algorithms. Experiments show that their approach always predict the best sorting algorithm and the runtime overhead due to the selection is below 5%. On top of their first study of selecting a "pure" sorting algorithm at the outset of the computation as a function of the input characteristics, they further develop algorithms and a classifier system to build hierarchically-organized hybrid sorting algorithms capable of adapting to the input data. The results show that such algorithms generated using the approach presented are quite effective at taking into account the complex interactions between architectural and input data characteristics and that the resulting code performs significantly better than conventional sorting implementations. In particular, the routines generated using their approach perform better than all the commercial libraries that they tried including IBM ESSL (Engineering Scientific Subroutine Library), INTEL MKL (Math Kernel Library) and the C++ STL (Standard Template Library) for the chosen input distribution.

Chapter 3

Streaming Processing and Graphics Hardware Architecture

The advancement in semi-conductor fabrication technology has made it possible to incorporate enormous computation resources into a single chip. How to translate the increase in transistor density into the increase in performance becomes a big challenge for computer architects. Designers of graphics hardware have been quite successful in meeting the challenges by utilizing an unconventional architecture which takes advantage of the so-called streaming processing model.

3.1 Technology Trend

Because of the increase in clock speed, the amount of time it takes for a signal to travel across the entire chip, measured in clock cycles, is also increasing. Nowadays, on the fastest processors, it takes multiple clock cycles to send a signal from one side of chip to another. This worsens new generations of processors. Put it another way, the communication cost increases relative to computation cost.

In the mean time, memory latency will continue to improve more slowly than bandwidth, designers must implement solutions that can tolerate larger and larger amounts of latency by continuing to do useful work while waiting for data to return from operations that take a long time.

Power consumption also starts to be a big challenge to architecture designers. Although smaller transistors require less power than larger ones, the number of transistors on a single processor die is rising faster than the amount at which power per transistor is falling. Consequently, each generation of processors requires more power: it is estimated that the maximum power allowed for 2008 chips with a heat sink is 198 watts. This power constraint will be one of the primary limitations of future processors; the future figure of merit may no longer be the number of operations per second but instead the number of operations per second per watt.

In addition to the above technology trends, the fact that real-time gaming has emerged into a huge market provides the economical foundation for enormous investment in specialized software/hardware systems to optimize its performance. The need for specialized hardware is especially stressed when its performance is far from optimal on general purpose processors. The reason is because conventional general purpose processors target general programs

that have less parallelism, more complex requirements, and lower performance goals than graphics rendering pipeline. As a result, CPU designers made different design choices that result in poor mapping to the graphics pipeline. In addition, CPU memory systems are optimized for minimum latency rather than the maximum throughput targeted by GPU memory systems.

Therefore, to optimize graphics application's performance, specialized graphics hardware that takes advantage of graphics application's enormous data parallelism has been designed and implemented. Before we dive into graphics architecture, we first described the streaming programing model for graphics applications.

3.2 Streaming Programming Model

In this section, we describe the streaming programming model used by graphics hardware. This section is mostly based on the description from [Owe05]. In the stream programming model, all data is represented as a stream, which we define as an ordered set of data of the same data type. That data type can be simple (a stream of integers or floating-point numbers) or complex (a stream of points or triangles or transformation matrices). While a stream can be any length, we will see that operations on streams are most efficient if streams are long (hundreds or more elements in a stream). Allowed operations on streams include copying them, deriving sub-streams from them, indexing into them with a separate index stream, and performing computation on them with kernels.

A kernel operates on entire streams, taking one or more streams as inputs and producing one or more streams as outputs. The defining characteristic of a kernel is that it operates on entire streams of elements as opposed to individual elements. The most typical use of a kernel is to evaluate a function on each element of an input stream (a "map" operation); for example, a transformation kernel may project each element of a stream of points into a different coordinate system. Other desirable kernel operations include expansions (in which more than one output element is produced for each input element), reductions (in which more than one element is combined into a single output element), or filters (in which a subset of input elements are output).

Kernel outputs are functions only of their kernel inputs, and within a kernel, computations on one stream element are never dependent on computations on another element. These restrictions have two major advantages. First, the data required for kernel execution is completely known when the kernel is written (or compiled). Kernels can thus be highly efficient when their input elements and their intermediate computed data are stored locally or are carefully controlled global references. Second, requiring independence of computation on separate stream elements within a single kernel allows mapping of what appears to be a serial kernel calculation onto data-parallel hardware.

In the stream programming model, applications are constructed by chaining multiple kernels together. For instance, implementing the graphics pipeline in the stream programming model involves writing a vertex program kernel, a

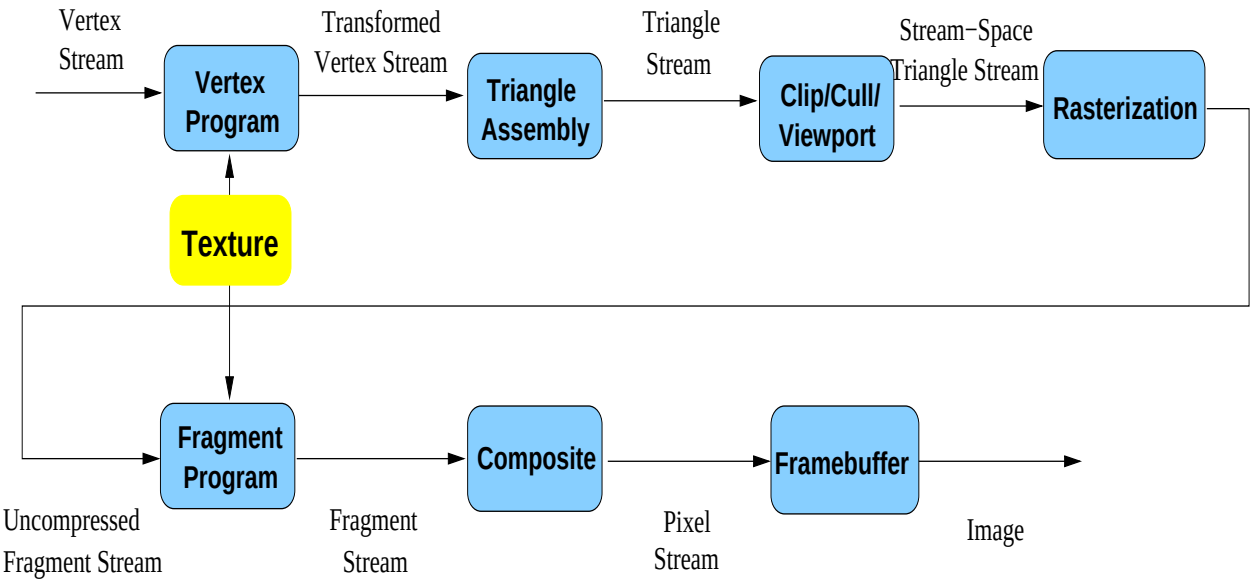


Figure 3.1: Mapping the Graphics Pipeline to the Stream Model

triangle assembly kernel, a clipping kernel, and so on, and then connecting the output from one kernel into the input of the next kernel. Figure 3.1 shows how the entire graphics pipeline maps onto the stream model. This model makes the communication between kernels explicit, taking advantage of the data locality between kernels inherent in the graphics pipeline.

The graphics pipeline is a good match for the stream model for several reasons. The graphics pipeline is traditionally structured as stages of computation connected by data flow between the stages. This structure is analogous to the stream and kernel abstractions of the stream programming model. Data flow between stages in the graphics pipeline is highly localized, with data produced by a stage immediately consumed by the next stage; in the stream programming model, streams passed between kernels exhibit similar behavior. And the computation involved in each stage of the pipeline is typically uniform across different primitives, allowing these stages to be easily mapped to kernels.

The stream model enables efficient computation in several ways. Most important, streams expose parallelism in the application. Because kernels operate on entire streams, stream elements can be processed in parallel using data-parallel hardware. Long streams with many elements allow this data-level parallelism to be highly efficient. Within the processing of a single element, we can exploit instruction-level parallelism. And because applications are constructed from multiple kernels, multiple kernels can be deeply pipelined and processed in parallel, using task-level parallelism.

Dividing the application of interest into kernels allows a hardware implementation to specialize hardware for one or more kernels' execution. Special-purpose hardware, with its superior efficiency over programmable hardware, can thus be used appropriately in this programming model.

Finally, allowing only simple control flow in kernel execution (such as the data-parallel evaluation of a function on

each input element) permits hardware implementations to devote most of their transistors to data path hardware rather than control hardware.

Efficient communication is also one of the primary goals of the stream programming model. First, off-chip (global) communication is more efficient when entire streams, rather than individual elements, are transferred to or from memory, because the fixed cost of initiating a transfer can be amortized over an entire stream rather than a single element. Next, structuring applications as chains of kernels allows the intermediate results between kernels to be kept on-chip and not transferred to and from memory. Efficient kernels attempt to keep their inputs and their intermediate computed data local within kernel execution units; therefore, data references within kernel execution do not go off-chip or across a chip to a data cache, as would typically happen in a CPU. And finally, deep pipelining of execution allows hardware implementations to continue to do useful work while waiting for data to return from global memories. This high degree of latency tolerance allows hardware implementations to optimize for throughput rather than latency.

The stream programming model structures programs in a way that both exposes parallelism and permits efficient communication. Expressing programs in the stream model is only half the solution, however. High-performance graphics hardware must effectively exploit the high arithmetic performance and the efficient computation exposed by the stream model. How do we structure a hardware implementation of a GPU to ensure the highest overall performance?

The first step to building a high-performance GPU is to map kernels in the graphics pipeline to independent functional units on a single chip. Each kernel is thus implemented on a separate area of the chip in an organization known as task parallel, which permits not only task-level parallelism (because all kernels can be run simultaneously) but also hardware specialization of each functional unit to the given kernel. The task parallel organization also allows efficient communication between kernels: because the functional units implementing neighboring kernels in the graphics pipeline are adjacent on the chip, they can communicate effectively without requiring global memory access.

Within each stage of the graphics pipeline that maps to a processing unit on the chip, GPUs exploit the independence of each stream element by processing multiple data elements in parallel. The combination of task-level and data-level parallelism allows GPUs to profitably use dozens of functional units simultaneously.

Inputs to the graphics pipeline must be processed by each kernel in sequence. Consequently, it may take thousands of cycles to complete the processing of a single element. If a high-latency memory reference is required in processing any given element, the processing unit can simply work on other elements while the data is being fetched. The deep pipelines of modern GPUs, then, effectively tolerate high-latency operations.

For many years, the kernels that make up the graphics pipeline were implemented in graphics hardware as fixed-function units that offered little to no user programmability. In 2000, for the first time, GPUs allowed users the opportunity to program individual kernels in the graphics pipeline. Today's GPUs feature high-performance data-

parallel processors that implement two kernels in the graphics pipeline: a *vertex program* that allows users to run a program on each vertex that passes through the pipeline, and a *fragment program* that allows users to run a program on each fragment. Both of these stages permit single-precision floating-point computation. Although these additions were primarily intended to provide users with more flexible shading and lighting calculations, their ability to sustain high computation rates on user-specified programs with sufficient precision to address general-purpose computing problems has effectively made them *programmable stream processors* – that is, processors that are attractive for a much wider variety of applications than simply the graphics pipeline.

3.3 Graphics Hardware Architecture

In this section, we introduce the graphics architecture exemplified by the GeForce 6 series GPU architecture [KF05]. Readers interested in deeper treatment of graphics hardware architecture are referred to [Ake93, MBDM97] and to vendor specifications of graphics hardware products.

3.3.1 General Architecture

Most modern graphics hardware has multiple vertex processors and fragment processors. Figure 3.2 depicts a conceptual view of a graphics hardware with sixteen fragment processors and six vertex processors[1]. Vertex processors perform geometric transformations and lighting operations on geometric primitives. After vertices have been projected to screen space, the rasterizer calculates fragment[2] information by interpolating vertex information. Then rasterizer assigns fragment-rendering tasks to fragment processors. A fragment processor renders one fragment at a time. After a fragment has been rendered, the fragment processor writes the final color information into the fragment's designated location in the frame buffer for display.

The graphics hardware's memory subsystem, namely texture memory, is mainly designed to support texture mapping operations. Most modern GPUs do not support write operations by fragment processors to the texture memory. Fragment processors can only perform writes to the frame buffer. If a program needs to store intermediate results to texture memory, it can either copy the intermediate results from the frame buffer to texture memory, or use a render-to-texture technique, which allows rendering results in the frame buffer to be used as input texture for further computations. Most modern GPUs do not allow vertex processor to access texture memory [3].

Data organization in graphics hardware's cache, namely texture cache, is also designed to improve spatial locality of texture mapping operation. Optimizations for temporal locality of texture mapping are implemented in the rasterizer

[1] Note that figure 3.2 ignores many graphics related components. It does not represent any real graphics hardware architecture but only serves to facilitate the understanding of general purpose computing on GPU.

[2] In graphics terminology, "fragment" refers to screen element before shading, "pixel" refers to screen element after shading.

[3] Latest GPUs start to add the support for vertex processors to access texture memory.

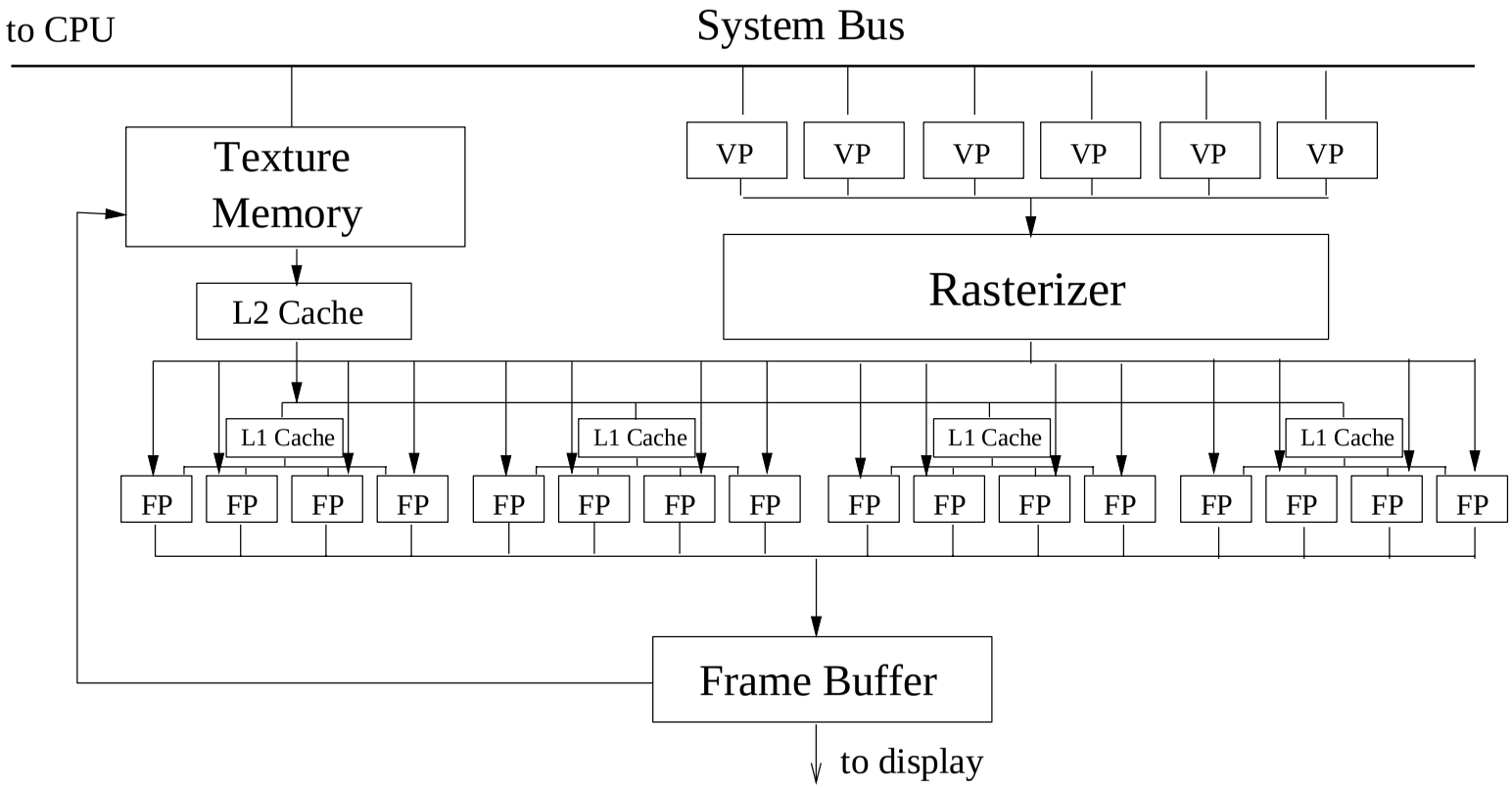


Figure 3.2: Architecture of Graphics Hardware

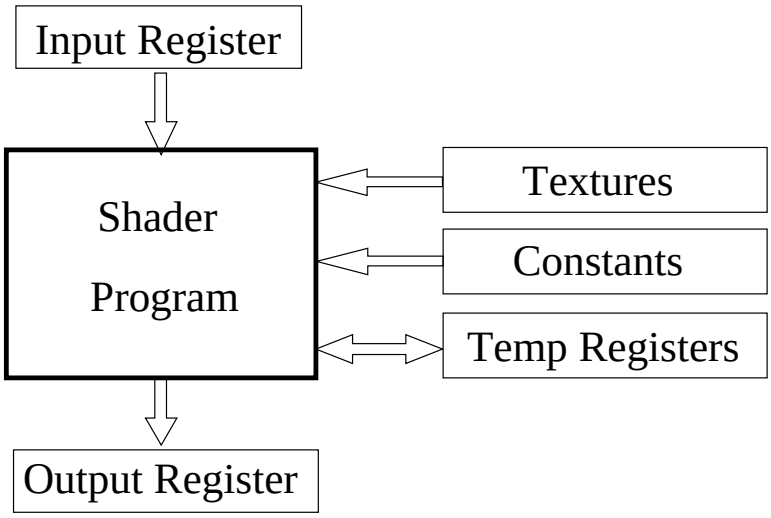


Figure 3.3: Programming model for GPU

by rasterizing fragments in some special order. As cache organization and rasterization algorithms used in GPU products are usually considered as commercial secrets, there is little public knowledge about their internal details.

Due to vertex processors' inability to access texture memory and the rasterizer's lack of programmability, most GPGPU applications rely on fragment processors to perform intensive computation. A program executed by a fragment processor is called "fragment program" or "fragment shaders". Each execution of a "fragment program" renders one fragment. Therefore, one can consider a GPU as a stream processor performing the same kernel function (fragment program) on streams of data elements (fragments).

The programming model for fragment processors is illustrated in figure 3.3. A fragment program reads input data from input registers filled by the rasterizer. It can read a number of constant values set by the host application, read from texture memory, and read and write a number of temporary registers. After the execution, the result in the output registers is written into corresponding positions in the frame buffer for display.

3.3.2 Computational Concepts on GPUs

In this section, we expand the description on GPU's general architecture and it's available computational resources with a focus on general purpose computation on GPUs (GPGPU). This section is mainly based on descriptions from [Har05].

As described in the previous section, GPUs have two types of programmable processors: *vertex processors* and *fragment processors*. Vertex processors process streams of vertices (made up of positions, colors, normal vectors, and other attributes), which are the elements that compose polygonal geometric models. Computer graphics typically represents 3D objects with triangular meshes. The vertex processors apply a *vertex program* (sometimes called a *vertex*

shader) to transform each vertex based on its position relative to the camera, and then each set of three vertices is used to compute a triangle, from which streams of *fragments* are generated. A fragment can be considered a "proto-pixel." It contains all information needed to generate a shaded pixel in the final image, including color, depth, and destination in the frame buffer. The fragment processors apply a *fragment program* (sometimes called a *pixel shader*) to each fragment in the stream to compute the final color of each pixel.

Vertex Processors

Modern GPUs have multiple vertex processors (the NVIDIA GeForce 6800 Ultra and the ATI Radeon X800 XT both have six). These processors are fully programmable and operate in either SIMD- or MIMD-parallel fashion on the input vertices. The basic primitives of 3D computer graphics are 3D vertices in projected space, represented by an (x, y, z, w) vector, and four component colors stored as (red, green, blue, alpha) vectors (often abbreviated RGBA), where alpha typically represents an opacity percentage. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles.

Vertex processors are capable of changing the position of input vertices. If you think about this, the position of these vertices ultimately affects where in the image pixels will be drawn. An image is nothing but an array of memory; thus, because vertex processors can control where in memory data will be written, they are thus capable of scatter. However, most current vertex processors cannot directly read information from vertex elements in the input stream other than the one currently being processed. Therefore, they are incapable of gather. The NVIDIA GeForce 6 Series GPUs have a new feature called *vertex texture fetch* (VTF). This means that GeForce 6 vertex processors are capable of random-access memory reads. So, we can store part or all of our input stream data in a vertex texture and use VTF to implement a gather operation.

Fragment Processors

Modern GPUs also have multiple fragment processors (the NVIDIA GeForce 6800 Ultra and the ATI X800 XT both have 16). Like vertex processors, these are fully programmable. Fragment processors operate in SIMD-parallel fashion on input elements, processing four-element vectors in parallel. Fragment processors have the ability to fetch data from textures, so they are capable of gather. However, the output address of a fragment is always determined before the fragment is processed: the processor cannot change the output location of a pixel. Fragment processors are thus not natively capable of scatter.

For GPGPU applications, the fragment processors are typically used more heavily than the vertex processors. There are two main reasons for this. First, there are more fragment processors than vertex processors on a typical programmable GPU. Second, the output of the fragment processors goes more or less directly into memory, which can

be fed straight back in as a new stream of texture data. Vertex processor output, on the other hand, must pass through the rasterizer and fragment processors before reaching memory. This makes direct output from vertex processors less straightforward.

Rasterizer

As mentioned earlier, after the vertex processors transform vertices, each group of three vertices is used to compute a triangle (in the form of edge equations), and from this triangle a stream of fragments is generated. This work of generating fragments is done by the rasterizer. We can think of the rasterizer as an address interpolator. Later we show how memory addresses are represented as texture coordinates. The rasterizer interpolates these addresses and other per-vertex values based on the fragment position. Because it generates many data elements from only a few input elements, we can also think of the rasterizer as a data amplifier. These functions of the rasterizer are very specialized to rendering triangles and are not user-programmable.

Texture Unit

Fragment processors (and vertex processors on the latest GPUs) can access memory in the form of textures. We can think of the texture unit as a read-only memory interface.

Render-to-Texture

When an image is generated by the GPU, it can be written to frame-buffer memory that can be displayed, or it can be written to texture memory. This *render-to-texture* functionality is essential for GPGPU, because it is the only current mechanism with which to implement direct feedback of GPU output to input without going back to the host processor. (Indirect feedback is also available via *copy-to-texture*, which requires a copy from one location in the GPU's memory to another.) We can think of render-to-texture as a write-only memory interface.

The reason we don't consider the texture unit and render-to-texture together as a read-write memory interface is that the fragment processor can read memory as many times as it wants inside a kernel, but it can write data only at the end of the kernel program (this is *stream out*). Thus, memory reads and writes are fundamentally separate on GPUs, so it helps to think about them that way.

Data Types

When programming CPUs, we are used to dealing with multiple data types, such as integers, floats, and Booleans. Current GPUs are more limited in this regard. Although some of the high-level shading languages used by GPUs expose integer and Boolean data types, current GPUs process only real numbers in the form of fixed- or floating point

values. Also, there are multiple floating-point formats supported by current GPUs. For example, NVIDIA GeForce FX and GeForce 6 Series GPUs support both 16-bit (a sign bit, 10 mantissa bits, and 5 exponent bits) and 32-bit (a sign bit, 23 mantissa bits, and 8 exponent bits: identical to the IEEE-754 standard) floating-point formats. All current ATI products, including the Radeon 9800 and X800, support a 24-bit floating-point format, with a sign bit, 16 mantissa bits, and 7 exponent bits. The lack of integer data types on GPUs is a current limitation. This can typically be worked around using floating-point numbers, but, for example, not all 32-bit integers can be represented in 32-bit floating-point format (because there are only 23 bits in the mantissa). One must be careful because floating-point numbers cannot exactly represent the same range of whole numbers that their same-size integer counterparts can represent.

3.3.3 Mapping CPU Computational Concepts to GPUs

Even for expert CPU programmers, getting started in GPU programming can be tricky without some knowledge of graphics programming. In this section, we draw some very simple analogies between traditional CPU computational concepts and their GPU counterparts. We start with the concept of streams and kernels.

Streams: GPU Textures = CPU Arrays

This one is easy. The fundamental array data structures on GPUs are textures and vertex arrays. As we observed before, fragment processors tend to be more useful for GPGPU than vertex processors. Therefore, anywhere we would use an array of data on the CPU, we can use a texture on the GPU.

Kernels: GPU Fragment Programs = CPU "Inner Loops"

The many parallel processors of a GPU are its computational workhorses—they perform the kernel computation on data streams. On the CPU, we would use a loop to iterate over the elements of a stream (stored in an array), processing them sequentially. In the CPU case, the instructions inside the loop are the kernel. On the GPU, we write similar instructions inside a fragment program, which are applied to all elements of the stream. The amount of parallelism in this computation depends on the number of processors on the GPU we use, but also on how well we exploit the instruction-level parallelism enabled by the four-vector structure of GPU arithmetic. Note that vertex programs can also be thought of as kernels operating on a stream of vertices.

Render-to-Texture = Feedback

As mentioned before, most computations are broken into steps. Each step depends on the output of previous steps. In terms of streams, typically a kernel must process an entire stream before the next kernel can proceed, due to dependencies between stream elements. Also, in the case of physically based simulation, each time step of the simulation

depends on the results of the previous time step.

All of this feedback is trivial to implement on the CPU because of its unified memory model, in which memory can be read or written anywhere in a program. Things arent so easy on the GPU, as we discussed before. To achieve feedback, we must use renderto- texture to write the results of a fragment program to memory so they can then be used as input to future programs.

Geometry Rasterization = Computation Invocation

Now we have analogies for data representation, computation, and feedback. To run a program, though, we need to know how to invoke computation. Our kernels are fragment programs, so all we need to know is how to generate streams of fragments. This should be clear from the previous section; to invoke computation, we just draw geometry. The vertex processors will transform the geometry, and the rasterizer will determine which pixels in the output buffer it covers and generate a fragment for each one.

In GPGPU, we are typically processing every element of a rectangular stream of fragments representing a grid. Therefore, the most common invocation in GPGPU programming is a single quadrilateral.

Texture Coordinates = Computational Domain

Each kernel (fragment program) that executes on the GPU takes a number of streams as input and typically generates one stream of output. Newer GPUs that support multiple render targets can generate multiple output streams (currently limited to four RGBA streams). Any computation has an input domain and an output range. In many cases, the domain of a computation on the GPU may have different dimensions than the input streams.

GPUs provide a simple way to deal with this, in the form of texture coordinates. These coordinates are stored at vertices, and the rasterizer linearly interpolates the coordinates at each vertex to generate a set of coordinates for each fragment. The interpolated coordinates are passed as input to the fragment processor. In computer graphics, these coordinates are used as indices for texture fetches. For GPGPU, we can think of them as array indices, and we can use them to control the domain of the computation. The domain and range may be the same size, or the domain can be smaller than the range (data amplification/magnification), or the domain can be larger than the range (data minification). The rasterizer makes it easy to correctly sample the input stream at the correct intervals for each of these cases.

Vertex Coordinates = Computational Range

As discussed before, fragments are generated from input geometry by the rasterizer, and these fragments become output pixels after fragment processing. Because the fragment processors are not directly capable of scatter, the input

vertices and the vertex program determine which pixels are generated. Typically, we specify four vertices of a quad in output pixel coordinates and apply a vertex program that simply passes the vertices through untransformed. Thus, vertex coordinates directly control the output range of the computation.

Reductions

Everything we've discussed up to this point has assumed purely data parallel computation: each element is computed largely independently of the rest of the stream. However, there are times when we need to *reduce* a large vector of values to a smaller vector, or even to a single value. For example, we might need to compute the sum or the maximum of all values in an array. This sort of computation is called a *parallel reduction*.

On GPUs, reductions can be performed by alternately rendering to and reading from a pair of buffers. On each pass, the size of the output (the computational range) is reduced by some fraction. To produce each element of the output, a fragment program reads two or more values and computes a new one using the reduction operator, such as addition or maximum. These passes continue until the output is a single-element buffer, at which point we have our reduced result. In general, this process takes $O(logn)$ passes, where n is the number of elements to reduce. For example, for a 2D reduction, the fragment program might read four elements from four quadrants of the input buffer, such that the output size is halved in both dimensions at each step.

3.3.4 Untraditional GPU concepts

In this section, we describe several additional features of processors on GPUs that are not found in conventional microprocessors and that are relevant to our work.

SIMD instructions with swizzling and smearing

Fragment processors support four-way SIMD instructions. Each register has four components corresponding to four color channels of a pixel (RGBA). Color channels can be permuted, which is called "$swizzling$", and can be replicated, which is called "$smearing$". In the following code, register R1 is used with "$swizzling$", register R0 is used with "$swizzling$" and "$smearing$".

```
R2=R1.abgr * R0.ggab
```

Branch instruction

Early graphics hardware either does not support shaders with branches, or supports branches indirectly through predicated instructions or loop-unrolling. Latest graphics hardware starts to support dynamic branch instructions. However, using dynamic branch instructions can cause expensive performance penalties.

Instruction count

Most graphics hardware has limit on the static number of instructions a shader can contain. With branch instructions, it is possible that dynamic instruction count is vastly higher than static instruction count. Some graphics hardware may have limit on dynamic instructions executed by a shader.

Outputs per shader

A shader is only able to output to designated pixels in the frame buffer. With the introduction of multi-render-targets support in latest graphics hardware, a shader is capable to write to a limited number of auxiliary buffers in addition to the frame buffer.

3.3.5 GeForce 6 Series architecture

Functional Block Diagram for Graphics Operations

Figure 3.4 illustrates the major blocks in the GeForce 6 Series architecture. Here we take a trip through the graphics pipeline, starting with input arriving from the CPU and finishing with pixels being drawn to the frame buffer.

First, commands, textures, and vertex data are received from the host CPU through shared buffers in system memory or local frame-buffer memory. A command stream is written by the CPU, which initializes and modifies state, sends rendering commands, and references the texture and vertex data. Commands are parsed, and a vertex fetch unit is used to read the vertices referenced by the rendering commands. The commands, vertices, and state changes flow downstream, where they are used by subsequent pipeline stages.

The vertex processors (sometimes called “vertex shaders”) allow for a program to be applied to each vertex in the object, performing transformations, skinning, and any other per-vertex operation the user specifies. For the first time, a GPU — the GeForce 6 Series — allows vertex programs to fetch texture data. All operations are done in 32-bit floating-point (fp32) precision per component. The GeForce 6 Series architecture supports scalable vertex-processing horsepower, allowing the same architecture to service multiple price/performance points. For example, high-end models may have six vertex units, while low-end models may have two.

Because vertex processors can perform texture accesses, the vertex engines are connected to the texture cache, which is shared with the fragment processors. In addition, there is a vertex cache that stores vertex data both before and after the vertex processor, reducing fetch and computation requirements. This means that if a vertex index occurs twice in a draw call (for example, in a triangle strip), the entire vertex program doesn’t have to be rerun for the second instance of the vertex — the cached result is used instead.

Vertices are then grouped into primitives, which are points, lines, or triangles. The Cull/Clip/Setup blocks perform

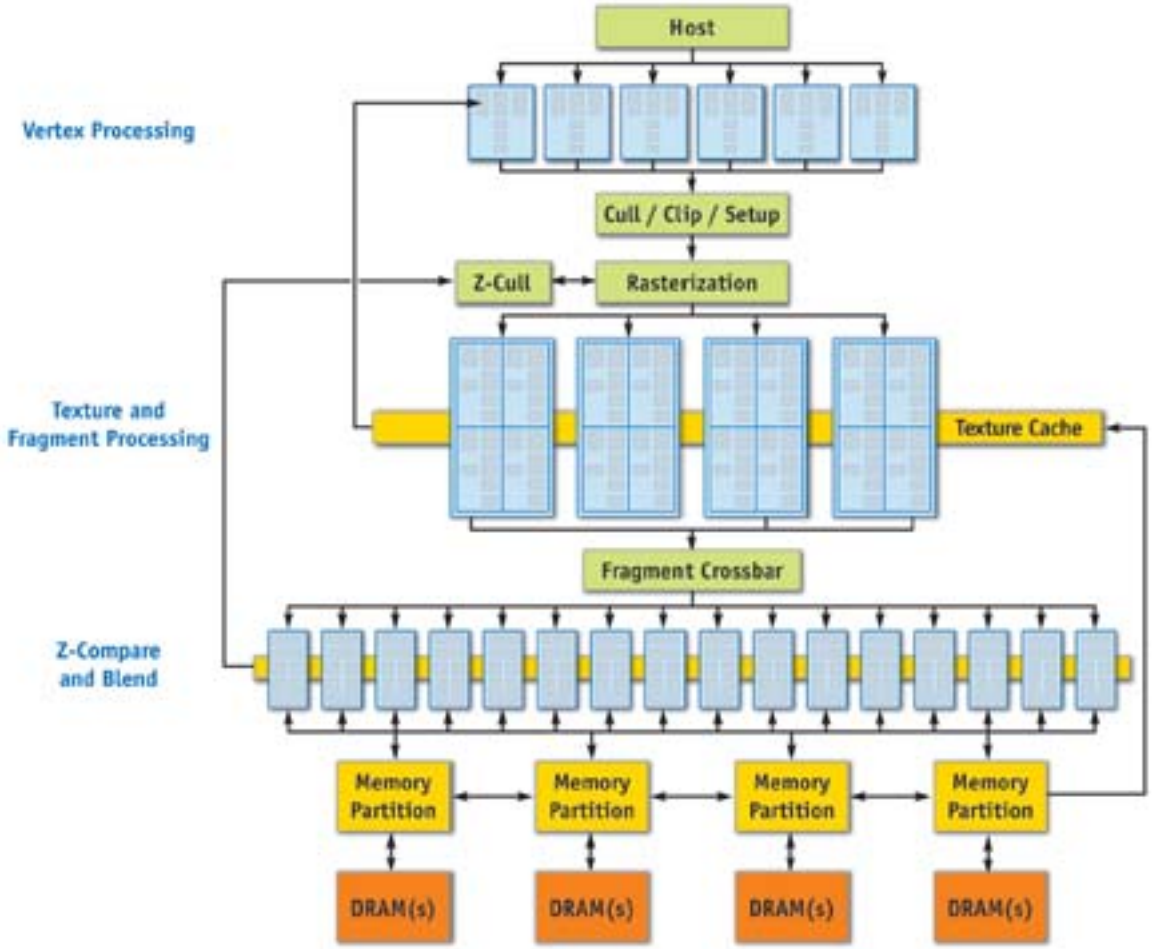


Figure 3.4: A Block Diagram of the GeForce 6 Series Architecture

per-primitive operations, removing primitives that aren't visible at all, clipping primitives that intersect the view frustum, and performing edge and plane equation setup on the data in preparation for rasterization.

The rasterization block calculates which pixels (or samples, if multisampling is enabled) are covered by each primitive, and it uses the z-cull block to quickly discard pixels (or samples) that are occluded by objects with a nearer depth value. Think of a fragment as a "candidate pixel": that is, it will pass through the fragment processor and several tests, and if it gets through all of them, it will end up carrying depth and color information to a pixel on the frame buffer (or render target).

The texture and fragment-processing units operate in concert to apply a shader program to each fragment independently. The GeForce 6 Series architecture supports a variable amount of fragment-processing horsepower by having a varying number of fragment pipelines (or "pixel pipelines"). Similar to the vertex processor, texture data is cached on-chip to reduce bandwidth requirements and improve performance.

The texture and fragment-processing unit operates on squares of four pixels (called quads) at a time, allowing for direct computation of derivatives for calculating texture level of detail. Furthermore, the fragment processor works on groups of hundreds of pixels at a time in single-instruction, multiple-data (SIMD) fashion (with each fragment processor engine working on one fragment concurrently), hiding the latency of texture fetch from the computational performance of the fragment processor.

The fragment processor uses the texture unit to fetch data from memory, optionally filtering the data before returning it to the fragment processor. The texture unit supports many source data formats. Data can be filtered using bilinear, trilinear, or anisotropic filtering. All data is returned to the fragment processor in fp32 or fp16 format. A texture can be viewed as a 2D or 3D array of data that can be read by the texture unit at arbitrary locations and filtered to reconstruct a continuous function. The GeForce 6 Series supports filtering of fp16 textures in hardware. The fragment processor has two fp32 shader units per pipeline, and fragments are routed through both shader units and the branch processor before recirculating through the entire pipeline to execute the next series of instructions. This rerouting happens once for each core clock cycle. Furthermore, the first fp32 shader can be used for perspective correction of texture coordinates when needed, or for general-purpose multiply operations. In general, it is possible to perform eight or more math operations in the pixel shader during each clock cycle, or four math operations if a texture fetch occurs in the first shader unit.

On the final pass through the pixel shader pipeline, the fog unit can be used to blend fog in fixed-point precision with no performance penalty. Fog blending happens often in conventional graphics applications and uses the following function:

```
out = FogColor * fogFraction + SrcColor * (1 - fogFraction)
```

This function can be made fast and small using fixed-precision math, but in general IEEE floating point, it requires

two full multiply-adds to work effectively. Because fixed point is efficient and sufficient for fog, it exists in a separate small unit at the end of the shader. This is a good example of the trade-offs in providing flexible programmable hardware while still offering maximum performance for legacy applications. Fragments leave the fragment-processing unit in the order that they are rasterized and are sent to the z-compare and blend units, which perform depth testing (z comparison and update), stencil operations, alpha blending, and the final color write to the target surface (an off-screen render target or the frame buffer).

The memory system is partitioned into up to four independent memory partitions, each with its own dynamic random-access memories (DRAMs). GPUs use standard DRAM modules rather than custom RAM technologies to take advantage of market economies and thereby reduce cost. Having smaller, independent memory partitions allows the memory subsystem to operate efficiently regardless of whether large or small blocks of data are transferred. All rendered surfaces are stored in the DRAMs, while textures and input data can be stored in the DRAMs or in system memory. The four independent memory partitions give the GPU a wide (256 bits), flexible memory subsystem, allowing for streaming of relatively small (32-byte) memory accesses at near the 35 GB/sec physical limit.

Functional Block Diagram for Non-Graphics Operations

As graphics hardware becomes more and more programmable, applications unrelated to the standard polygon pipeline (as described in the preceding section) are starting to present themselves as candidates for execution on GPUs.

When used for non-graphics applications, GPU can be viewed as two programmable blocks that run serially: the vertex processor and the fragment processor, both with support for fp32 operands and intermediate values. Both use the texture unit as a random-access data read unit and access data at a phenomenal 35 GB/sec (550 MHz DDR memory clock $\times$ 256 bits per clock cycle $\times$ 2 transfers per clock cycle). In addition, both the vertex and the fragment processor are highly computationally capable.

The vertex processor operates on data, passing it directly to the fragment processor, or by using the rasterizer to expand the data into interpolated values. At this point, each triangle (or point) from the vertex processor has become one or more fragments.

Before a fragment reaches the fragment processor, the z-cull unit compares the pixel's depth with the values that already exist in the depth buffer. If the pixel's depth is greater, the pixel will not be visible, and there is no point shading that fragment, so the fragment processor isn't even executed. (This optimization happens only if it's clear that the fragment processor isn't going to modify the fragment's depth.) Thinking in a general-purpose sense, this early culling feature makes it possible to quickly decide to skip work on specific fragments based on a scalar test.

After the fragment processor runs on a potential pixel (still a "fragment" because it has not yet reached the frame buffer), the fragment must pass a number of tests in order to move farther down the pipeline. (There may also be

more than one fragment that comes out of the fragment processor if multiple render targets [MRTs] are being used. Up to four MRTs can be used to write out large amounts of data — up to 16 scalar floating-point values at a time, for example — plus depth.)

First, the scissor test rejects the fragment if it lies outside a specified subrectangle of the frame buffer. Although the popular graphics APIs define scissoring at this location in the pipeline, it is more efficient to perform the scissor test in the rasterizer. Scissoring in x and y actually happens in the rasterizer, before fragment processing, and z scissoring happens during z-cull. This avoids all fragment processor work on scissored (rejected) pixels. Scissoring is rarely useful for general-purpose computation because general-purpose programmers typically process all data fetched from memory in the first place. Next, the fragment's depth is compared with the depth in the frame buffer. If the depth test passes, the fragment moves on in the pipeline. Optionally, the depth value in the frame buffer can be replaced at this stage.

After this, the fragment can optionally test and modify what is known as the stencil buffer, which stores an integer value per pixel. The stencil buffer was originally intended to allow programmers to mask off certain pixels (for example, to restrict drawing to a cockpit's windshield), but it has found other uses as a way to count values by incrementing or decrementing the existing value. This feature is used for stencil shadow volumes, for example.

If the fragment passes the depth and stencil tests, it can then optionally modify the contents of the frame buffer using the blend function. A blend function can be described as

```
out = src * srcOp + dst * dstOp
```

where $source$ is the fragment color flowing down the pipeline; dst is the color value in the frame buffer; and $srcOp$ and $dstOp$ can be specified to be constants, source color components, or destination color components. Full blend functionality is supported for all pixel formats up to fp16$\times$4. However, fp32 frame buffers don't support blending — only updating the buffer is allowed.

Finally, a feature called *occlusion query* makes it possible to quickly determine if any of the fragments that would be rendered in a particular computation would cause results to be written to the frame buffer. (Recall that fragments that do not pass the z-test don't have any effect on the values in the frame buffer.) Traditionally, the occlusion query test is used to allow graphics applications to avoid making draw calls for occluded objects, but it is useful for GPGPU applications as well. For instance, if the depth test is used to determine which outputs need to be updated in a sparse array, updating depth can be used to indicate when a given output has converged and no further work is needed. In this case, occlusion query can be used to tell when all output calculations are done.

Shader 3.0 Programming Model

With Shader Model 3.0, the programming models for vertex and fragment processors are converging: both support fp32 precision, texture lookups, and the same instruction set. Specifically, here are the new features that have been added.

Vertex processor

- **Instruction count** The total instruction count is 512 static instructions and 65,536 dynamic instructions. The static instruction count represents the number of instructions in a program as it is compiled. The dynamic instruction count represents the number of instructions actually executed. In practice, the dynamic count can be much higher than the static count due to looping and subroutine calls.
- **Temporary registers**. Up to 32 four-wide temporary registers can be used in a vertex program.
- **Dynamic flow control**. On the GeForce 6 Series vertex engine, branching and looping have minimal overhead of just two cycles. Also, each vertex can take its own branches without being grouped in the way pixel shader branches are. So as branches diverge, the GeForce 6 Series vertex processor still operates efficiently.
- **Vertex texturing**. Textures can be fetched in a vertex program, although only nearest-neighbor filtering is supported in hardware. More advanced filters can of course be implemented in the vertex program. Up to four unique textures can be accessed in a vertex program, although each texture can be accessed multiple times. Vertex textures generate latency for fetching data, unlike true constant reads. Therefore, the best way to use vertex textures is to do a texture fetch and follow it with arithmetic operations to hide the latency before using the result of the texture fetch.

Each vertex engine is capable of simultaneously performing a four-wide SIMD MAD (multiply-add) instruction and a scalar special function per clock cycle. Special function instructions include:

Fragment Processor

- **Instruction count**. The total instruction count is 65,535 static instructions and 65,535 dynamic instructions. There are limitations on how long the operating system will wait while the shader finishes working, so a long shader program working on a full screen of pixels may time-out. This makes it important to carefully consider the shader length and number of fragments rendered in one draw call. In practice, the number of instructions exposed by the driver tends to be smaller, because the number of instructions can expand as code is translated from Direct3D pixel shaders or OpenGL fragment programs to native hardware instructions.
- **Multiple render targets**. The fragment processor can output to up to four separate color buffers, along with a depth value. All four separate color buffers must be the same format and size. MRTs can be particularly

useful when operating on scalar data, because up to 16 scalar values can be written out in a single pass by the fragment processor. Sample uses of MRTs include particle physics, where positions and velocities are computed simultaneously, and similar GPGPU algorithms. Deferred shading is another technique that computes and stores multiple four-component floating point values simultaneously: it computes all material properties and stores them in separate textures. So, for example, the surface normal and the diffuse and specular material properties could be written to textures, and the textures could all be used in subsequent passes when lighting the scene with multiple lights.

- **Dynamic flow control (branching)**. Shader Model 3.0 supports conditional branching and looping, allowing for more flexible shader programs. Indexing of attributes.With Shader Model 3.0, an index register can be used to select which attributes to process, allowing for loops to perform the same operation on many different inputs.

- **Up to ten full-function attributes**. Shader Model 3.0 supports ten full-function attributes/texture coordinates, instead of Shader Model 2.0's eight full-function attributes plus specular color and diffuse color. All ten Shader Model 3.0 attributes are interpolated at full fp32 precision, whereas Shader Model 2.0's diffuse and specular color were interpolated at only 8-bit integer precision.

- **Centroid sampling**. Shader Model 3.0 allows a per-attribute selection of center sampling, or centroid sampling. Centroid sampling returns a value inside the covered portion of the fragment, instead of at the center, and when used with multisampling, it can remove some artifacts associated with sampling outside the polygon (for example, when calculating diffuse or specular color using texture coordinates, or when using texture atlases).

- **Support for fp32 and fp16 internal precision**. Fragment programs can support full fp32-precision computations and intermediate storage or partial-precision fp16 computations and intermediate storage.

- **3:1 and 2:2 coissue**. Each four-component-wide vector unit is capable of executing two independent instructions in parallel: either one threewide operation on RGB and a separate operation on alpha, or one two-wide operation on red-green and a separate two-wide operation on blue-alpha. This gives the compiler more opportunity to pack scalar computations into vectors, thereby doing more work in a shorter time.

- **Dual issue**. Dual issue is similar to coissue, except that the two independent instructions can be executed on different parts of the shader pipeline. This makes the pipeline easier to schedule and, therefore, more efficient.

Chapter 4

GPU Algorithms for Matrix Multiplication

Due to the rapid improvement in GPU performance and programmability, growing interest has been attracted to performing general purpose computation on GPUs. Matrix multiply as the building block for many other scientific computing tasks, is one of most studied general purpose computing problems on GPUs. In this section, we will describe the GPUs algorithms for matrix multiplication. Larsen et al. [LM01] first presented a single-pass matrix-multiplication algorithm for graphics hardware. Moravánszky [AM03] and Hall et al. [HCH03] introduced two algorithms, which extended Larsen's algorithm to utilize graphics hardware's SIMD instruction with swizzling and smearing by data packing. Fatahalian et al. [FSH04] thoroughly studied the performance efficiency of previously proposed algorithms on a variety of graphics hardware and reached the conclusion that due to the limit of cache-to-processor bandwidth, it is not possible to fully utilize the tremendous computing power of graphics hardware without changing the underlying architecture.

4.1 Naïve GPU Algorithm on GPUs

To begin with, figure 4.1 shows the naïve three nested loop algorithm for multiplying two matrices on CPU (assuming matrix C is initialized to zero).

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        for (k=0; k<K; k++)
            c[i][j] += A[i][k] * B[k][j]
```

Larsen et al. [LM01] first described an algorithm to map the above code onto GPU. It essentially is based on a graphic representation of a parallel algorithm for matrix multiplication. It starts with imagining processors arranged to fill a cube. We can then imagine the first matrix, lying horizontally, distributed in the usual block manner across all the processors on the top face of the cube. This distribution is replicated downwards, so that each horizontal "layer" is a distributed copy of the matrix on the top face. Likewise, we can imagine the second matrix, transposed, on a side face, distributed among the processors on that face, and replicated sideways through the cube. Each processor then performs the small matrix-matrix multiply of its sub-matrices. Finally, these results are summed onto the front face of

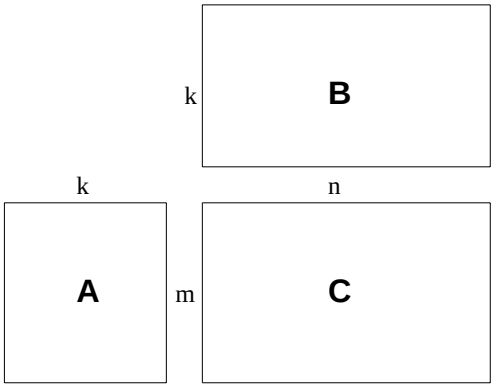


Figure 4.1: Matrix multiplication of $C = A \times B$, where matrix A ($m \times k$) matrix B ($k \times n$) and matrix C ($m \times n$)

the cube, and this now holds our answer.

Let us first assume we have $x \times y \times z$ processors, so each will only get one element of A and one element of B to multiply. This is done by creating two texture maps, one holds the data from A and the other B. Then we set the multi-texturing mode to "modulate" and assign the corresponding element of A and B to each processor. We axis align the cube with our viewing window so that the front face of the cube is all we see, observing that the processors on the front face now occlude all the other processors. We use an orthographic view (no perspective adjustments) so that the little squares line up where they should. Finally, we set the blend mode to "sum", and draw all of these little squares so that their individual results are added into the correct place on the screen, at the front face of the cube. The resulting matrix is retrieved as a memory copy from the graphics card to main memory.

To simplify things, and speed them up, we first combine all the little squares that are in planes parallel to the front face into large squares — there are z of them. One column of texture A and one row of texture B are used for each large square, with the textures spreading perpendicularly to replicate the data. Figure 4.2 illustrates this. In short, the technique is to render z parallel rectangles (each $x \times y$) one behind the other — we can only see the first, and it fills our view. Then texture map both matrices A and B onto each one, and sum them up onto the screen. The answer is then there on the screen. Hence, we say that our technique is to literally "visualize a simple parallel algorithm."

Basically, they propose to store matrix A and matrix B as two textures and to compute the result matrix C in the frame buffer. The shader program fetches one row from texture A and one column from texture B, computes the dot product, and stores the result into the frame buffer.

There are several caveats to this scheme. First, it fails to utilize the SIMD instructions of the fragment processor. Second, no data reuse is exploited. As matrix multiplication performs $O(n^3)$ operations on $O(n^2)$ elements, exploiting data reuse can significantly increase the computation to memory access ratio, thus resulting in better register and cache usage and improved performance. Third, on fragment processors that do not support dynamic branch instructions, the dot-product computation needs to be fully unrolled, which can easily exceed the instruction count limit when the

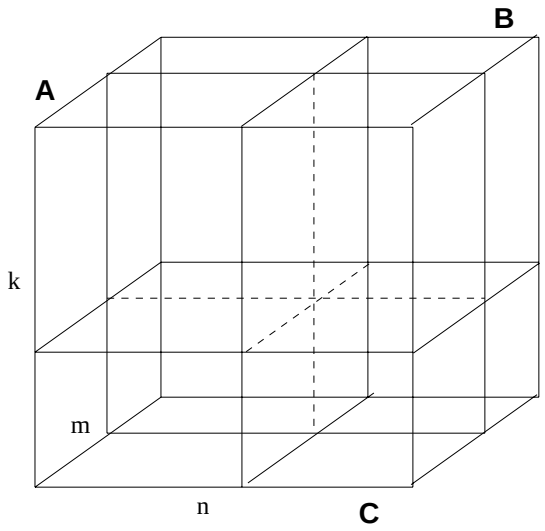


Figure 4.2: Visualization of parallel matrix multiplication algorithm of $C = A \times B$, where matrix A ($m \times k$) matrix B ($k \times n$) and matrix C ($m \times n$)

matrix is large.

4.2 Multiple Pass GPU Algorithms with Data Packing

To address those problems, Moravánszky [AM03] and Hall et al. [HCH03] proposed two multi-pass algorithms with data packing. Multi-pass techniques essentially use strip-mining loop transformations to decompose the inner loop into two nested loops. The shader calculates a partial sum for an element of C. The outer-most loop accumulates partial sums into the correct result for a C element. The following is the multiple-pass algorithm.

```
for (m=0; m<K; m+=b)
   for (i=0; i<M; i++)
       for (j=0; j<N; j++)
            for (k=m; k<m+b; k++)
               c[i][j] += A[i][k] * B[k][j]
```

Data packing is used to pack four elements of a matrix into four color channels of a texel (texture element) so that each memory access operation can load/store four matrix elements instead of just one element in Larsen's algorithm. By packing four elements in one register, the fragment processors are able to execute SIMD instructions.

However, Hall [HCH03] and Moravánszky [AM03] propose different data packing schemes. Hall uses 2×2 scheme, which packs four elements from four sub matrices of the original matrix. Whereas Moravánszky uses 1×4 scheme, which packs four consecutive elements into one texel. The following code is the multi-pass algorithm with

2×2 data packing scheme.

```
for k = 1 ... n/2 step b
    for i = 1 ... n/2
        for j = 1 ... n/2
                R3 = 0
                for m = k ... (k+b - 1)
                    R1 = lookup(i,m,A)
                    R2 = lookup(m,j,B)
                    R3 = R1.rrbb * R2.rgrg + R3
                    R3 = R1.ggaa * R2.baba + R3
                end
                R4 = lookup(i,j,C)
                R0 = R3 + R4
                C(i,j) = R0
        end
    end
end
```

The 2×2 scheme allows each element loaded from memory to be used twice in the shader. Thus, each execution of the shader reads from two rows of matrix A and two columns of matrix B, and produces four elements of matrix C. The 1×4 scheme reads from one row of matrix A and four columns of matrix B to generate four elements for matrix C. The shader performs a few 1×4 vector by 4×4 matrix products. Hence, elements from matrix A are used four times, whereas elements from matrix B are not reused.

4.3 GPU Algorithms with Multi-Render-Targets

Data packing not only enables SIMD instruction but also improves data reuse in the GPU. In this paper, we propose another technique that can further improve data reuse beyond the previous two algorithms. The technique is based on multiple-render-targets (MRT), which is supported in the latest graphics hardware (see section 3.3). MRT allows a shader to write multiple results. One of the results is written to the frame buffer, the others are written to a number of auxiliary buffers. Figure 4.3 illustrates a multi-pass matrix multiplication algorithm with the 1×4 data packing scheme and 2×2 MRT scheme.

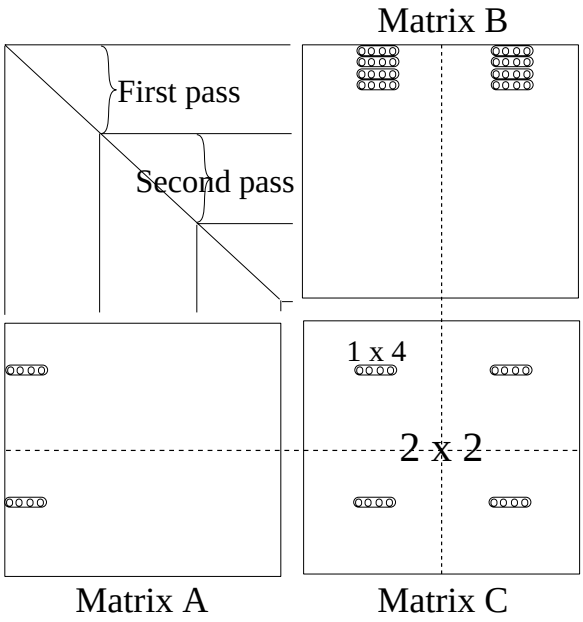


Figure 4.3: Multiply with MRT and data packing

MRT based matrix multiplication algorithms naturally extend data-packing based algorithms. The idea is to divide matrix C into $m \times n$ blocks of sub-matrices. One of them will be assigned to the frame buffer, the other sub-matrices are distributed to auxiliary buffers. $a \times b$ data-packing based algorithms effectively performs strip-mining loop transformation on the i and j loops by factors a and b. The $m \times n$ MRT based matrix multiplication further strip-mines the resulting i and j loops by m and n. With MRT, elements loaded from matrix A can be reused n times further after data packing, elements loaded from matrix B can be reused m times further after data packing.

Chapter 5

Automatic Tuning Matrix Multiplication on Graphics Hardware

In this chapter, we will describe in details the design and implementation of our automatic tuning system for matrix multiply on graphics hardware. We also include the performance results of the built system.

5.1 Automatic Tuning System

Typically, automatic tuning approach involves three components as shown in figure 5.1. A code generator inputs the values of tuning parameters and outputs the program version that is specified by these parameters. An evaluator empirically evaluates the performance metrics of the generated code and feeds back the metrics to search engine. A search engine searches over the implementation space by controlling the tuning parameter values fed into the code generator according to some search strategy. We will elaborate our tuning system with regarding to figure 5.1.

5.1.1 Code Generator

In designing our code generator, we adopt similar strategy as ATLAS [WPD01]. We focus on tuning the kernel routine for multiplying 1024×1024 matrices. Input matrices are first divided into blocks of 1024×1024 sub matrices. Then the matrix multiplication is performed in terms of multiplying the block matrices with the tuned kernel routine. Matrices of size not multiples of 1024 will result in clean-up code, which can be executed either by the CPU or by the GPU

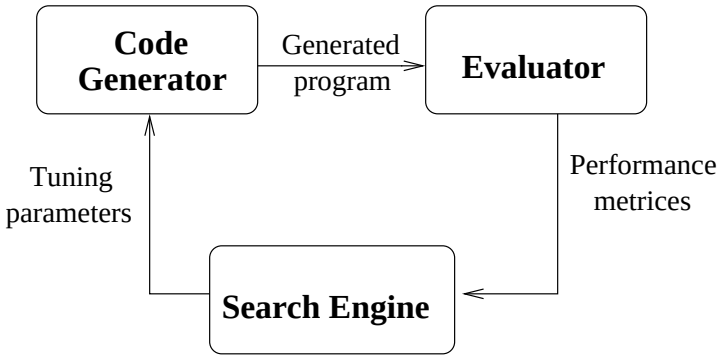


Figure 5.1: Components of automatic tuning

with code generated similarly. We choose the particular value of 1024 because it yields the best performance.

The other issue to address in designing our code generator is what programming language generated programs should be coded into. There are three levels of programming language that can be used to program graphics hardware: assembly level shading languages such as the "ARB_fragment_program" extension to OpenGL, high-level shading languages such as the Cg language [nVi] from nVidia, and high-level general purpose languages such as BrookGPU [BFH+04].

Assembly level shading languages require programmers to explicitly manipulate registers and arithmetic instructions. Assembly programs are encoded in string variables in C/C++ programs that use OpenGL or Direct3D commands to maintain graphics pipeline states and load/execute the assembly program.

High level shading languages like Cg and HLSL allow shaders to be written in C-like language. Similar to assembly program, the C-like shading programs are encoded in string variables in C/C++ programs that use OpenGL or Direct3D and the high level shading language's runtime library to explicitly control the graphics hardware pipeline and to load/compile/execute the high-level shading code.

High-level general purpose languages go one step further by hiding the graphics hardware characteristics. Programs written in the BrookGPU language are first source-to-source translated into C++ programs containing fragment programs coded in Cg that appear as string variables and wrapper C++ code for setting up and maintaining graphics pipeline states. From this point, on top of the BrookGPU's runtime library, the generated C++ program will execute just as a normal graphics program written in C++ with fragment programs encoded in Cg.

We decided to generate programs in the highest level language, specifically the BrookGPU language, mainly for two reasons. First, generated code should be portable to various architectures, even future architectures that are not yet defined. Generating high level program will permit fundamental changes in hardware and graphics API as long as the compiler and runtime library for the high level language keep up with those changes. Whereas, code generated in assembly language or Cg language is tied to particular generation of hardware and may need to be modified to utilize new features of the hardware or graphics API. Second, implementing the code generator is a very tedious and error-prone process. The generator is easier to debug when its output is high-level code. The downside of this decision is that the code compiled from BrookGPU is less efficient than manually generated code. One can hope that as high level languages for GPUs and their associated compilers and run-time libraries mature, the performance penalty for the use of high level languages will shrink or disappear, as it happened with conventional processors.

Our code generator is implemented in the Python script language to generate BrookGPU programs according to input tuning parameter values.

5.1.2 Tuning Strategies and Parameters

In this subsection, we describe the tuning strategies and their associated tuning parameters for our system.

Tuning Multi-Render-Targets

Today's most advanced GPU offers up to three auxiliary buffers in addition to the frame buffer known as multiple render targets (MRT). The MRT strategy can help improve data reuse and therefore reduce the number of memory accesses and improve performance. However, MRT necessitates the copying of intermediate results stored in auxiliary buffers into texture memory for further passes. Furthermore, MRT requires more temporary registers by the shader, which reduces the performance of the fragment processors. Hence the optimal scheme to decompose matrices to use MRT needs to be tuned to the target platform.

[**mrt_w**, **mrt_h**]: The matrix C is divided into $mrt\_w \times mrt\_h$ sub-matrix blocks to utilize MRT. The valid values for these two parameters are limited by the number of auxiliary buffers supported in hardware. Since latest hardware supports up to 3 additional buffers, the possible values of these two parameters range over 8 cases, which have the product of mrt_w and mrt_h less or equal to 4.

Tuning Data Packing

This is the strategy of utilizing SIMD instructions with data packing. As introduced in section 4.2, the two data-packing schemes 1×4 and 2×2 have different advantages and disadvantages. Our automatic tuning system relies on actual execution to decide which one is better on target platform.

[**mc_w**, **mc_h**]: Tuning parameters "mc_w" and "mc_h" decide how to pack consecutive elements into one texel. $mc\_w \times mc\_h$ block of elements are packed into one texel. As there are only four available channels (RGBA) for each texel, the product of mc_w and mc_h must be less or equal to 4. Hence, there are totally 8 cases.

Tuning Number of Passes

It would be nice to have a long shader that calculates the resulting matrix C in one pass, which can eliminate the expensive texture-copy or render-to-texture operation for intermediate results. However, due to fragment processor's limit on instruction count and temporary registers, a shader can not be too long. Even within the valid range of instruction count limit, longer shader may perform worse than shorter shader. As a result, the number of k-loop iterations to be executed in a shader needs to be tuned to the target platform.

[**np**] Tuning parameter "np" determines how many iterations in k-dimension loop are executed by the fragment shader. We observed from experiments that np larger than 256 is either not supported by hardware or has already started to suffer from performance penalty. Hence, in our tuning system, we limit the range of np from 1 to 256.

Tuning for Branch Instruction

Latest graphics hardware adds support for dynamic branch instruction. This allows implementing a loop-based shader without having to fully unroll it as is the case for earlier generation of graphics hardware, which does not have dynamic branch instructions. Using loop-based shader could help reduce static instruction count. However, as branch instructions come with an expensive performance penalty, whether to use branching or loop unrolling needs to be tuned on actual hardware.

[**unroll**] Tuning parameter "$unroll$" decides whether or not to use branch instruction to implement a loop-based shader. The valid values of $unroll$ are either 0 or 1. If $unroll$ equals 1, the inner loop of the generated code will be fully unrolled.

Other Tuning Parameters

[**compiler**] BrookGPU uses either "cgc" or "fxc", which are compilers from nVidia's Cg Toolkit and Microsoft's DirectX9 respectively, to compile Cg program into assembly fragment program. Since these two compilers might perform different optimizations, the generated code can execute differently. We use a tuning parameter "$compiler$" to determine which one to use to compile shader. The valid values of $compiler$ are either "cgc" or "fxc".

[**profile**] This tuning parameter comes from different shader models to interface with graphics hardware. Currently, there are two popular graphics API's, namely Direct3D and OpenGL. They provide somewhat equivalent functionalities through different programming API. BrookGPU is able to use either of them as the back end API to interact with GPU. For both Direct3D and OpenGL, there are several shader profiles. Specifically, Direct3D has four shader profiles, "$ps20$", "$ps2a$", "$ps2b$" and "$ps30$". OpenGL has three profiles "arb", "$fp30$", "$fp40$". The profiles provide different capabilities to shader programs. For example, "$fp40$" and "$ps30$" support dynamic branch instruction. Also, different profiles have different limits on instructions count and number of temporary registers. We use a tuning parameter "$profile$" to choose among back-ends and shader models. The valid values of $profile$ are "$ps20$", "$ps2a$", "$ps2b$", "$ps30$", "arb", "$fp30$", "$fp40$".

5.1.3 Performance Evaluator

The performance evaluator executes the generated code and returns MFLOPS (million floating point operations per second) as the performance metric to evaluate its quality. Zero MFLOPS is returned for invalid programs, i.e. programs that fail to execute or programs that exceed the instruction count limit.

5.1.4 Search Engine

The search engine is responsible for searching over the implementation space to find the version with the best performance. As optimization in multi-dimensional discrete space is generally an NP-hard problem, there is no general algorithm that can solve this discrete optimization without exhaustive search. In our case, exhaustive search over all possible versions would require $8 \times 8 \times 256 \times 2 \times 2 \times 7 = 458752$ evaluations. If each evaluation takes 10 seconds, the whole search would take 53 days, which may not be acceptable.

We implement an ad-hoc search algorithm specifically for our tuning problem. We employ two techniques to limit our search to around four hours without sacrificing too much performance. The first technique is to employ some problem specific heuristics to prune the search space of tuning parameters. The second technique is to search tuning parameters in some predetermined order to effectively decompose the high-dimensional search space into multiple lower-dimensional spaces.

Space Pruning

According to the symmetric property of mc_w and mc_h parameters, we impose an additional constraint that $mc\_w \leq mc\_h$. Since the matrix size is 1024×1024, we also limit mc_w and mc_h to powers of two. Now we have only four possible cases $(mc\_w, mc\_h) \in \{(1,1),(1,2),(1,4),(2,2)\}$. Similarly mrt_w and mrt_h can be limited to four cases: $(mrt\_w, mrt\_h) \in \{(1,1),(1,2),(1,4),(2,2)\}$.

Parameter np decides the number of iterations in k-loop to be executed in the shader. Intuitively, as np increases, the performance will first improve due to fewer number of passes. When np exceeds some optimum value, the instruction count issue and excessive use of temporary registers start to outweigh the benefits of fewer passes.

The search problem essentially boils down to finding the maximum value of a unimodal function. Theoretically, the best algorithm has complexity of $O(log(n))$. However, for our particular problem, since the np value range is rather small and we believe the optimum np value should be near power of two values, we designed algorithm 1 for finding optimum np value. The idea of algorithm 1 is to exponentially increase stride and evaluate the performance at corresponding np until either the end of interval is reached or the performance is less than the minimum of the previous two evaluated performances. The procedure is then recursively invoked on both sides of the best np found in the exponential search until the length of interval is less or equal to a predetermined threshold. Its theoretical worst case complexity complies with the following recursion.

$$f(n) = f(\frac{n}{2}) + f(\frac{n}{4}) + log(n)$$

Solving this recursion gives the algorithm's worst case complexity of $O((log_2 n)^{\frac{\sqrt{5}+1}{2}})$. In our tuning system, it is often

the case that the loop at step 8 of algorithm 1 is exited prematurely because performance goes below the previously evaluated two values. Therefore, algorithm 1 practically has better performance than generic $O(log(n))$ algorithms for our problem.

Algorithm 1 Finding Optimum np

Input: *start* – starting value of np in the interval

length – length of the interval

direction – left(-1) or right(1)

Output: update global variable storing the best np

procedure *find_np*(*start*, *length*, *direction*)

1: **if** ($length \leq threshold$) **return;**

2: Initialize p, $last\_two$, max_mflops, $best\_np$

3: **repeat**

4: Evaluate *mflops* at $np = start + direction * p$

5: **if** (*mflops* $> max\_flops$)

6: update *max_mflops*, *best_np*

7: exponentially increase stride p

8: **until** out of range or performance $\leq$ *min*(last_two).

9: *find_np*(*best_np*, *left_size*, *left*)

10: *find_np*(*best_np*, *right_size*, *right*)

11: **return**;

Search in Phases

In addition to space pruning, search in phases can further reduce the search space by decomposing the high dimensional space into several lower dimensional spaces. The assumption is that the optimal values of some tuning parameters are independent of each other, so that we can search the best values for some tuning parameters while fixing the others. Formal proof of independence relationship between parameters of multi-variate function is difficult. In our case, from experimental results, we speculate that the np parameter is independent of mc_* and mrt_* parameters to some extent, therefore we decouple the nested search for np and mrt_*, mc_* into a sequential search. Algorithm 2 describes the search order of tuning parameters we use in our tuning system. The search for np parameter is further divided into two stages. In step 4, only power of two values are searched. In step 8, after mc_* and mrt_* are determined, algorithm 1

is applied to pin down the best np.

Algorithm 2 Search Order of Tuning Parameters

```
1: For each ***compiler*** value
2:     For each ***profile*** value
3:         For each ***unroll*** value
4:             Search ***np*** in power of two values
5:             For each ***mc_**** value
6:                 For each ***mrt_**** value
7:                     Evaluate Performance
8:             Recursively search ***np*** in both sides of
               best ***np*** found in step 4.
```

After applying the above two techniques, the typical evaluation running time reduces to around 4 hours.

5.2 Performance Evaluation

We run the automatic tuning system on four graphics cards. Their configurations are given in table 5.1. The host CPU and operating system are 2.6Ghz Pentium 4 and Windows XP.

	G6800U	G6800G	QF3400	G5800U
Model Name	GeForce 6800 Ultra	GeForce 6800 GT	Quadro FX 3400	GeForce FX 5800 Ultra
Pixel Processor	16	16	16	4
Core Frequency	400 MHz	350 MHz	350 MHz	500 MHz
Mem Frequency	1100 MHz	1000 MHz	900 MHz	1000 MHz
Mem Width	256 bit	256 bit	256 bit	128 bit
Bandwidth	35.2GB/s	32.0GB/s	28.8GB/s	16GB/s
Driver	6693	7568	6176	6693
GPU	NV40	NV40	NV45GL	NV30
DirectX	9.0c	9.0c	9.0c	9.0c
OpenGL	1.5.2	2.0.0	1.5.1	1.5.2

Table 5.1: Four GPU platforms

We conducted experiments only on nVidia cards. We did not test on ATI cards mainly because no ATI cards truly support 32-bit floating point. The most advanced ATI cards, ATI Radeon X800 XT only supports 24-bit floating point data operations in pixel processors.

We benchmarked the performance of multiplying two matrices of size 1024×1024, whose elements are randomly generated. Timing operations performed by a GPU is difficult because the GPU and the CPU work asynchronously. We work around this problem by measuring the time from the start of the multiplication until one element of the result matrix is read from the GPU to the CPU. This potentially could involve an overhead of moving large matrices between GPU and CPU and the serial overhead of setting up graphics pipeline states. In order to reduce the impact of this overhead, we force the GPU to perform the same matrix multiplication operation ten times and use the average as the execution time. Our experiments show that the overhead is typically below 10% of measured performance, and the error range of measured performance is below 3%.

5.2.1 Manually Tuned Implementation

Fatahalian et al. [FSH04] thoroughly studied the performance efficiency of matrix multiplication algorithms on a variety of graphics hardware and presented two hand-tuned implementations as the most efficient implementations. They included these two implementations in GPUBench [Sta], which is a benchmark suite designed to analyze performance of programmable graphics processor for GPGPU. To test the effectiveness of our automatic tuning approach, we compare the performance of our automatic tuned version with these two expert hand-tuned implementations.

Table 5.2 summarizes the high level structure of these two implementations in terms of our tuning parameters described in section 5.1.2. We use the same names "NV_Single" and "NV_Multi" as in [FSH04] to refer to them.

	mrt_*	mc_*	np	unroll	profile	compiler
NV_Single	1×1	2×2	128	1	fp30	NA
NV_Multi	1×1	1×4	6	1	arb	NA

Table 5.2: Param. for manual implementations

It is important to note that the two implementations are implemented in C++ and OpenGL API with fragment program in carefully crafted assembly code. Whereas, our automatic tuning system generates high level BrookGPU code. As explained in section 5.1.1 and confirmed by our experiment data, the difference in implementation level has significant impact on the performance.

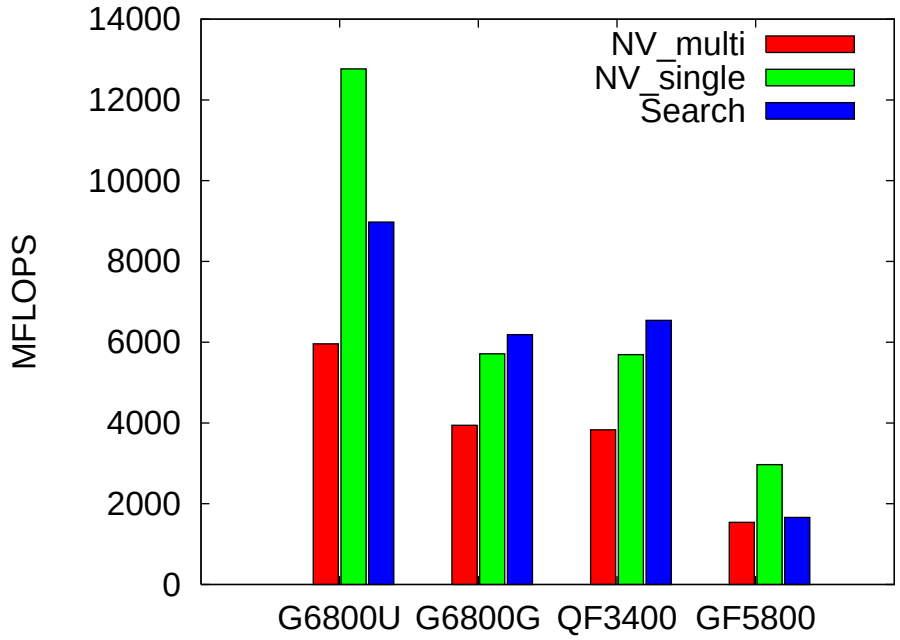


Figure 5.2: Performance on four platforms

5.2.2 Experiment Results

Now we present the experiment results. We first compare the performance of our automatically generated matrix multiplication with the manually tuned versions on four platforms. Then we study the sensitivities of the tuning parameters to overall performance. In all figures shown in the subsection, Y axis represents MFLOPS.

Automatic Vs. Manual

Figure 5.2 shows the performances of the two hand-tuned implementations and the automatically tuned version, which is denoted as "Search", on the four platforms. As we can see, "NV_multi" consistently performs the worst among the three implementations. Between "NV_Single" and "Search", on G6800U and G5800U, "Search" achieves 70%, 56% of the performance of "NV_Single". On G6800G and QF3400, "Search" achieves 8% and 15% speedup over "NV_Single" respectively.

This result might look surprising, because both of the hand-tuned implementations are within the search range of automatic tuning. The reason for the lower performance of "Search" is the overhead associated with using the high level BrookGPU language. We found some inefficiencies in the BrookGPU's runtime system and "cgc"/"fxc" compilers. For example, instead of using "render-to-texture" technique, BrookGPU's OpenGL backend uses the expensive copy operation to move intermediate results from the frame buffer to texture memory. Also in dealing with array-indexing operation, BrookGPU seems to generate auxiliary instructions to map index values to texture

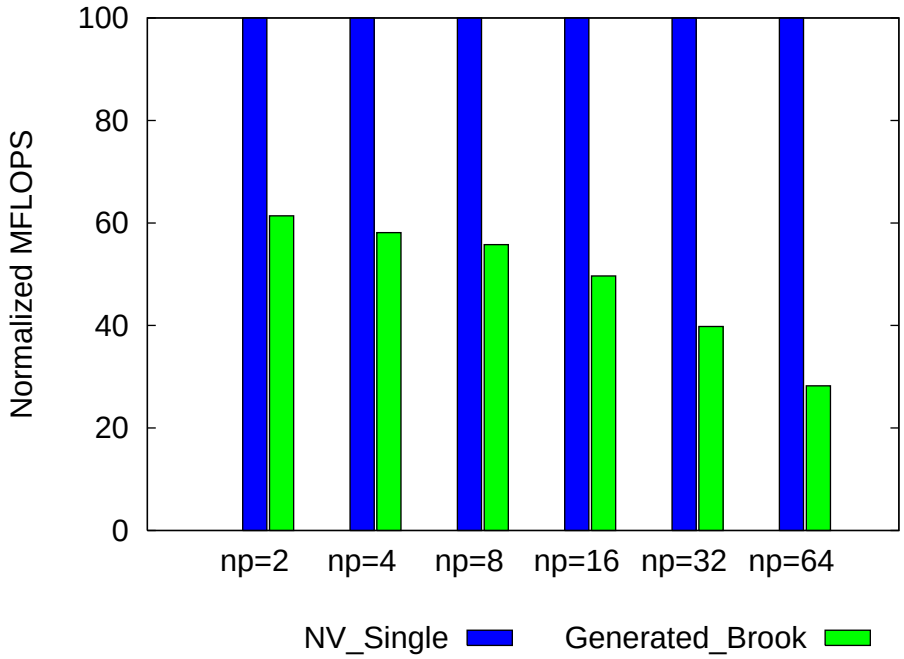


Figure 5.3: Performance penalty associated w/ runtime library and compiler optimization

coordinates. The addition of extra instructions compared to carefully crafted assembly code would hurt performance. We also suspect that "cgc" and "fxc" compilers' register allocation strategy is not optimum for some cases. For instance, when a loop is unrolled, occasionally the compiler fails to reuse registers across unrolled iterations of the loop, which greatly increases the pressure on registers and limits the ability of unrolling loop to improve performance.

In order to roughly measure the performance overhead of using high level BrookGPU language, we compare the performance of "NV_Single" with the performance of its counterpart implementation in BrookGPU. We force the code generator to generate implementations in BrookGPU with the same mrt_*, mc_*, $unroll$, $profile$ values as "NV_Single" 's corresponding values in table 5.2. For $compiler$ parameter, we use the $compiler$ value with the best performance. We vary the np tuning parameter from 2 to 64 in power of two values. We choose this range because "NV_Single" does not support $np = 1$ case and larger np in power of two values will exceed the instruction limit. Figure 5.3 , which is based on data collected on G6800U platform, shows the relative performance of the "NV_Single" and its counterpart implementation in BrookGPU. As can be observed, the generated BrookGPU version never reaches more than 60% of the performance of "NV_Single". As np increases, the relative overhead also increases. We don't fully understand the reason.

If taking into account the overhead of using the high level BrookGPU language, the performance achieved in figure 5.2 is satisfactory. On two platforms, the automatic tuned version can even outperform the hand-tuned version. This is mainly because "NV_Single" was specially tuned for graphics hardware similar to "GeForce 6800 Ultra"

graphics card. When changing to other platforms, the performance of "NV_Single" is far from optimum. This testifies the benefit of automatic tuning system to adapt to changing underlying architecture.

Parameter Sensitivity

In this subsection, we present the sensitivities of the tuning parameters described in section 5.1.2 to performance.

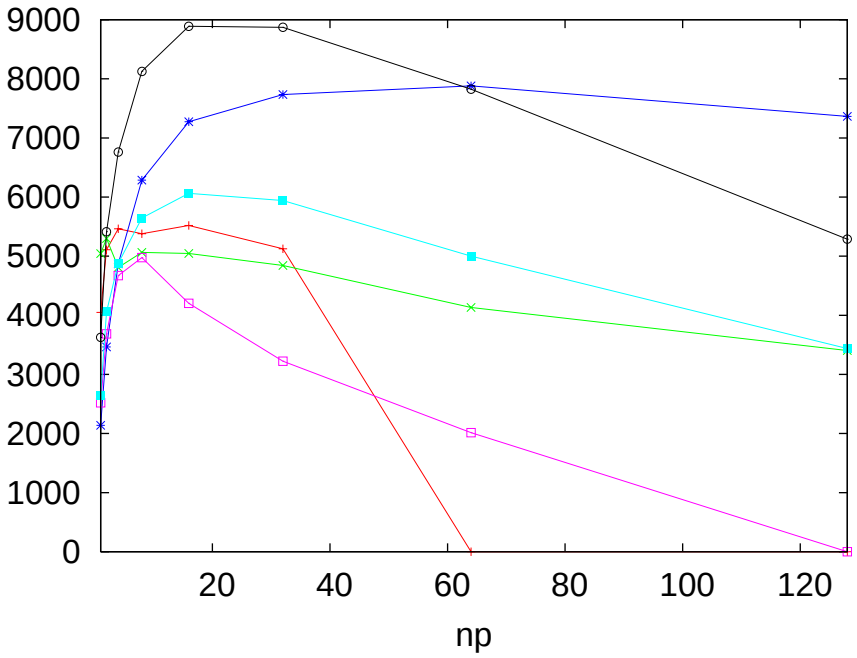
Figure 5.4(a) shows the performance curves over power of two np values. These curves are searched in step 4 of algorithm 2. Different curves correspond to fixing the other parameters to different values. All curves have a single maximum. Figure 5.4(b) shows the performance curves over np ranging from 1 to 128. These curves are searched in step 8 of algorithm 2. As can be observed that there are some performance drops off the original curves in 5.4(a) at some particular np values. The performance curves recover from those drops gradually to the original curves. We don't understand the underlying reason for these performance drops, however, since the dropping points are not in power of two values, in most cases algorithm 1 can still find the global optimum as if the curve is a unimodal function.

Figure 5.5, which is based on data collected on G6800U platform, shows the sensitivities and interaction of mrt_* and mc_* parameters. As described in section 5.1.4, both mrt_* and mc_* range over $\{(1,1),(1,2),(1,4),(2,2)\}$. For each combination of mrt_* and mc_*, we tested five np values at $\{2,4,8,16,32\}$. On G6800U platform, $mc = 2\times2$ achieves 2X to 2.5X speedup over $mc = 1\times1$. $mrt = 2\times2$ can further achieve 10% speedup over $mrt = 1\times1$. The best mrt_* and mc_* values are platform dependent.

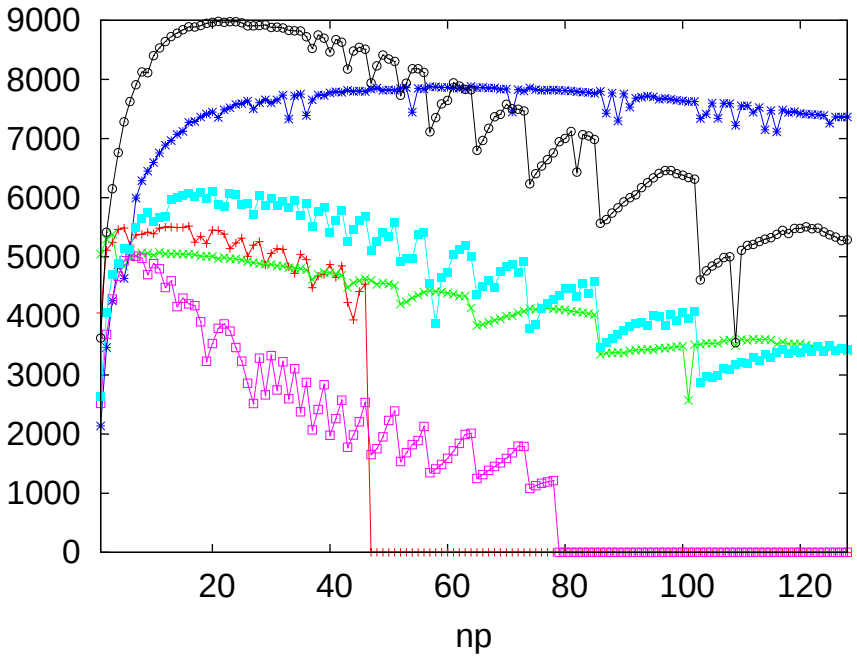
For the "$unroll$" parameter, our experiments show that $unroll = 0$ is almost always better than $unroll = 1$. It is because for profiles that do not support branch instruction, the fxc and cgc compilers automatically unrolls the loop even if $unroll$ is set to zero. For profiles supporting branch instruction, the compilers determine whether to unroll the loop based on the length of the shader even if $unroll$ is set to zero. Hence, generating high-level code with explicit loop-unrolling does not benefit performance in both cases.

For the $profile$ parameter, we find in all of the four platforms we tested that profiles supporting more capabilities generally perform better than profiles supporting fewer capabilities. For example, for "DirectX" back end, performance increases in the order of "ps20", "ps2b", "ps2a", "ps30". For "OpenGL" back end, performance increases in the order of "arb", "fp30" and "fp40".

For the $compiler$ parameter, we find both fxc and cgc generate codes of equivalent quality on all platforms.



(a) **Power of two values.**



(b) **All values**

Figure 5.4: Sensitivity of np parameter.

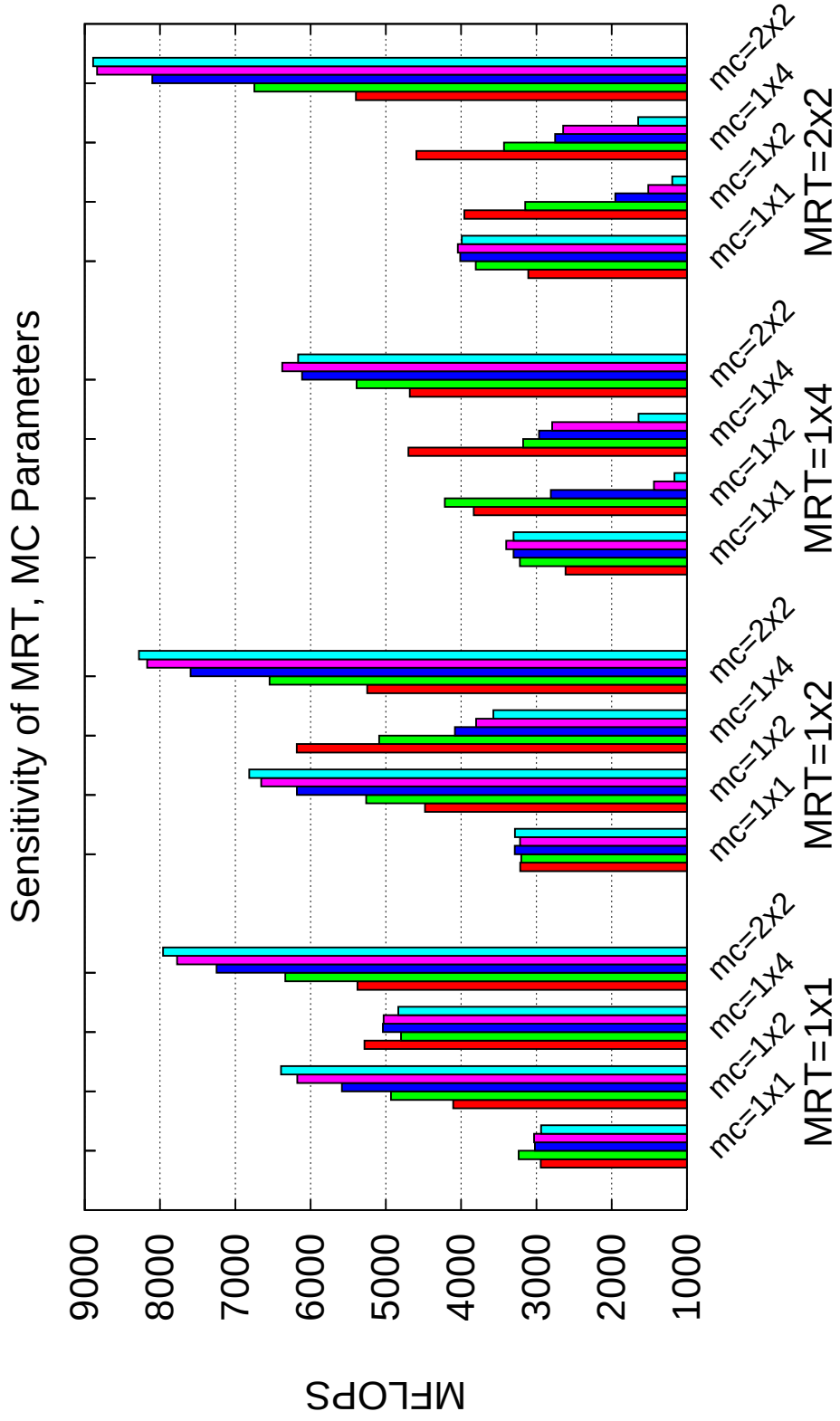


Figure 5.5: Sensitivity of MRT and MC

Chapter 6

Frequent Pattern Mining and Support Vector Machine

6.1 Frequent Pattern Mining

Frequent pattern mining, which is also known as frequent itemset mining, was first formalized by Agrawal et al. [AIS93] for association rule mining. It can be described as follows: Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n items, and let $\mathcal{D} = \{T_1, T_2, \ldots, T_m\}$ be a set of m transactions, where each transaction T_i is a subset of I. Any subset of I is called an *itemset*. An itemset of length k is called a *k-itemset*. For an itemset X, the projected transaction database for X, $\mathcal{T}(X) = \{t | X \subseteq t, t \in \mathcal{D}\}$, is the set of transactions in $\mathcal{D}$ including X. The *support* ($|\mathcal{T}(X)|$) for an itemset X is the number of transactions including the itemset. An itemset is *frequent* if its support is beyond a user specified threshold, called *minimum support* or *support threshold*. The task of frequent pattern mining is to discover all frequent item sets.

The problem of identifying all frequent item sets is computationally intensive. Given a transaction database with n items, there are potentially 2^n item sets, which form a *lattice of subsets* over I as shown in figure 6.1. However, usually only a small fraction of the whole subset lattice space is frequent. Several algorithms have been proposed in literature [AIS93, AS94, GZ01, Goe02, BCG01, SON95, ZPOL97, HPY00, LPWH02, PHL$^+$01, ZG03] to mine frequent patterns efficiently. Almost all of the algorithms take advantage of the *downward closure* property to prune the subset lattice – the property that all subsets of a frequent itemset must be frequent themselves. According to the order they search in the subset lattice, these algorithms can be divided into *breadth-first-search* and *depth-first-search* types. *Apriori* [AS94] and *FP-Growth* [HPY00] are two representatives of each. In this paper, we consider only *depth-first-search* algorithms as they were found more efficient than *breadth-first-search* algorithms by prior research.

During the FIMI workshops [GZ03a, JGZ04] more than 20 different implementations of the several published algorithms were evaluated and compared on several data sets, most of which were real-world data sets. The FIMI workshop showed that there was no single algorithm that outperformed all the others for all the inputs. Out of all the tested implementations, two implementations of the FP_Growth [GZ03b] algorithm and the LCM [UKA04] algorithm got awards for best average performance at the FIMI workshops of 2003 and 2004, respectively. For the work in this paper we take these two algorithms and Eclat. Eclat has been added because it seems to perform well on inputs that are bad for LCM and FP_Growth, so that it complements these two algorithms.

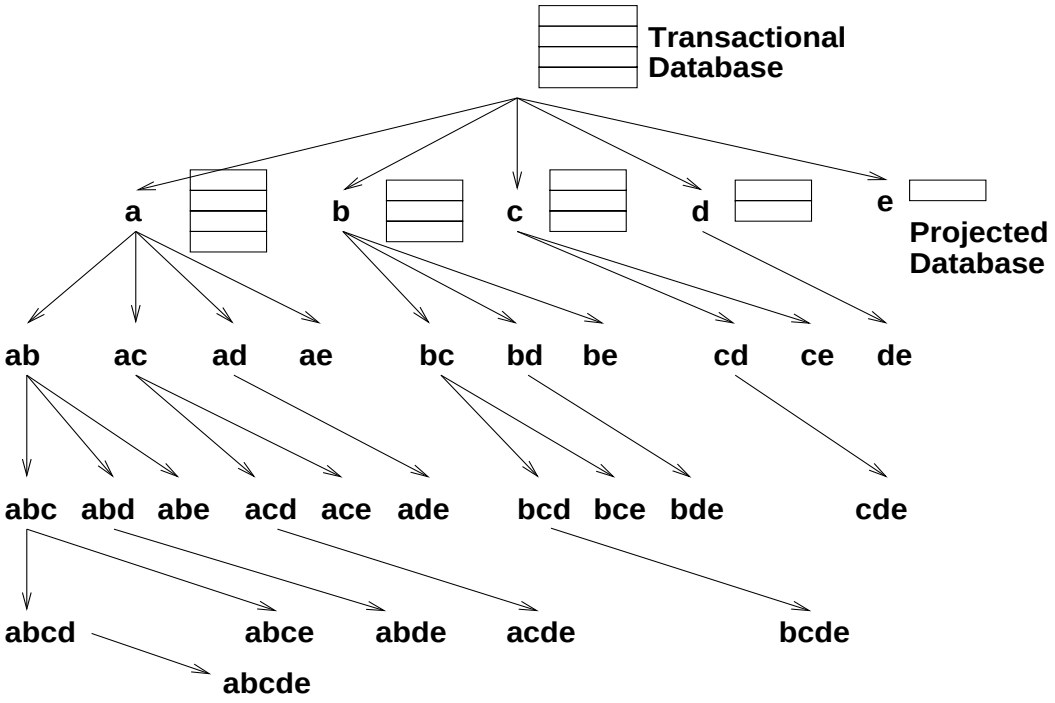


Figure 6.1: The search space of the subset lattice

Next we describe the three algorithms we focus on, paying special attention to the input characteristics that determine the performance of these algorithms compared to the others.

6.2 Algorithm Description

6.2.1 FP_Growth

FP_Growth was first proposed by Han et al. [HPY00]. This algorithm uses an augmented prefix tree, called *FP_Tree* (Frequent Pattern Tree) to represent the database in a compact way. The FP_Tree is very efficient at compressing datasets when many transactions share common prefixes, as shown in Figure 6.2. The FP_Growth algorithm proceeds by performing two data scans over the original database; the first one counts the number of occurrences of each item, and the second one builds the initial FP_Tree. Then, recursively builds smaller FP_Trees that represent projected databases, consisting of all transactions containing a particular itemset. Experimental results [GZ03b, GBP+05] show that FP_Growth spends most of the time building and traversing the FP_Trees. To reduce this overhead, the authors of [GZ03b] implemented a variant of the original FP_Growth algorithm where a 2D array that counts the frequencies of all pairs of frequent items is constructed at the same time as each FP_Tree. This optimization results in significant performance savings when the dataset is sparse. The implementation in [GZ03b] only uses the 2D array optimization when the dataset appears to be sparse. Another potential problem of the FP_Growth algorithm is that each node in

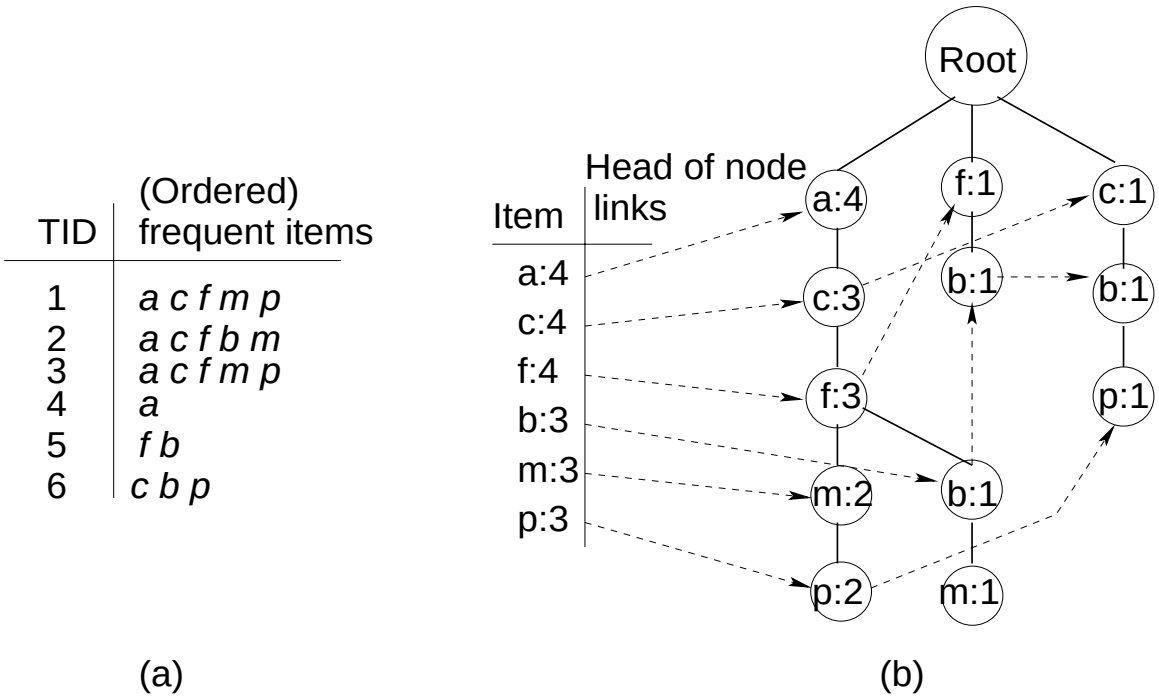


Figure 6.2: FP_Tree example.(a) Dataset with only the (ordered) frequent items and (b) corresponding FP_Tree

the FP_Tree requires four pointers, one to the parent, one to the child, one to the sibling to the right, and another to the next node with the same item. These pointers may add significant overhead to the traversal of the FP_Tree and increase the memory consumption.

For the results reported in this paper we use the implementation from [GZ03b], which outperformed many other popular frequent mining algorithms and received the best implementation award from the FIMI 2003 workshop [GZ03a].

6.2.2 LCM

LCM [UAUA03, UKA04] (*Linear time Closed itemset Miner*) algorithm creates projected databases only for frequent closed item sets. An itemset P is a *closed itemset* if it is not properly contained in an itemset Q with the same support as P. LCM generates the remaining frequent item sets by enumeration. This technique is called *hyper-cube decomposition*: Suppose that P is a frequent itemset, $P \cap Q = \emptyset$ and any transaction that contains P also contains Q. Then $P \cup Q'$ is a frequent itemset, for any $Q' \subseteq Q$. Furthermore, if P is an itemset, then there is an itemset $Q \subseteq P$ so that Q is a closed itemset with the same support as P. Thus, the item sets can be partitioned so that each component consists of all the item sets $\{P \cup Q' : Q' \subseteq Q\}$, where P is closed, and $P \cap Q = \emptyset$. This *hyper-cube decomposition* can significantly speedup the mining process when projected databases have many co-occurring items.

Algorithm 3 FP-Growth algorithm

FP-GROWTH ($\mathcal{T}$: FPTree, *suffix*: itemset)

If tree has only one path

Output $2^{path} \cup$ *suffix*

Else

Foreach frequent one item e in the header table

Output the $\{e\} \cup$ *suffix* as frequent

FIRSTSCAN: Use the header list for e to find all frequent items in conditional pattern base C for e

SECONDSCAN: If we find at least one frequent item in the conditional pattern base, use the header list for e, and $\mathcal{T}$ to generate conditional prefix tree $\mathcal{N}$

If $\mathcal{N} \neq \{\}$ then

FP-GROWTH($\mathcal{N}$, $\{e\} \cup$*suffix*)

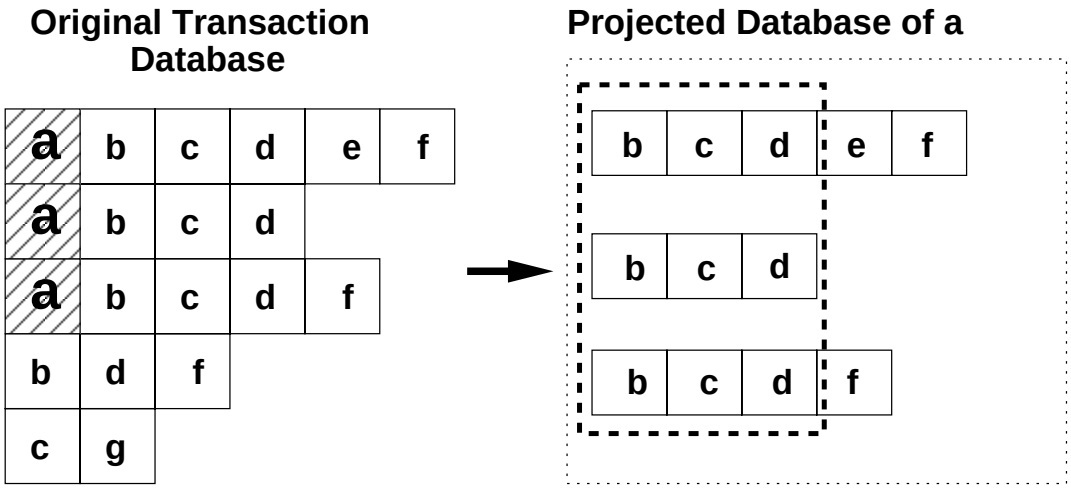


Figure 6.3: Hyper-cube decomposition in LCM

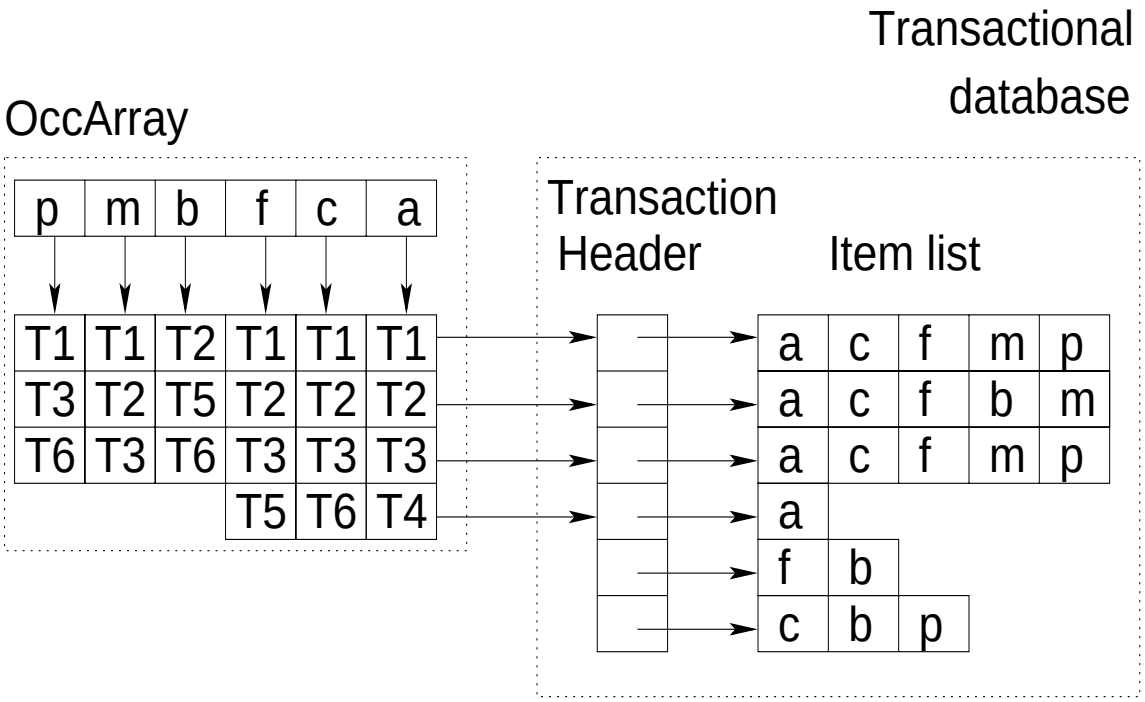


Figure 6.4: Array representation of the dataset in Figure 6.2-(a) in LCM

LCM uses arrays to represent projected transaction databases instead of FP_Trees. Figure 6.4 illustrates the data structure used by LCM to represent the same database shown in Figure 6.2-(a). The *transaction database* is a list of transactions, where each transaction is represented by an array containing item IDs. The *OccArray* has one record for each item; the record contains an array of pointers to the transactions that include the corresponding item. In most cases, the array representation of LCM is less compact than the FP_Tree in FP_Growth. However, the advantage of LCM is that it has more spatial locality on memory accesses than FP_Growth where the extensive use of pointers and associated pointer-chasing during the FP_Tree traversal can degrade performance.

6.2.3 Eclat

Eclat [ZPOL97] is another well-known depth-first-search frequent pattern mining algorithm. Eclat uses a vertical (item-major) representation of the database; each column record corresponds to an item, or an itemset, and lists the transactions containing this item (resp. itemset). During the recursive depth-first-search of the subset lattice, records are intersected to compute the record corresponding to the union of the two corresponding item sets.

Eclat can store the itemset records either in sparse or in dense format. Figure 6.6-(a) shows the dense representation of the data set in Figure 6.2-(a), where each item record is a bit vector, and 1 indicates the occurrence of an item in a transaction. Bit vector representation allows direct use of bit operation instructions. However, when there are too few 1's in the bit matrix, it is more efficient to represent the bit matrix in sparse format as shown in figure 6.6-(b), where

Algorithm 4 LCM algorithm

LCM ($\mathcal{T}$: transactional database, max: max item)

for $i \leftarrow 0$ to $max - 1$

call LCMITER(T, i)

LCMITER (T: transactional database, max: max item)

// θ: threshold

// $\mathcal{P}$: *current solution*

// $\mathcal{C}$: *closed itemset*

// $\mathcal{CAN}$:*candidate itemset*

// $\mathcal{INF}$: *infrequent set of items*

// $\mathcal{F}$: *frequent itemset*

$\mathcal{P} \leftarrow \mathcal{P} \cup \{max\}$

call CALCFREQ($\mathcal{T}$, max, $\mathcal{T}(max)$)

Foreach item $i < max$

If $|\mathcal{T}(i)| == |\mathcal{T}(max)|$

$\mathcal{C} \leftarrow \mathcal{C} \cup \{i\}$

If $\theta \leq |\mathcal{T}(i)| < |\mathcal{T}(max)|$

$\mathcal{CAN} \leftarrow \mathcal{CAN} \cup \{i\}$

// $\mathcal{P}|Set$ union each $s \in Set$ with $\mathcal{P}$

$\mathcal{F} \leftarrow \mathcal{F} \cup (\mathcal{P}|2^{\mathcal{C}})$

rebuild $TransTable$, removing items in $\mathcal{INF}, \mathcal{C}$

rmDupTrans($TransTable$)

for each item i in $\mathcal{CAN}$

LCMITER($TransTable$, i)

CALCFREQ(T: transaction database, max: max item

, *occ*: $\mathcal{T}(max)$)

// $freq[i] : |\mathcal{T}(i)|$

Foreach $t \in occ$

Foreach $i \in t$ and $i < max$

freq[i]++

Foreach $i < max$ and $|\mathcal{T}(i)| < \theta$

$\mathcal{INF} \leftarrow \mathcal{INF} \cup \{i\}$

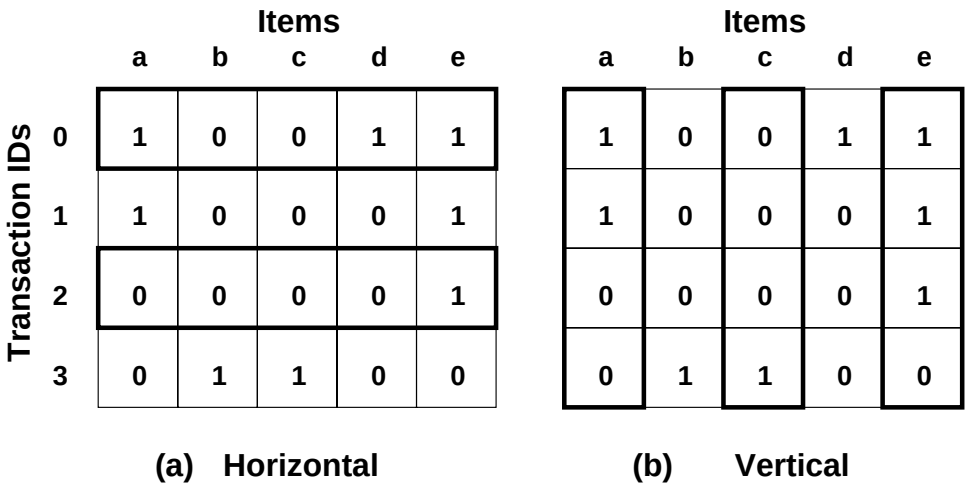


Figure 6.5: Horizontal vs. vertical representations

each record is a list of transaction IDs. [ZG03] proposed an optimization for vertical algorithms based on *diffset*. The idea is to only keep track of the differences in the transaction IDs of a candidate itemset from its generating item sets. The *diffset* idea can significantly reduce the memory usage of Eclat.

In this paper, we use the Eclat implementation [Bor04] from the FIMI workshop. This implementation switches between dense and sparse representation based on the bit-matrix density, but does not implement the *diffset* idea from [ZG03].

Algorithm 5 Eclat algorithm

ECLAT (M: transaction database)
  For $i \leftarrow n-1$ down to 0
    For $j \leftarrow 0$ to $i-1$
      // M_i is the ith row of matrix M
      $newRow \leftarrow M_i \wedge M_j$ ———— $(*)$
      $support \leftarrow count\_support(newRow)$
      If $support > \theta$
        add $newRow$ to $m\prime$
  ECLAT($m\prime$)

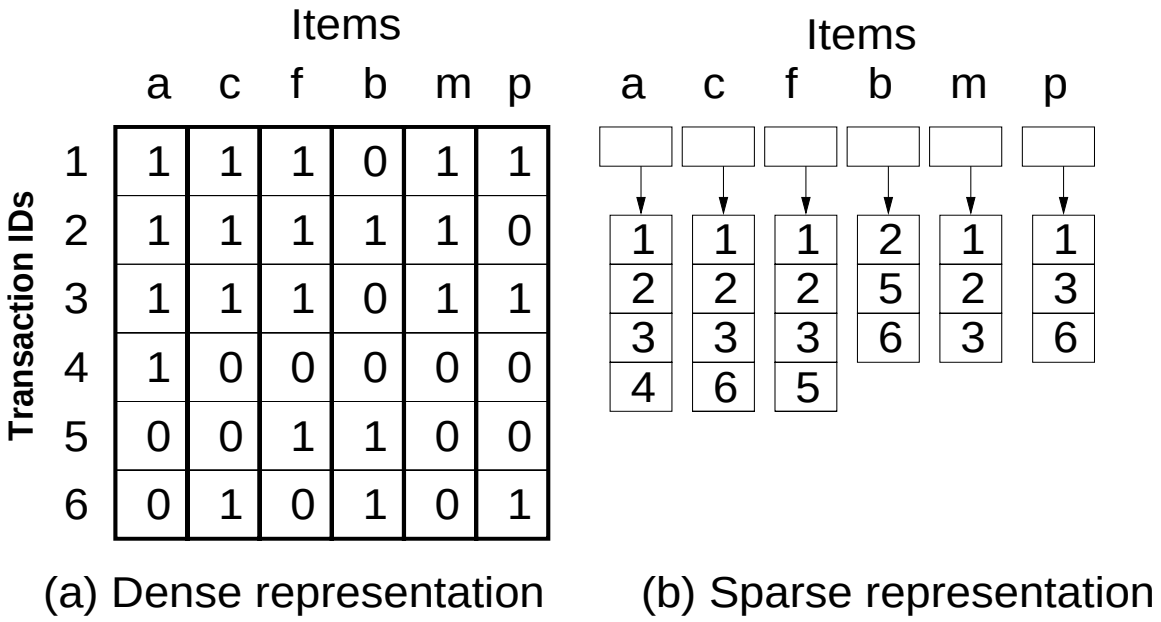


Figure 6.6: Dense and sparse vertical representation

6.3 Support Vector Machine

Support vector machine [Vap95] is a powerful kernel based machine learning algorithm. It has been widely used in many application domains for hard learning problems, such as optical character recognition, text categorization, and biological sequencing, etc. Next we briefly describe the SVM learning algorithm.

Support vector machine is a kernel based learning algorithm. The main idea of kernel based learning is to embed an input space X into a vector space $\Re^N$, of high dimensionality. After that linear algorithms, that are efficient and well understood, can be used for classification and regression. Figure 6.7 shows that two linearly non-separable classes become linearly separable after embedding the points from a two dimensional space into a three dimensional space. The embedding mapping is often denoted by $\phi : X \rightarrow \Re^N$.

We do not need to perform the embedding explicitly as long as we can compute the pairwise inner products of the image vectors of any pair of data points. We assume that a *kernel function* $K(x, y) = \langle \phi(x), \phi(y) \rangle$ is available to perform this calculation.

Consider a binary classification problem. A support vector machine embeds the input data points into a high dimensional feature space and then searches for a separating hyperplane that maximizes the minimum distance from any point to the hyperplane, as illustrated in Figure 6.9.

After an SVM based classification system is obtained, it can be used to predict the class of a test point by calculating on which side of the hyperplane a point lies. The framework can be easily extended to classifiers with more than two classes, using multiple separating hyperplanes.

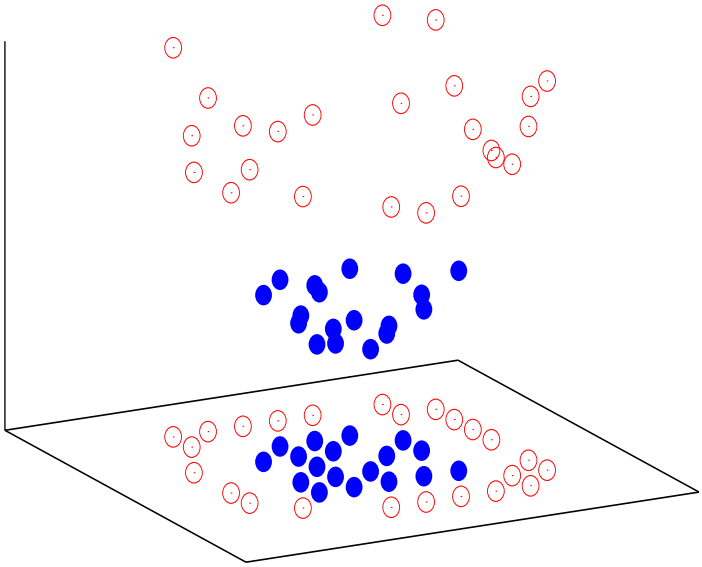


Figure 6.7: Two linearly non-separable classes become linearly separable after embedding the points from a two dimensional space into a three dimensional space

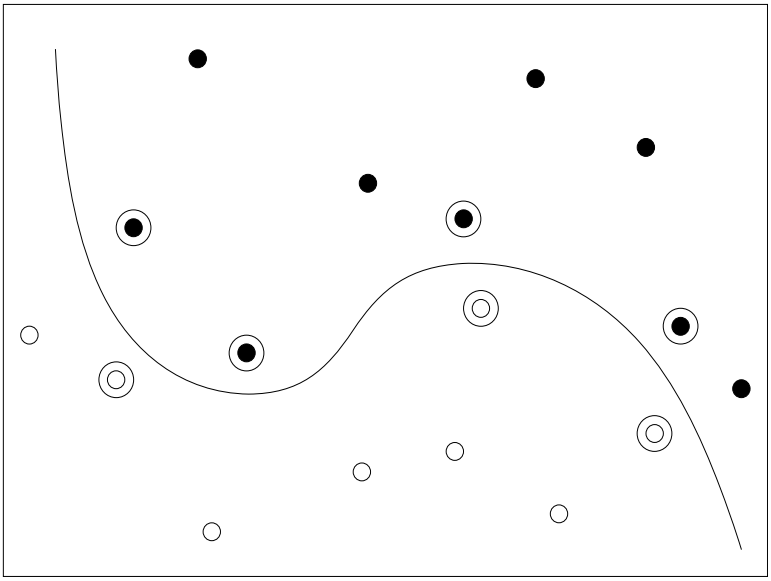


Figure 6.8: A separating plane in high dimensional space corresponding to non-linear separating plane in the original space

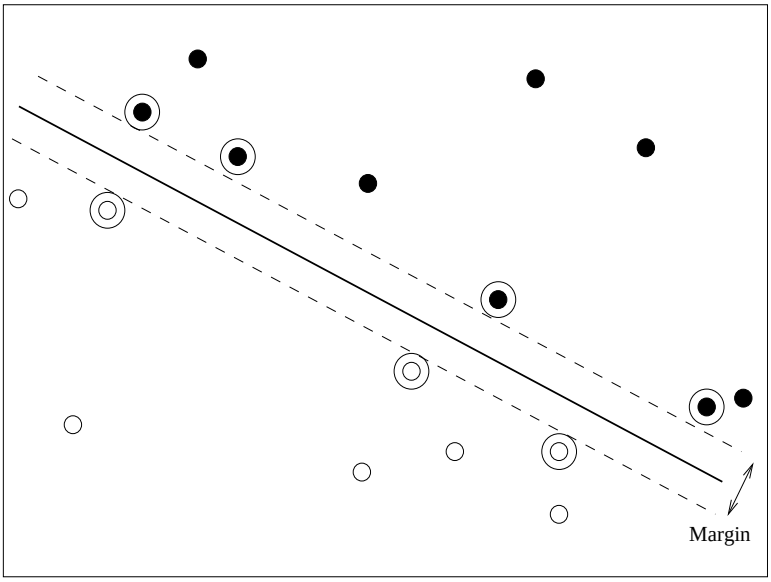


Figure 6.9: a separating hyperplane that maximizes the minimum distance from any point to the hyperplane

Kernels	**Definition**
Linear	$K(x_i, x_j) = x_i^\top x_j$
Polynomial	$K(x_i, x_j) = (\gamma x_i^\top x_j + r)^d, \gamma > 0$
Radial Basis Function (RBF)	$K(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}, \gamma > 0$

Table 6.1: Some commonly used SVM kernels

Different kernel functions can be plugged into the SVMs framework in a modular manner. Table 6.1 shows some commonly used kernel functions by SVM. Customized kernel functions can also be used to incorporate domain-specific knowledge.

Chapter 7

Automatic Algorithm Selection for Frequent Pattern Mining using Support Vector Machine

7.1 Introduction

Frequent pattern mining is the problem of finding subsets of items that occur frequently in a set of transactions. It has many applications such as finding association rules, correlations, or causality. Since the introduction by Agrawal et al. [AIS93], a large number of algorithms have been proposed. During the FIMI (Frequent Itemset Mining Implementations) workshops [GZ03a, JGZ04] different implementations of well-known pattern mining algorithms were submitted and their relative performances were evaluated on a few datasets. The results of these experiments showed that there was no single algorithm that outperformed all the others on all data sets. Figure 7.1 shows some experimental results that we ran with three frequent pattern mining algorithms (FP_Growth, LCM and Eclat) on some of the real-world data sets taken from the repository in the FIMI workshop. As the figure shows none of these algorithms is always the best. However, there is no clear understanding of the input characteristics that determine which is the best algorithm. The problem of how to systematically choose the best algorithm for any given input remains largely untouched.

When no algorithm dominates, one is faced with the problem of deciding which algorithm to use; in 1976 Rice named this the *algorithm selection problem* [Ric76]. The basic idea is that the algorithms provide a classification of inputs, with each class consisting of the set of inputs for which a particular algorithm is better. One wishes to find a classifier that associates (with high probability) each input to the correct class. One approach that has been studied by several authors [LBNA+03, LGP05, TTT+05] is to use machine learning algorithms for this purpose: A training set of inputs are selected; the algorithms are run on these inputs and each input is labeled with the best algorithm; the classifier is trained to identify the labels; the resulting classifier is then used during execution to select which algorithm to run.

There are two choices that are critical to the success of this approach: (1) one needs to select a set of features that are used to classify inputs; this feature set should be easy to evaluate and should be sufficient to distinguish inputs that have different labels; and (2) one needs to use a training set that is reasonably representative of real inputs. Both are hard problems especially for frequent pattern mining. To solve (1) we need to understand the design

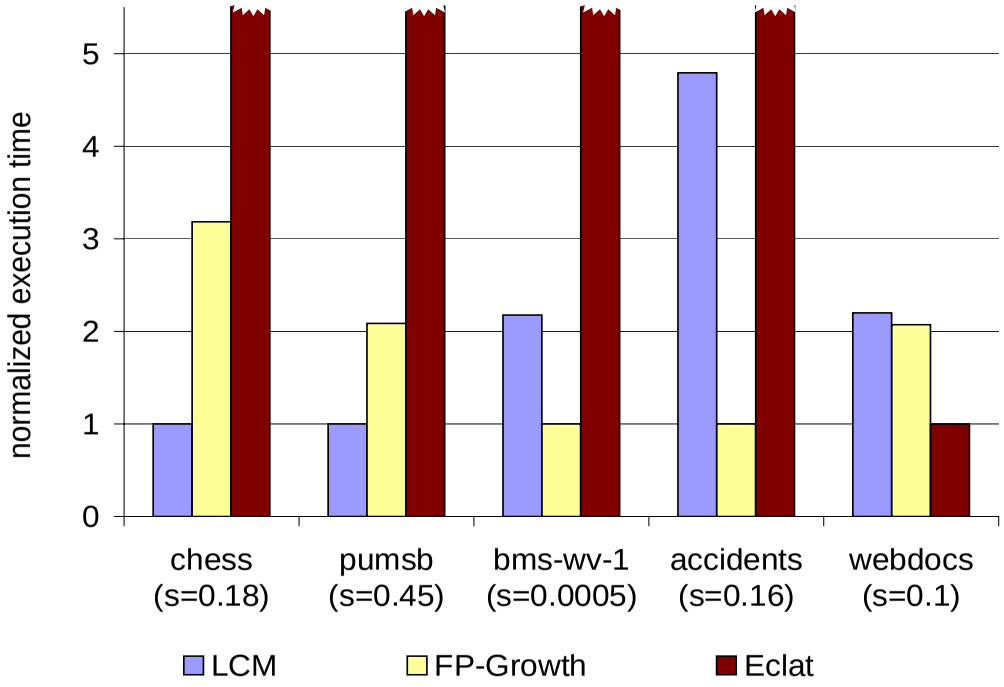


Figure 7.1: Relative performances of FP_Growth, LCM and Eclat on some real datasets; s stands for support threshold.

and implementation details of the different algorithms, understand the interactions between the algorithms and the target platform, and find the input characteristics that are more relevant to determine their relative performances. The difficulty for (2) is that there are only a few publicly available real-world data sets, so synthetically generated data sets need to be used during training. However, the data sets generated by the currently available generators such as the IBM Quest Dataset Generator are not representative of the real world data sets.

In this Chapter, we present an SVM (support vector machine) [Vap95] based learning method to train such a classification system. We took implementations of FP_Growth, LCM and Eclat from the FIMI workshop implementation repository and trained our SVM system by running these three algorithms on synthetic data sets. Then, we tested the trained SVM system to predict the best algorithm on a few real-world data sets.

The main contribution of this paper is to demonstrate the viability of this machine learning approach for frequent pattern mining algorithm selection. In particular (1) we identify a set of input features that can guide the selection and (2) we show how to generate synthetic data sets that are representative of the real-world data sets and that can be used to train the input classifier. As a result of our contribution, we obtain an algorithm that is better that any of the three selected algorithms (LCM, FP_Growth, and Eclat). Our experiments show that the performance of the predicted algorithm is on the average only 12.5% worse than that of the optimal algorithm, and 65.3% better than LCM that

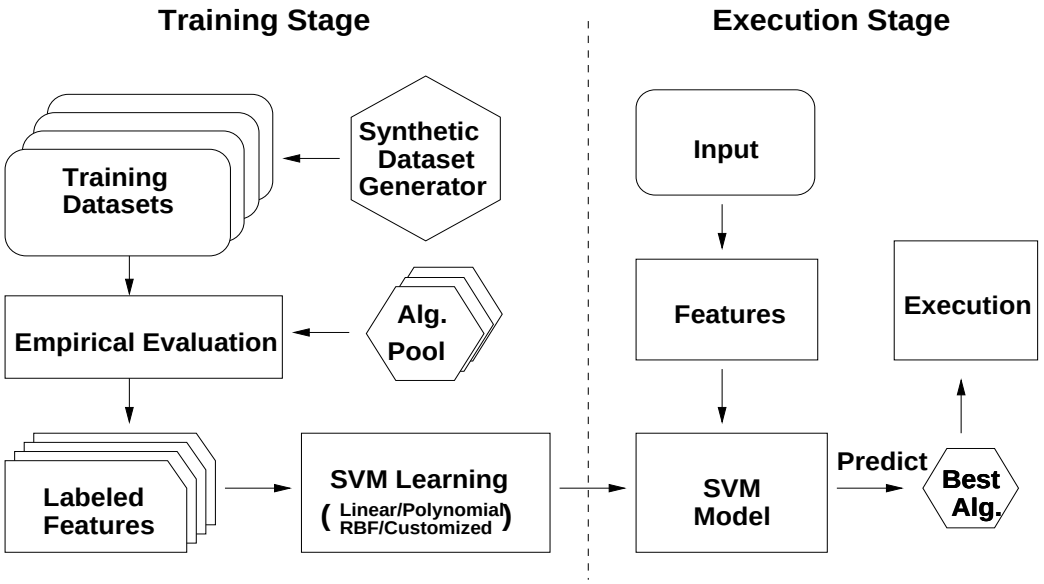


Figure 7.2: The components and the work flow of our SVM based algorithm selection system

obtained the best average performance for all the tested inputs. We believe that the approach will extend to other choices of basic frequent pattern mining algorithms.

7.2 Our Approach

In this Section we describe the approach we have followed for algorithm prediction. Section 7.2.1 presents an overview of the approach, Section 7.2.2 discusses the features used for the selection, and Section 7.2.3 discusses the synthetic data generator used to train the SVM system.

7.2.1 The Algorithm Prediction Framework

Figure 7.2 depicts the components in our SVM based algorithm prediction system, which can be divided into two stages: the *training stage* and the *execution stage*. In the training stage, a synthetic data set generator is used to randomly generate thousands of synthetic data sets. The three algorithms are empirically evaluated by running them on the synthetic data sets with different support thresholds. Each input $d = (t, \xi)$, which consists of a data set t and a support threshold ξ, is represented by a set of feature values $x \in \Re^N$ and labeled with the best algorithm found for it during the empirical evaluation. These labeled training points are input to the SVM learning module to train an SVM model.

During the execution stage, the feature values $x' \in \Re^N$ of an input (t', ξ') are extracted at runtime. The SVM model produced in the training stage is consulted to predict the best algorithm for the input based on its feature values

x'. The predicted algorithm is then invoked to perform the actual mining task.

7.2.2 Feature Selection

Selecting the right features for the learning module is critical for the system to predict accurately. In fact, most of our research efforts have been dedicated to the search of the appropriate features to differentiate the three algorithms. Before we explain our selected features, it is important to notice two issues. The first is that the operation needed to extract the feature values from a given data set must be computationally cheap. If the feature values are too expensive to compute, the benefit of accurate algorithm prediction will be offset by the added run-time overhead. This requirement precludes the use of features directly related to the actual mining results. The second issue is that the features should be extracted after filtering out all the infrequent items from the data set, rather than using the original input data set. The reason is that all frequent pattern mining algorithms need to filter out infrequent items before starting the mining process. The filtering process typically accounts for a small portion of the total execution time, and as a result, filtered data set's features capture more accurately the input characteristics determining algorithm performance. Thus, from now on in this section the input data set will refer to the filtered input data set.

Qualitative Analysis on Algorithm Selection

In order to select a useful feature set, it is important to have a qualitative understanding of input features that may cause one algorithm to run faster than another. Since all these algorithms traverse the search space in the same order (depth-first order), the major difference between them is in the data structure used to represent the database. FP_Growth uses a FP_Tree, LCM uses two arrays, and Eclat uses a bit matrix. The bit matrix representation is more efficient when the database is large and dense. However, since the Eclat implementation we use does not implement the *diffset* idea, the bit matrix gets sparser when recursing down the search space, and becomes more and more inefficient compared with LCM's arrays and FP_Growth's FP_Tree. Hence, intuitively Eclat will have better performance than LCM and FP_Growth when the input database is large, dense, and the search space is shallow.

The major difference between LCM and FP_Growth is the data structure and the number of projected databases. LCM only recurses for closed itemsets and enumerates all other frequent itemsets by using the *hyper-cube decomposition*. If the number of frequent closed itemsets is much smaller than the number of frequent itemsets, LCM should have better performance. When the number of frequent closed itemsets is close to the number of frequent itemsets, the representation of the database (arrays versus FP_Tree) is the main factor determining the performance difference between the two algorithms. If the FP_Tree representation can effectively compact the database so that the compressed tree can fit in the cache, while the array representation exceeds it, FP_Growth is likely to perform better. The problem of FP_Growth is that it uses several pointers for each node in the tree, and if the compression ratio of the FP_Tree

structure is not big enough, the FP_Tree may end up using more memory than the array. In addition, traversing the trees in FP_Growth requires extensive pointer chasing, what results in less spatial locality and more non-overlapped memory accesses than the array structure of LCM.

Selected Features

In our search for the appropriate features, we have tried many intuitive features and their combinations, such as '*the number of transactions*', '*the number of frequent items*', '*the support threshold*', '*the supports of all frequent items*', '*the supports of some pair itemsets*', '*the histogram of transaction lengths*', '*average transaction length*', '*standard deviation of transaction length*', '*average support of frequent items*', and '*standard deviation of frequent item supports*'. These features were not effective at predicting the best algorithm. To predict accurately, we had to thoroughly study every aspect of the algorithm design and the implementation details of them to search for features that best differentiate their performance.

The qualitative analysis of Section 7.2.2 suggests that we should focus on features that estimate the sparseness of the data set, the depth of the recursion tree, and the frequency of co-occurring items – the similarity between transactions. This motivates the choice below.

Feature	**Description**
size	The size of the input
density	The density of the data set
height	A measure of search depth
similarity	A measure of transaction similarity

Table 7.1: The selected features.

Table 7.1 lists the selected features: *size*, *density*, *height*, and *similarity*. We further justify the feature selection in Section 7.2.2.

- *size* captures the size of the problem. It is the total number of occurrences of items in transactions (the number of 1's in a bit matrix representation of the database).
- *density* captures the density of a data set. It is the average number of 1's in the bit matrix representing the database. This is mathematically equivalent to the average transaction length, divided by the number of items, or the average number of occurrences of an item, divided by the number of transactions.
- *height* is defined as $h = 1 - \xi/d$, where ξ is the support threshold and d is the density. Since d is a measure of item frequency, h is a measure of the "headroom" in the database, an estimate of how far from the support threshold it is.

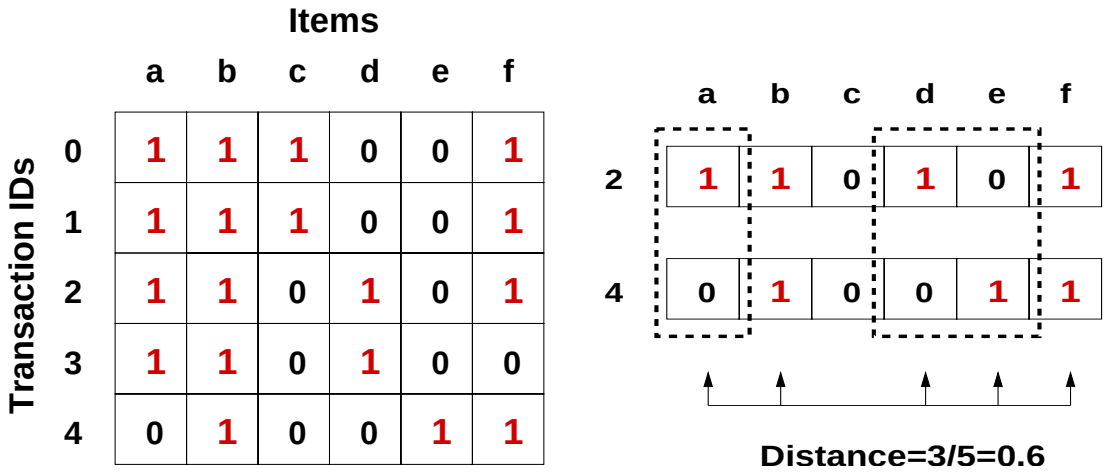


Figure 7.3: Examples for the Selected Features

- *similarity* captures how similar the transactions are to each other.

As an example, suppose that we have an input consisting of the data set in figure 7.3(a), with a normalized support threshold of $\xi = 0.2$. Its $size$ is 18, its $density$ is $18/30 = 0.6$ and its $height$ is $1 - 0.2/0.6 \approx 0.667$. These first three features are relatively easy to understand and compute, whereas the *similarity* feature needs more explanation. This is done in the next section.

Similarity Between Transactions

We use a normalized Hamming distance to measure the distance between transactions. By definition, Hamming distance between two strings of equal length is the number of positions for which the corresponding symbols are different. We normalize the Hamming distance by dividing by the total number of unique items contained in both transactions. The normalized Hamming distance between two itemsets P and Q is formally defined as $1 - |P \cap Q|/|P \cup Q|$.

Figure 7.3(b) illustrates the normalized Hamming distance of two transactions of the data set in figure 7.3(a). In the example, the number of items occurring in only one transaction is 3. The total number of unique items is 5. Hence the normalized Hamming distance between them is 0.6.

Normalized Hamming distance computes the distance between a pair of transactions, but what we need is a metric that captures global similarity. We do that by using clustering. We use the *average-linkage hierarchical agglomerative clustering*, which works as follows. We start with N transactions, each of which is considered a cluster, compute the $O(N^2)$ pair-wise distances between these clusters, and save them in an array. Then, the clustering process clusters the N transactions into one cluster in $N - 1$ steps. At each step, the two nearest clusters (the ones with the smallest

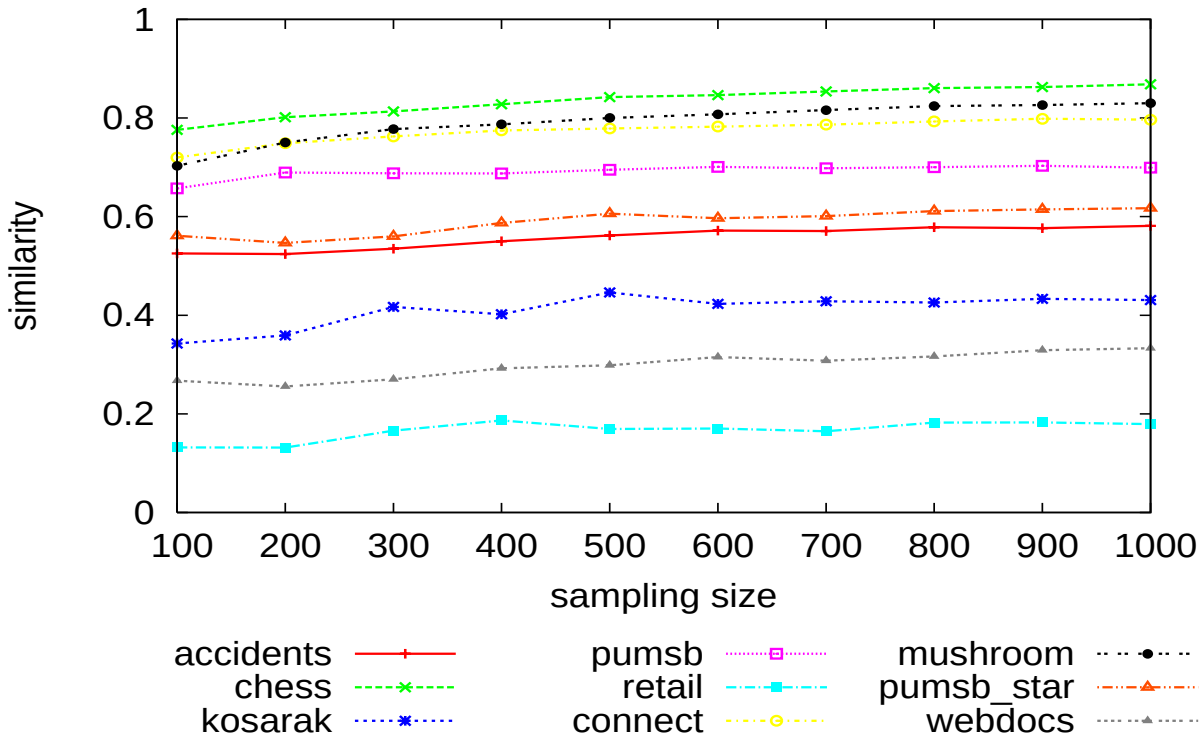


Figure 7.4: Similarity vs. Sampling Size

distance) are grouped into one new cluster. These two clustered are deleted from the array, and the new cluster is added to the array. To compute the distance between the new cluster and all the other clusters in the array we use the "*average-linkage*", which computes distance between two clusters as the average of distances between all pairs of transactions, where each pair is made up of one transaction from each cluster. During the $N-1$ steps, $N-1$ inter-cluster distances are found. We use the average value of these $N-1$ values as the metric that measures the similarities of all the transactions.

One problem with the hierarchical clustering algorithm is that it has $O(N^2)$ complexity. Real-world data sets usually have tens of thousands of transactions or even more, so it is very expensive to compute the clustering using all the transactions. To overcome the problem we have used random sampling. To validate the use of random sampling, we have checked the effect of sample size when varying the size of the sample from 100 to 1000 randomly chosen transactions from the real-world data sets described in Section 7.3.1. Figure 7.4 shows the similarity values measured. The Figure shows that as the sampling size increases, the similarity value changes slightly. It also shows that the relative differences between the data sets' similarity values is almost independent of the sampling size.

Rationale Behind the Selected Features

In section 7.2.2, we have discussed some qualitative characteristics of the input data set that will determine the relative performance of the three algorithms we are interested in. In this section we relate those characteristics to the selected features.

Eclat's performance would be better than the other two when the data set is large, dense, and the search space is shallow. *Size* and *density* can capture the size and density of the data set, and the *height* together with the *similarity* can determine the depth of the search space. *Height* tells how much room is available for the support to decrease from the average item support values to the support threshold. The higher this value is, the more room there is for the support to decrease. *Similarity* determines how quickly the support of the itemsets decrease in the subset lattice. The more correlated the transactions are, the more slowly the support values decrease in the search space. Therefore, the smaller *height* and *similarity* are, the more shallow the search space is.

Differentiating between LCM and FP_Growth is harder. The main factor to determine the best of these two algorithms is the difference between the number of frequent closed itemset number and the number of all frequent itemsets. This is captured by the *similarity* feature, since the more similar the transactions are, the more likely is that there are items occurring in all transactions of a projected database. If the transactions are not similar, the relative performance of LCM and FP_Growth will be determined by the efficiency of the array and FP_Tree representations. When the problem size is small enough to fit in L2 cache, FP_Growth will be less efficient. The pointer-chasing overhead to traverse the FP_Tree will degrade performance of FP_Growth compared to LCM, since the access to the array representation of LCM will be faster. If the data set is large, and overflows the L2, FP_Growth can outperform LCM because the FP_Tree structure can compress the data set more efficiently than the array representation. The *size* feature can help to predict this. Finally, experimental results have shown that LCM behaves better than FP_Growth when the data set is sparse, as measured by the *density* feature. However, we have not observed significant differences between LCM and FP_Growth based on the *height* feature. Table 7.2 shows each algorithm's favorite area in the feature space.

Algorithms	**Size**	**Density**	**Height**	**Similarity**
FP_Growth	large	medium	big	medium
LCM	small	sparse	big	high
Eclat	large	dense	small	low

Table 7.2: The three algorithms' favorite area in the feature space.

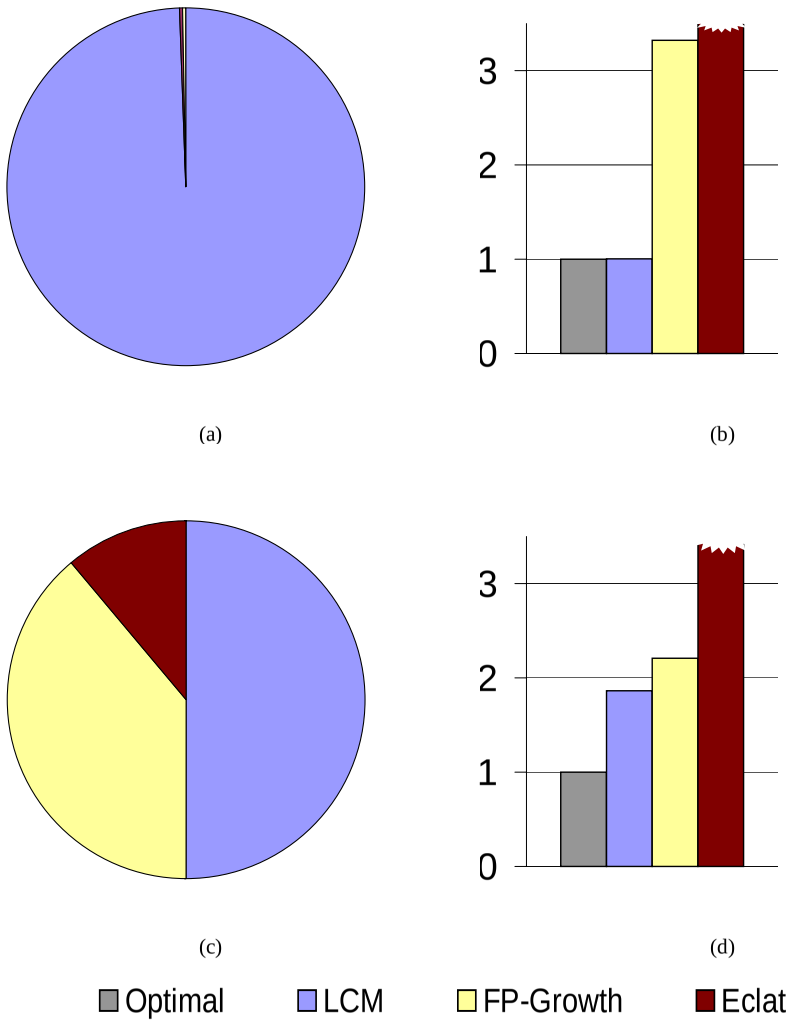


Figure 7.5: The fastest algorithm distribution on synthetic data sets and real data sets.

7.2.3 Synthetic Dataset Generator

In order to train a classification system, a large number of training points are needed. However, since there is only a limited number of real-world data sets, we had to train our system using synthetic data sets, and then use the real world data set for a blind test.

A natural choice of a synthetic data set generator is the IBM Quest Dataset Generator, which has been widely used by the data mining research community. However, when using the IBM generator we found that LCM was almost always the best algorithm for the synthetic data sets generated using the IBM generator. Figure 7.5(a) shows that for the data set generated using the IBM generator LCM was the best algorithm in 99.53% of the 1300 synthetic inputs

we tested, while the other two algorithms only won in 0.47% of the inputs. For this experiment the IBM generator parameter values for "*number of transaction*", "*number of items*", and "*average transaction length*" were set to a randomly selected value in the range of the actual values of the real-world data sets. Figure 7.5(b) shows that on the synthetic data sets, LCM's average performance is quite close to optimal, that corresponds to the performance obtained when the data set is mined using the best of the three algorithms. However, for the real-world data sets, figure 7.5(c) shows that LCM is the best algorithm in only 50% of the cases. The performance of optimal is almost twice better than that of LCM as shown in figure 7.5(d).

Previous research by Zheng et al. [ZKM01] already noticed the inconsistencies in the performance behavior of the mining algorithms on the IBM generated data sets and the real-world data sets. We have investigated the reasons for these differences, and found that the item frequency distribution of the real world data sets is very different from the one in the synthetic data sets. As shown in Figure 7.6, if we sort the item frequencies in descending order (item frequencies are normalized by the transaction number), the curves of the synthetic data sets, which are the two used by FIMI workshop, are much lower and smoother than those of the real-world data sets. The first consequence of this is that synthetic data sets tend to be sparser than real-world data sets (and LCM tends to perform better in sparse data sets).

Obviously, the data sets generated by the IBM generator are not appropriate to train our SVM system. As a result, we had to modify it to generate synthetic data sets that have algorithm variability. We studied how the IBM generator works, and observed that in a first stage, the generator uses an "*exponential*" distribution to generate $nitems$, which is a generator parameter, random numbers n_i. They are then normalized by dividing by the sum of all of them ($\sum_{i=1}^{nitems} n_i$). The normalized $nitems$ random numbers are used to obtain a cumulative probability distribution from where items will be selected to build *patterns* (in the IBM generator *patterns* are sets of items which allow duplicates) that are subsequently used to build transactions in the next stages of the generator. Since the number of items is usually large (the minimum value of $nitems$ in the IBM generator is 1000), the normalization will result in these $nitems$ items having a similar and low probability of being chosen, as shown by the item frequency curves in Figure 7.6. We modified the IBM generator in two aspects: (1) we decrease the minimum value of $nitems$ from 1000 to 100, which causes the result data set more similar to the real-world data sets. (2) In addition to the *exponential* distribution, we inject various kinds of item frequency distributions, such as "*Gaussian*", "*Uniform*", "*Zipf*", and "*Poisson*" distributions. Figure 7.7 shows that after injecting these new item frequency distributions, Eclat began to win in more than 30% of the cases, but FP_Growth was still relatively inferior in performance in almost all synthetic data sets. Therefore, we used a more advanced statistical tool – "*kernel density estimator*" to simulate the item frequency distribution from some real-world data sets.

Kernel density estimator is a generalization and improvement of histogram in statistics. Suppose we have a his-

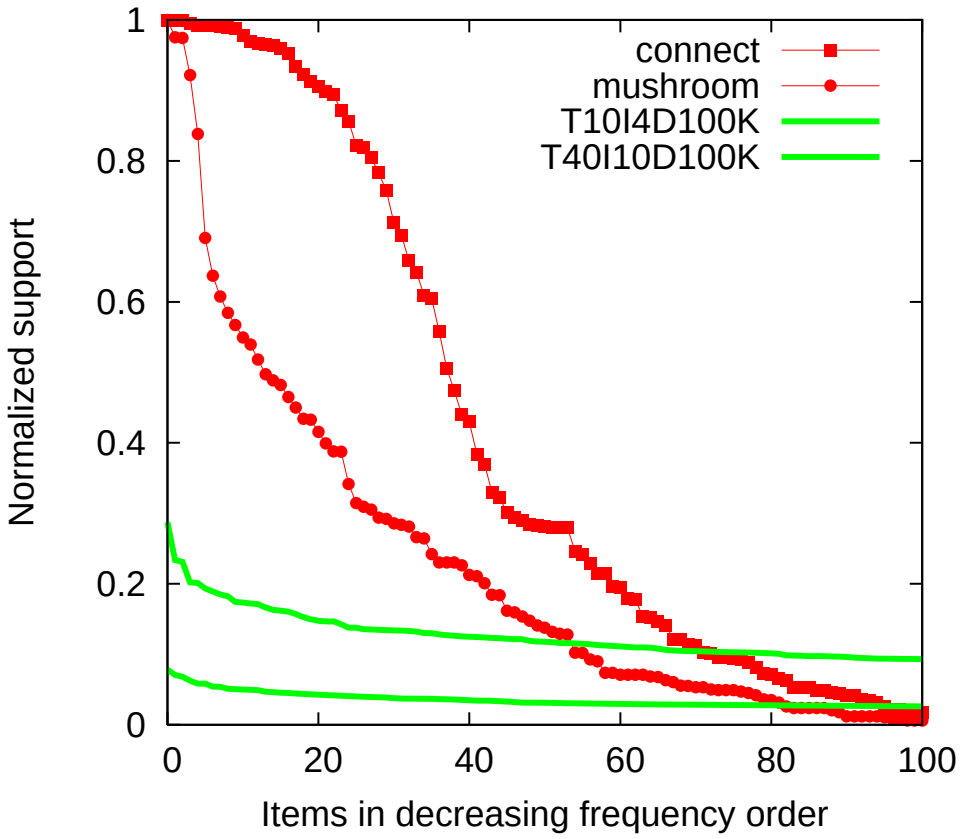


Figure 7.6: Item frequency curves for synthetic data sets and real-world data sets

togram of the item frequencies as illustrated in Figure 7.8(a), where the number of items that have a given item frequency is shown. The problem with histogram is that it is not smooth and depend on the width of the bins (in the Figure the width is 1). *Kernel density estimator* can be used to achieve a smooth non-parametric probability distribution function to approximate the underlying probability distribution function of the histogram.

The idea is to calculate the probability density function value of any point by a weighted sum of the function values at the observed points. Kernel estimators smooth out the contribution of each observed data point over a local neighborhood of that data point. The contribution of data point x_i to the estimate at some point x' depends on how apart x_i and x' are. The extent of this contribution is dependent upon the shape of the kernel function adopted and the width (bandwidth) accorded to it. If we denote the kernel function as K and its bandwidth by h, the estimated density at any point x' is:

$$f(x') = \frac{1}{n}\sum_{i=1}^{n} K(\frac{x' - x_i}{h})$$

The choice of the kernel function K usually is not as important as the choice of the bandwidth h. In our case, we use the "*Gaussian*" kernel function $K(u) = \frac{1}{\sqrt{2\pi}} exp(-\frac{1}{2}u^2)$ As for the bandwidth, we empirical tested a few values

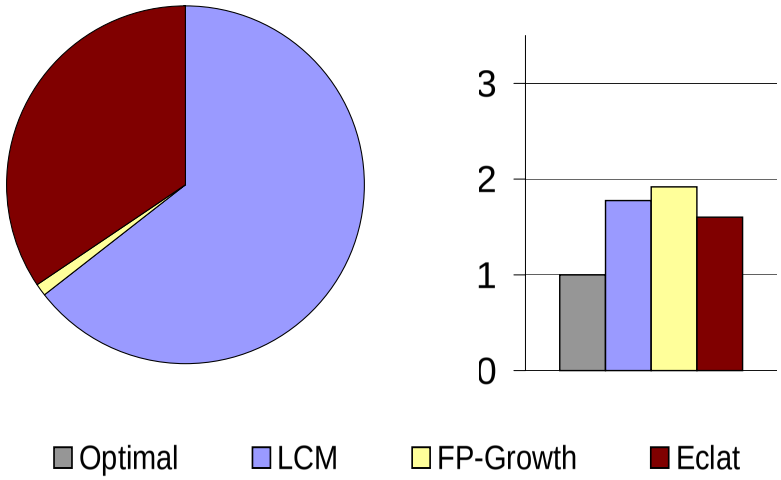


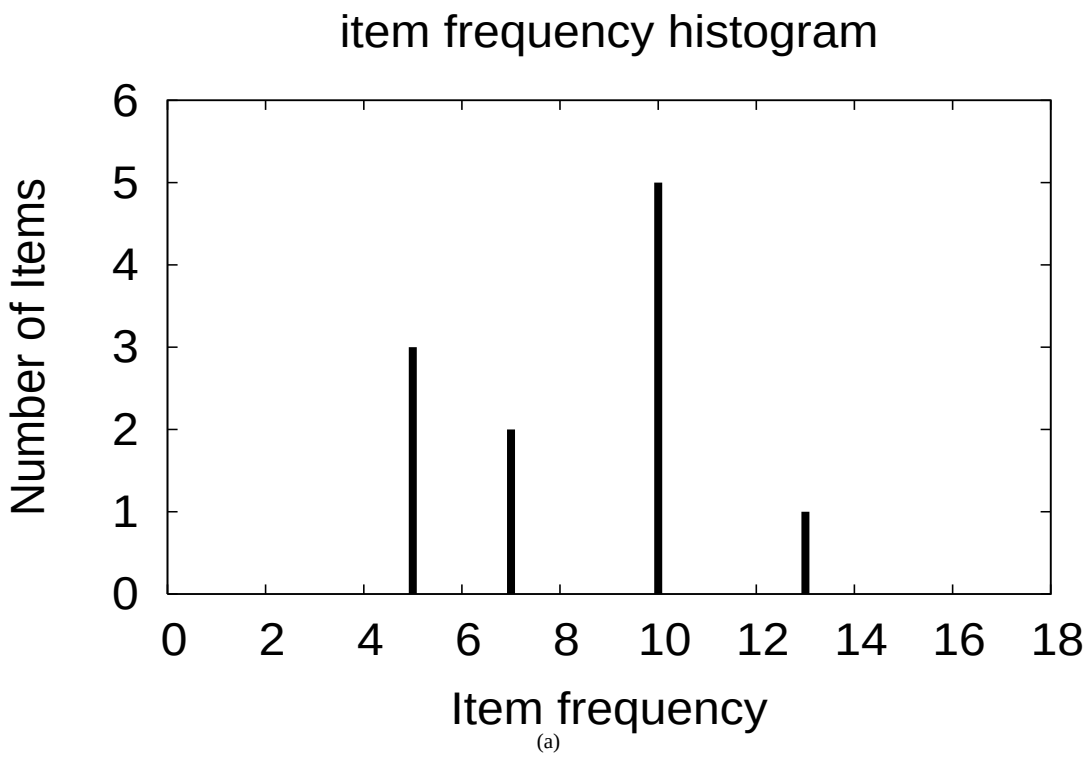
Figure 7.7: The fastest algorithm distribution on synthetic data sets generated by the modified generator with various injected item frequency distributions.

and choose the one that fits the best. In figure 7.8(b), it shows the estimated probability distribution function from the histogram of figure 7.8(a) with kernel density estimator.

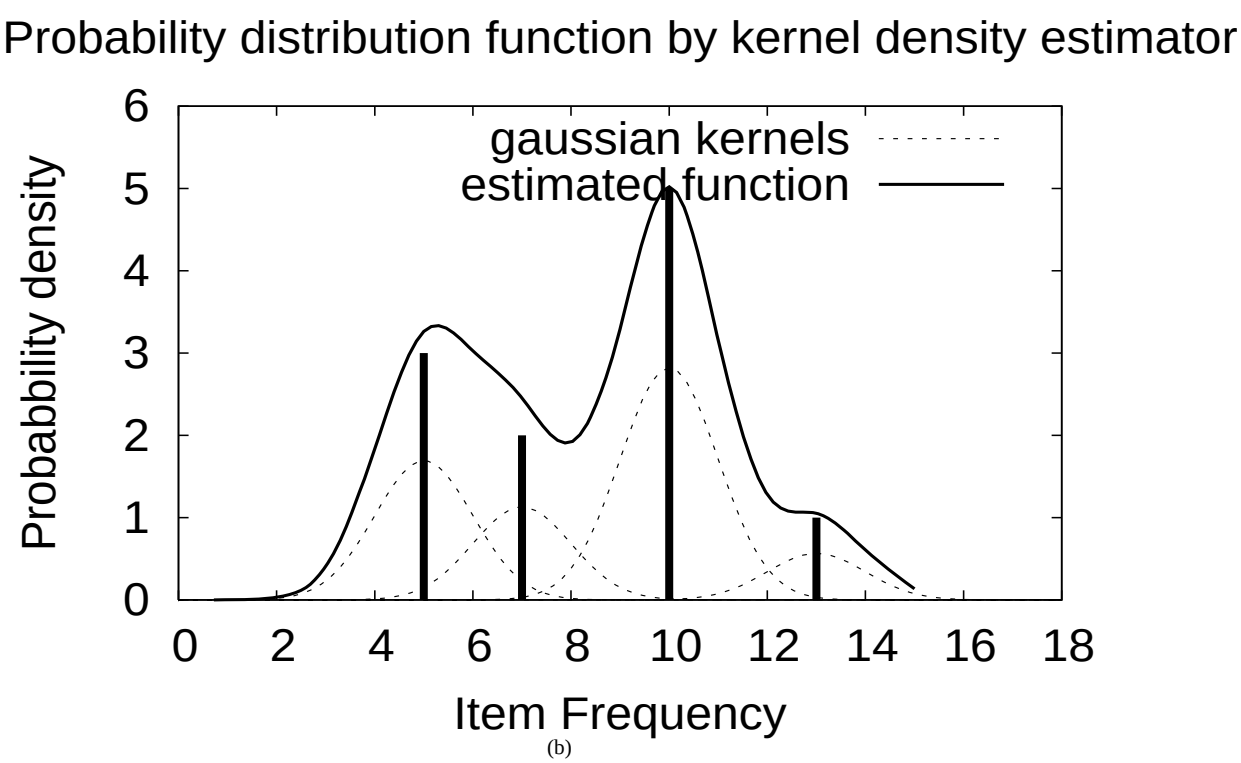
After the item frequency distribution function is estimated, we use "*rejection sampling*" technique to get random numbers from the estimated distribution function. Rejection sampling is a technique used to generate observations from a distribution function $f(x)$, where the form of $f(x)$ makes sampling difficult. Instead of sampling directly from the distribution $f(x)$, we use an envelope distribution $Mg(x)$ where sampling is easier. These samples from $Mg(x)$ are probabilistically accepted or rejected.

Figure 7.9(a) shows two item frequency curves obtained by using kernel density estimator compared with those curves of the real-world data sets. Figure 7.9(b) shows the curves of the final data sets generated by the modified generator compared with those of the real data sets. The curves in Figure 7.9(a) are used in the first step of the IBM generator, but due to IBM generator's special working mechanism, they are smoothed by the second and third step, resulting in the curves in Figure 7.9(b).

Figure 7.10 shows the best algorithm distribution on the synthetic data sets generated by the modified IBM generator, which uses a mix of some classic probability models and the probability distribution obtained from the real-world data sets through *kernel density estimator*.

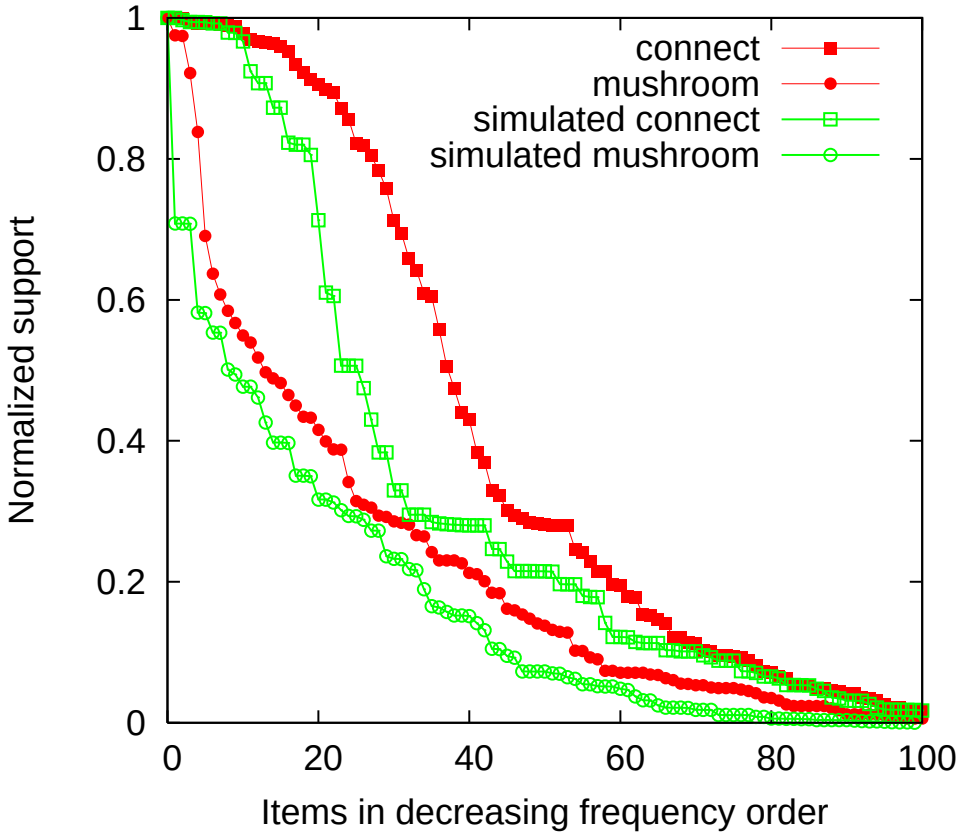


(a)

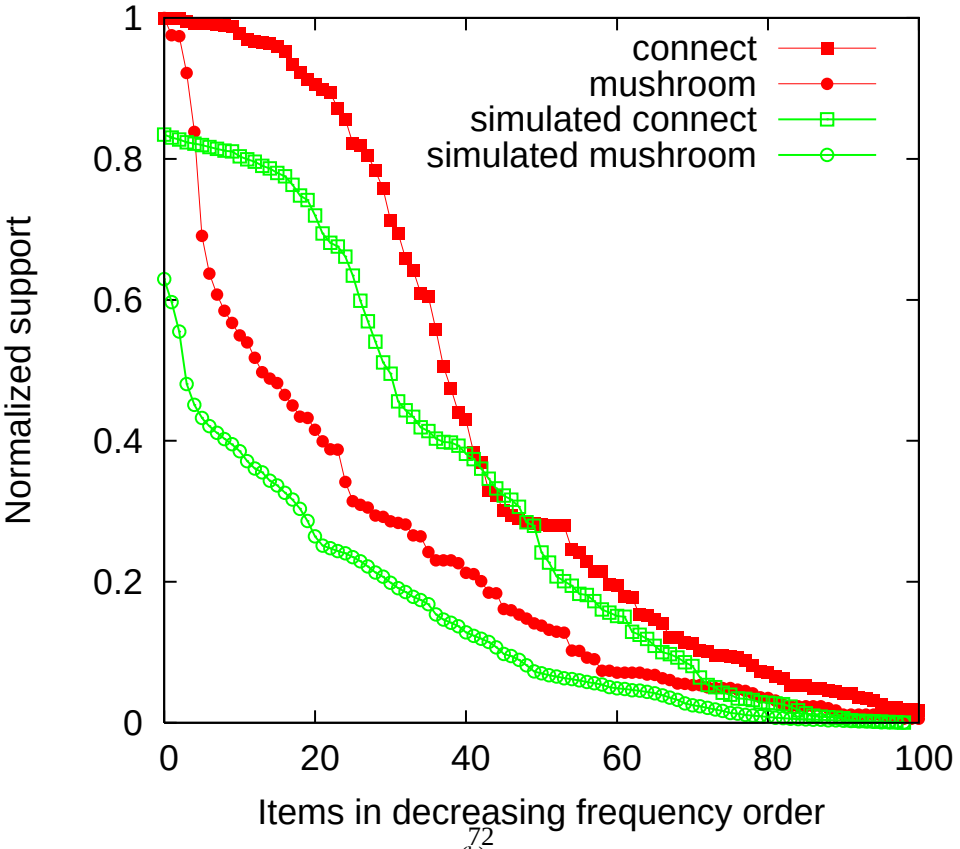


(b)

Figure 7.8: An Example for Kernel Density Estimator



(a)



(b)

Figure 7.9: Item frequency curves generated by the modified generator

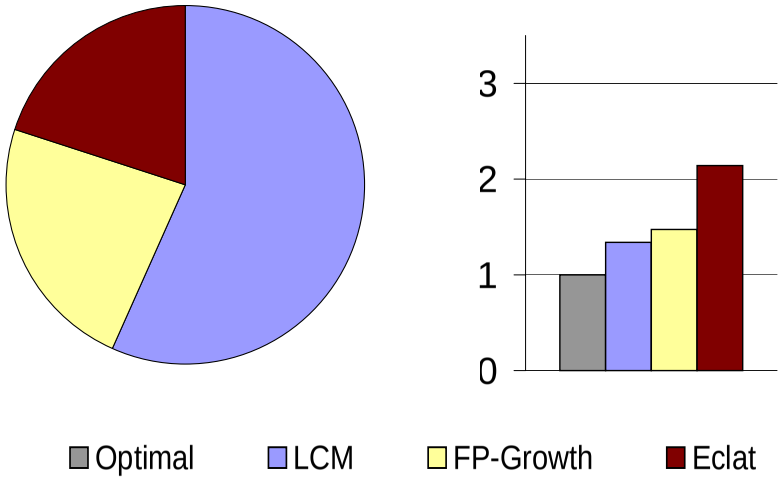


Figure 7.10: The fastest algorithm distribution on synthetic data sets generated by the modified generator with injected item frequency distributions by kernel density estimator.

7.3 Experimental Results

In this Section we describe the experiments we conducted to test the effectiveness of our prediction system. Section 7.3.1 discusses our experimental setup, while Section 7.3.2 presents the results obtained.

7.3.1 Experimental Setup

We trained our SVM based classification system using a large number of synthetic data sets. Then, we tested the trained SVM system on real-world data sets.

Real-world data sets: We used 12 real-world data sets from the FIMI workshop. These real-world data sets are: "*accidents*", "*chess*", "*webdocs*", "*pumsb*", "*pumsb_star*", "*connect*", "*mushroom*", "*retail*", "*korasak*", "*BMS-POS*", "*BMS-WebView-1*" and "*BMS-WebView-2*".

Training set generation: We generated one thousand synthetic data sets using the modified IBM generator described in Section 7.2.3. The input parameters to the generator, such as "*number of transactions*", "*number of items*", "*average transaction length*", are randomly chosen within the range of the values in the real world data sets. There are other parameters such as "*average pattern length*", "*confidence of patterns*" and "*correlation between patterns*" which are randomly generated from an arbitrary range. The item frequency distribution is randomly picked from "*Gaussian*", "*Zipf*", "*Poisson*", "*Uniform*", "*Exponential*" and "*Real*" distributions. The "*Real*" distribution simulates the item frequency distribution of the "*chess*", "*connect*", "*accidents*", "*mushroom*", "*pumsb*", "*pumsb_star*" or "*retail*" data sets with equal probability using the kernel density estimator. The 1,000 synthetic datasets are generated independently.

Each data set is used with 10 different support thresholds, which results in a total of 10,000 synthetic inputs. The 10 support thresholds were chosen from the range that make the number of frequent items to be in the range of the real-world data sets.

Performance evaluation: FP_Growth, LCM and Eclat are used to mine the 10,000 synthetic inputs. For this empirical evaluation we used a multi-computer cluster. The cluster consists of 100 computing nodes, where each node has a dual 1 GHz Pentium III Xeon processor and 1GB of memory. Only one of the processors was used to avoid memory contention between the two processors in each node. For the evaluation we enforce a maximum execution time of 800 seconds. If the algorithm does not complete within that period, the process is terminated and the maximum execution time is recorded as the execution time. By concurrently using 100 processors, the total performance evaluation takes 6 to 10 hours.

Algorithm implementations: The LCM, FP_Growth and Eclat implementations we use are all from the FIMI 2004 workshop's implementation repository. We fixed a performance bug of the FP_Growth implementation which deals with single-path FP_Tree by recursively enumerating the powerset of the items in the path. We replaced it with the routine from LCM which analytically calculates the supports for the powerset instead of enumerating them recursively. This optimization improves FP_Growth's performance on "*pumsb_star*" data set with small support thresholds by an order of magnitude, and it also improved FP_Growth's predictability. For Eclat, we fixed a small performance bug, which unnecessarily keeps all empty transactions after filtering infrequent items. The LCM implementation is used as provided by the FIMI workshop.

Training points selection: Although we generated a total of 10,000 synthetic inputs, not all of them were included as training points. The reason is that some inputs are more useful than the others for training. For example, some inputs are too trivial for the three algorithm since they all complete in less than 1 second. Some inputs are too hard for all of them and none of them completes within the given maximum period. Obviously, these two kinds of inputs are of little value for training. The operating system might introduce performance noise into the evaluations by context-switching, page-swapping etc. Some inputs are more susceptible to the noise especially when the difference between the fastest algorithm and the second fastest is small. Also, we prefer to improve the prediction accuracy on mining tasks that take longer than tasks that take shorter. Due to these considerations, we adopted the following training point selection strategy: an input is included into the training set if the fastest algorithm takes longer than 2 seconds and the fastest algorithm is at least faster than the second fastest by 10 seconds. When using this strategy, 2360 points remained in the training set and were used to train the SVM model.

SVM learning module: In our system, we use the popular SVM library *libsvm* [CL01] as the learning module. We choose to use the *RBF* kernel, because it offers non-linear learning capability and has fewer parameters to tune than the *polynomial* kernel. We directly take advantage of *libsvm*'s multi-class classification functionality to predict

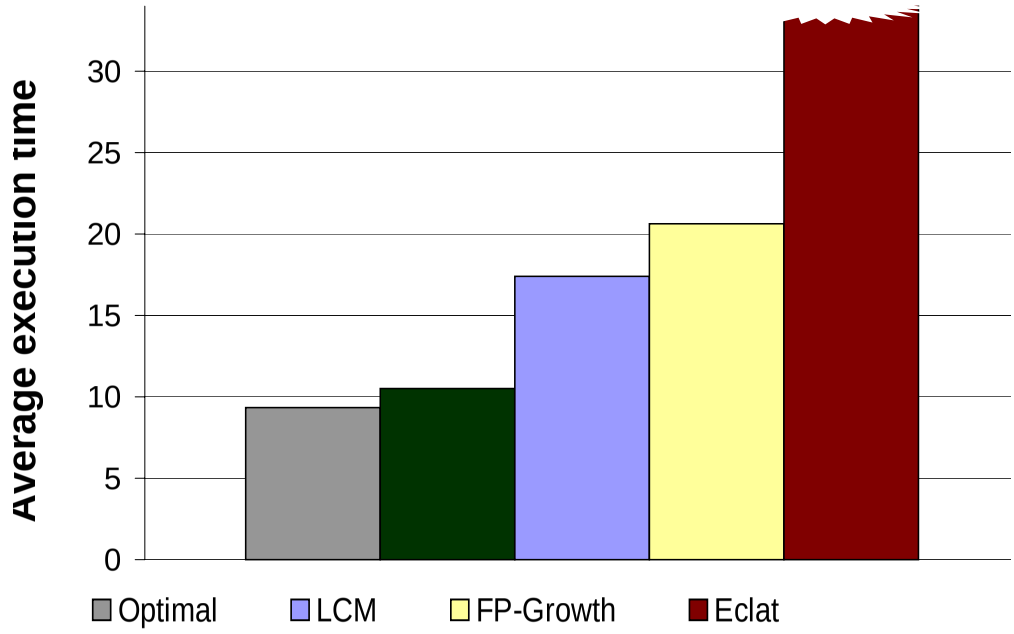


Figure 7.11: Average execution time (in seconds) of the predicted algorithm compared with those of the optimal and the single algorithms.

on the three algorithms.

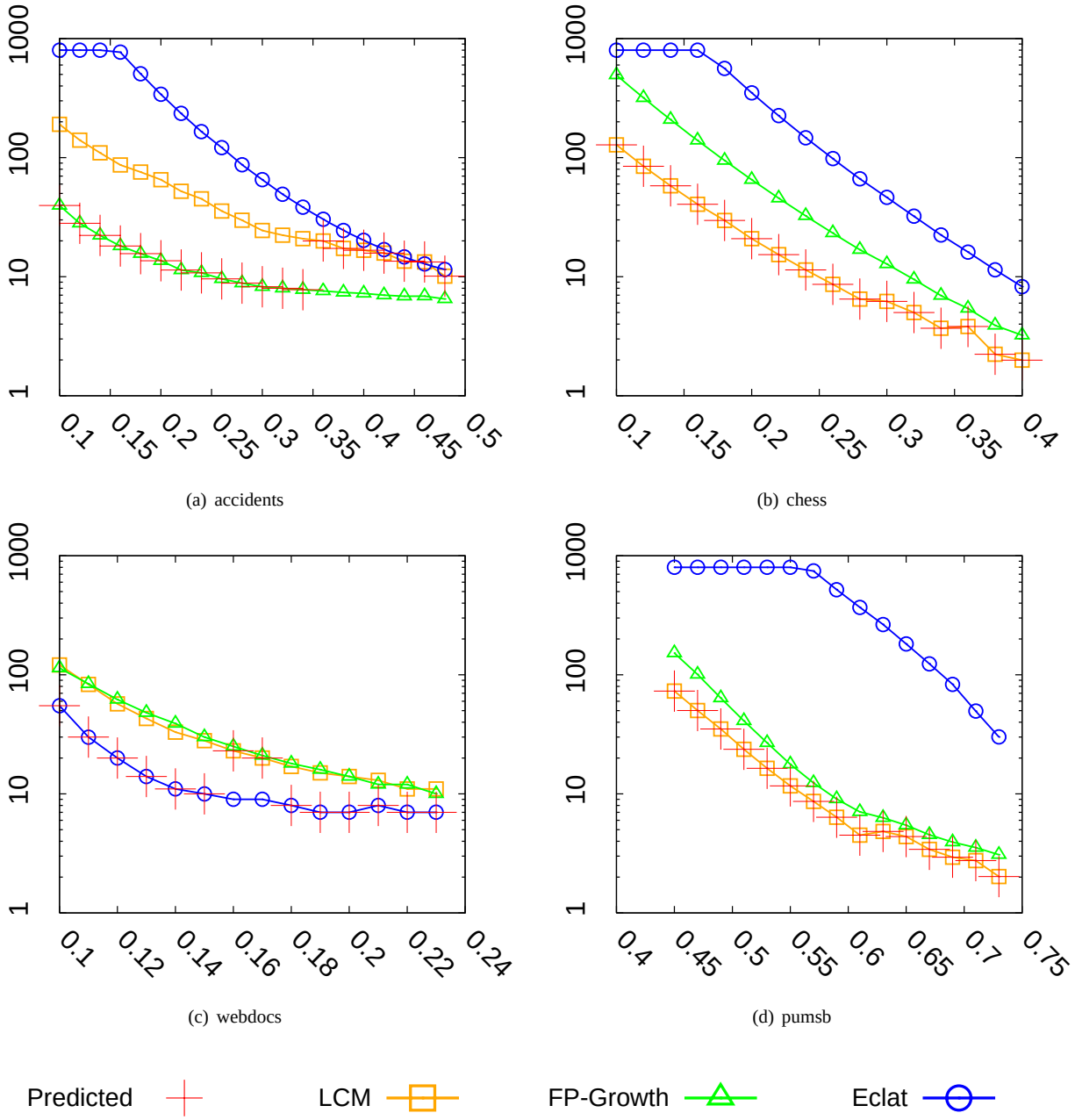
7.3.2 Prediction Results

The trained SVM model is used to predict the best algorithm. Figure 7.11 shows the average execution times for the optimal algorithm, the algorithm SVM predicted to be the best, and the three single algorithms on the 12 real-world data sets. Optimal shows the average execution time when the best algorithm is used to perform the mining. Predicted shows the average execution time when using the algorithm predicted by the SVM classification system. As it can be seen, the execution time of the predicted algorithm is very close to that of the optimal algorithm, and significantly better than of any single algorithm.

To better understand the impact of the selected features on the performance of the mining algorithms Table 7.3 lists for the 12 real-world inputs and a representative support threshold, the corresponding feature values, the three algorithms' execution time on it, and the algorithm predicted by the SVM system. The "*gain*" field computes the execution time saved from using the second fastest algorithm when the prediction is correct or the penalty in case of misprediction. As it can be seen, there are four data sets, "*accidents*", "*chess*", "*pumsb*" and "*webdocs*" where the fastest algorithm significantly outperforms the second fastest. Notice that each of the three algorithms is best for at least one of these four data sets, so that using a single algorithm will significantly degrade performance. The

dataset	support	size	similarity	density	height	LCM	FP_Growth	Eclat	predict	correct	gain
*a*ccidents	0.1	10125064	0.6334	0.3968	0.7480	190.53	39.61	800.00	**FP_Growth**	√	**+150.92**
chess	0.1	116661	0.8283	0.5984	0.8332	127.93	493.54	800.00	**LCM**	√	**+365.61**
webdocs	0.1	24493165	0.2699	0.1772	0.4356	121.00	114.00	55.00	**Eclat**	√	**+59.00**
pumsb	0.45	2251016	0.9170	0.7285	0.3823	72.99	152.26	800.00	**LCM**	√	**+79.27**
pumsb_star	0.1	1996699	0.7520	0.3701	0.7298	38.04	22.65	800.00	LCM	×	-15.39
connect	0.4	2310387	0.9137	0.8341	0.5205	2.96	4.00	800.00	LCM	√	+1.04
mushroom	0.01	186092	0.7769	0.2386	0.9582	1.33	0.82	238.34	LCM	×	-0.51
retail	0.001	665660	0.2312	0.0035	0.7171	0.98	3.72	138.40	FP_Growth	×	-2.74
kosarak	0.1	1612992	0.9773	0.4073	0.7545	2.62	4.81	10.34	LCM	√	+2.19
BMS-POS	0.0004	3323594	0.3072	0.0102	0.9610	20.95	22.66	400.00	LCM	√	+1.71
BMS-WebView-1	0.0005	148407	0.4408	0.0067	0.9269	16.15	7.42	400.00	LCM	×	-8.73
BMS-WebView-2	0.0005	334046	0.1879	0.0023	0.7859	0.74	2.75	102.07	LCM	√	+2.01

Table 7.3: Feature values and prediction results on 12 real inputs.



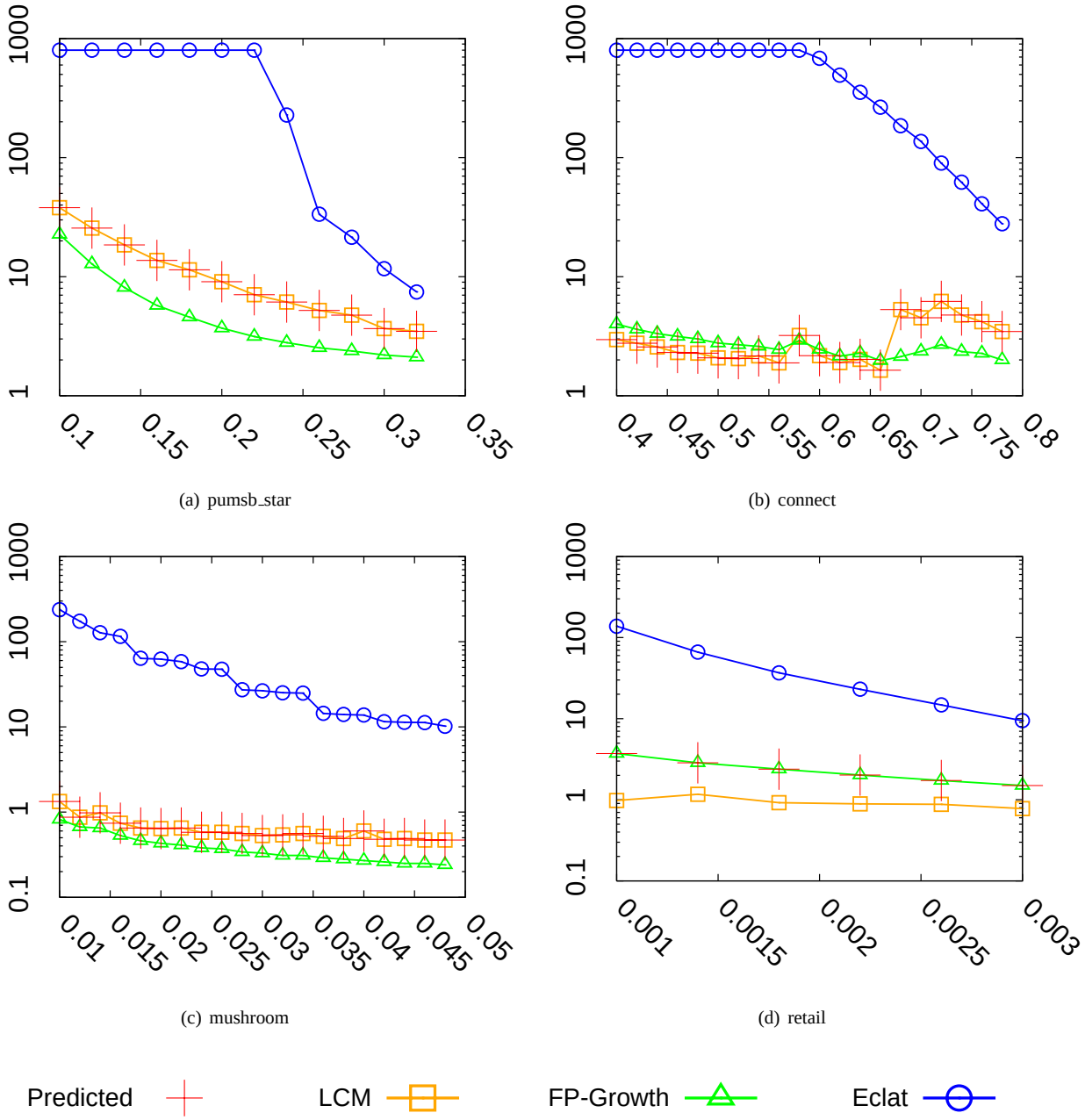
(a) accidents

(b) chess

(c) webdocs

(d) pumsb

Figure 7.12: Prediction results on four real data sets: *accidents*, *chess*, *webdocs* and *pumsb*. The X-axis plots the support threshold, while the Y-axis plots the execution time in seconds using a logarithmic scale.



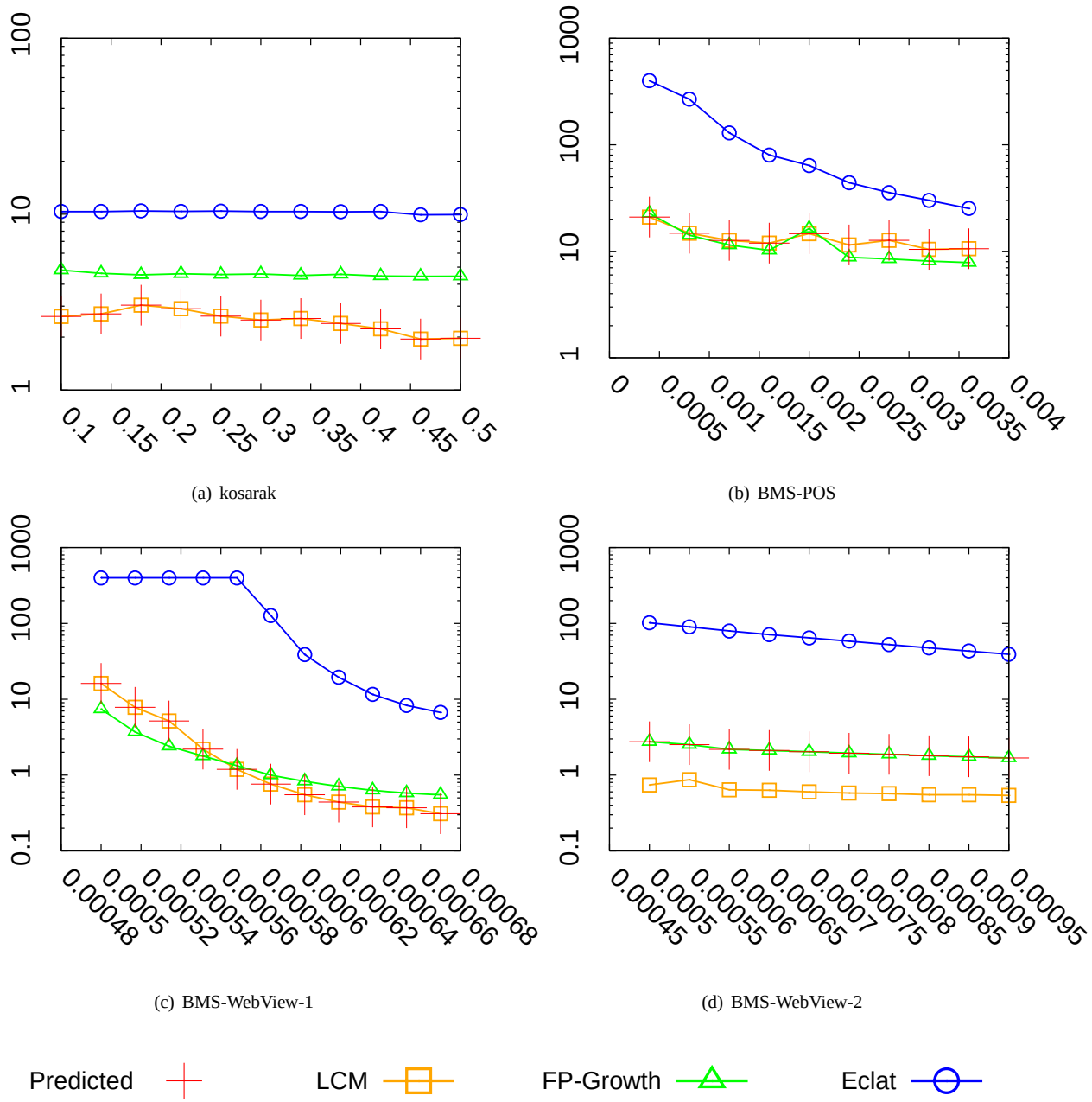
(a) pumsb_star

(b) connect

(c) mushroom

(d) retail

Figure 7.13: Prediction results on four real data sets: *pumsb_star*, *connect*, *mushroom* and *retail*. The X-axis plots the support threshold, while the Y-axis plots the execution time in seconds using a logarithmic scale.



(a) kosarak

(b) BMS-POS

(c) BMS-WebView-1

(d) BMS-WebView-2

Figure 7.14: Prediction results on four real data sets: *kosarak*, *BMS-POS*, *BMS-WebView-1* and *BMS-WebView-2*. The X-axis plots the support threshold, while the Y-axis plots the execution time in seconds using a logarithmic scale.

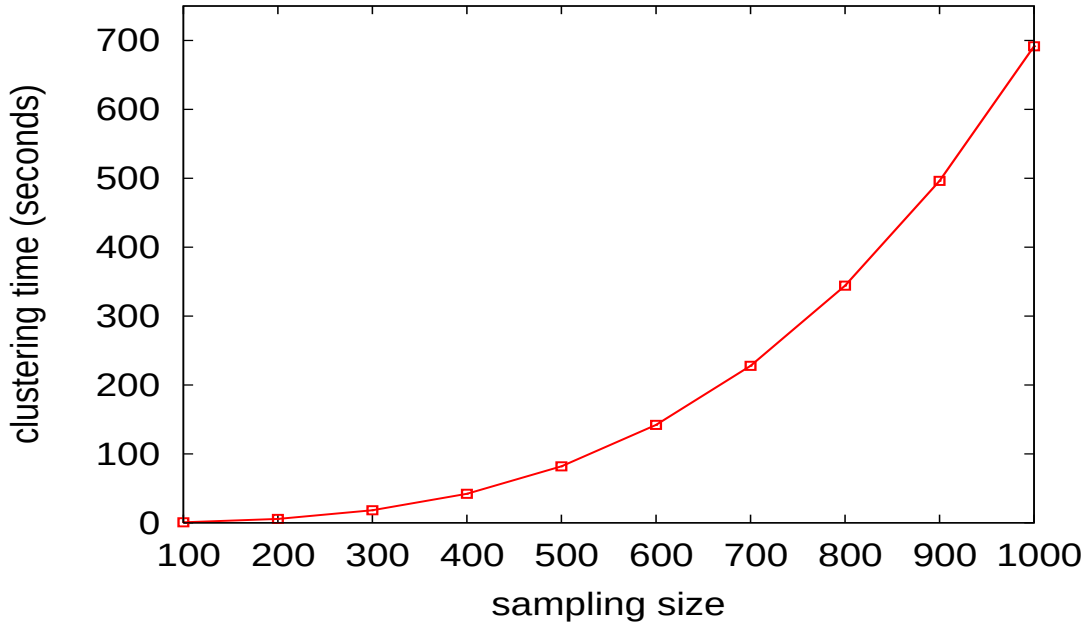


Figure 7.15: Clustering time vs. sampling size.

SVM system correctly predicts on these important inputs, although it mispredicts on some inputs where the penalty of misprediction is relatively small. This misprediction is mainly due to our training point selection strategy described in Section 7.3.1 that emphasizes prediction accuracy on important tasks.

The feature values in Table 7.3 can explain why each algorithm wins on the four important inputs. For example,"*webdocs*" has a large "*size*", its little "*similarity*" and small "*height*" results in a shallow search space, and the "*density*" is reasonably big. As a result, Eclat is better than the other two algorithms. Both "*chess*" and "*pumsb*" have small "*size*", high "*similarity*" and large "*height*", what makes LCM's "*hyper-cube decomposition* much more efficient by constructing significantly fewer transaction databases. "*Accidents*" has a large "*size*" and medium "*similarity*", which makes "FP_Tree" representation more efficient than LCM's array representation.

Figure 7.12, 7.13 and 7.14 shows the prediction results on all the real-world inputs for different values of support threshold. In most cases the prediction identifies the best algorithm, and in case of misprediction the selected algorithm is still competitive in performance. Notice that the prediction is more accurate on inputs that take longer to mine than on the smaller inputs.

7.3.3 Prediction Overhead

We measured the runtime prediction overhead of our prediction system, which consists of two parts: the extraction of the features and the prediction using the SVM model. The "*size*", "*density*" and "*height*" features can be extracted while filtering out infrequent items, therefore they incur negligible overhead. The "*similarity*" feature is expensive to

obtain because the "*hierarchical clustering*" algorithm has $O(N^2)$ complexity. For our current system, the clustering is implemented in Perl without any performance optimization consideration. Figure 7.15 shows the execution time of the clustering algorithm versus the sampling size. As it can be seen, the execution time grows very fast with increasing sampling size. However, as shown in Figure 7.4, increasing sampling size does not affect "*similarity*" value much. Hence, in our system, we used the sampling size of 100, whose clustering time is less than 0.5 seconds. Calling the SVM module to perform the prediction is usually completed in less than 10 miliseconds. Thus, the overall runtime prediction overhead is less than 1 second. For non trivial mining tasks which take longer than one minute, this overhead is negligible.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In conclusion, the thesis extends the automatic library generation methodology to two new domains, a specialized architecture and a new application domains. Specifically, we built the first automatic tuning system to generate numerical libraries for graphics hardware. We studied automatic generation of high-performance matrix multiplication on graphics hardware, as matrix multiplication is the most important building block for a variety of numerical libraries. In contrast to ATLAS [WPD01], which utilizes register tiling, cache blocking and instruction scheduling to achieve high performance on pipelined processor with deep memory hierarchy, our approach automatically tunes matrix multiplication to graphics hardware's unconventional architecture features, such as SIMD instruction with swizzling and smearing, multiple-render-targets, limited instruction count, limitation on branch instruction, varying shader models, etc. Our automatic tuning system is capable to generate matrix multiplication implementations with comparable performance to expert manually tuned version despite the significant overhead incurred due to the use of a high level language.

To attack problems where the relative performance among several algorithms differ on varying input data, we implement an automatic algorithm selection system for frequent pattern mining problem using support vector machine. In particular (1) we identify a set of input features that can guide the selection and (2) we show how to generate synthetic data sets that are representative of the real-world data sets and that can be used to train the input classifier. As a result of our contribution, we obtain an algorithm that is better that any of the three selected algorithms (LCM, FP_Growth, and Eclat). Our experiments show that the performance of the predicted algorithm is on the average only 12.5% worse than that of the optimal algorithm, and 65.3% better than LCM that obtained the best average performance for all the tested inputs. We believe that the approach will extend to other choices of basic frequent pattern mining algorithms.

8.2 Future research directions

There are several potential future research directions to extend the work presented in the thesis.

First it is possible to build library generators on graphics hardware for other libraries, such as sorting, signal processing, etc. Automatic tuning for sorting and signal processing for general purpose processors has been extensively studied by previous research. Mapping sorting and signal processing algorithms for graphics hardware have also been well studied. Extending the library generator for sorting and signal processing on graphics hardware generally should be a natural extension of the thesis's work.

Second, as graphics hardware evolves so fast, in the past one to two years after the work of automatic matrix multiply library generator for GPUs was built, graphics hardware underwent a significant advancement in architecture. Its architecture design and features have dramatically changed and become more general purpose. New high-level programming models is supported by nVidia graphics cards so that our approach becomes more practical. It is worth studying the opportunities in adapting the automatic matrix multiply library generator to the new architecture, which significantly divert from the previous architecture models.

Third, new specialized architectures such as the Cell processors and network processors also offers tremendous computing power to utilize. How to build library generators for these new specialized architectures poses new challenges different from those for graphics hardware. Extending automatic library generation for these architectures should make significant contributions as well.

Fourth, in the thesis, we only picked three algorithms and focus on the algorithm selection problem on them. The features selected are handmade according to the algorithms' performance characteristics. It would be an interesting and challenging research topic to extend the research into a more general framework, so that it can be applied to more algorithms and be as oblivious to each individual algorithm's performance behavior as possible.

Fifth, in this thesis, the algorithm is selected in the beginning of the mining process. As the frequent mining problem is a recursive process, which can be decomposed into small similar problems as the recursion goes down. It would be an interesting and challenging research topic to build a hybrid mining algorithm out of single algorithms by automatic switching between different algorithms in different stages of the recursion. This method has actually been successfully tested in building hybrid sorting algorithms out of well known sorting algorithms.

References

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining Association Rules between Sets of Items in Large Databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[Ake93] Kurt Akeley. Reality engine graphics. In *Computer graphics and interactive techniques*, pages 109–116. ACM Press, 1993.

[AM03] Ádám Moravánszky. Dense matrix algebra on the gpu, 2003. http://www.shaderx2.com/shaderx.PDF.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[BAwCD97] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *International Conference on Supercomputing*, Vienna, Austria, July 1997.

[BCG01] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. In *ICDE*, pages 443–452, 2001.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *SIGGRAPH*, August 2004.

[Bor04] Christian Borgelt. Efficient Implementations of Apriori and Eclat. In *FIMI*, 2004.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *ACM SIGGRAPH Graphics hardware*, pages 37–46, 2002.

[CL01] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a Library for Support Vector Machines*, 2001. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation",.

[Fri99] M. Frigo. A Fast Fourier Transform Compiler. In *PLDI'99 — Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.

[FSH04] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *ACM SIGGRAPH Graphics hardware*, 2004.

[GBP+05] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-conscious Frequent Pattern Mining on a Modern Processor. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 577–588. VLDB Endowment, 2005.

[GLW+04] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, pages 215–226. ACM Press, 2004.

[Goe02] Bart Goethals. *Efficient Frequent Pattern Mining*. PhD thesis, University of Limburg, Belgium, 2002.

[GZ01] Karam Gouda and Mohammed Javeed Zaki. Efficiently Mining Maximal Frequent Itemsets. In *ICDM*, pages 163–170, 2001.

[GZ03a] Bart Goethals and Mohammed Javeed Zaki, editors. *FIMI '03, Frequent Itemset Mining Implementations, Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, 19 December 2003, Melbourne, Florida, USA*, volume 90 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[GZ03b] G. Grahne and J. Zhu. Efficiently Using Prefix-trees in Mining Frequent Itemsets. In *FIMI*, 2003.

[Har05] Mark Harris. Mapping computational concepts to gpus. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 50, New York, NY, USA, 2005. ACM Press.

[HCH03] Jesse D. Hall, Nathan A. Carr, and John C. Hart. Cache and bandwidth aware matrix multiplication on the gpu. Technical Report UIUCDCS-R-2003-2328, University of Illinois Urbana Champaign, 2003.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining Frequent Patterns without Candidate Generation. In Weidong Chen, Jeffrey Naughton, and Philip A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, 05 2000.

[IYV04] Eun-Jin Im, Katherine A. Yelick, and Richard Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[JGZ04] Roberto J. Bayardo Jr., Bart Goethals, and Mohammed Javeed Zaki, editors. *FIMI '04, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Brighton, UK, November 1, 2004*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[KF05] Emmett Kilgariff and Randima Fernando. The geforce 6 series gpu architecture. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 29, New York, NY, USA, 2005. ACM Press.

[LBNA+03] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, James McFadden, and Yoav Shoham. A Portfolio Approach to Algorithm Selection. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1542–1543, 2003.

[LGP04] Xiaoming Li, María Jesús Garzarán, and David A. Padua. A Dynamically Tuned Sorting Library. In *CGO'2004 — IEEE / ACM International Symposium on Code Generation and Optimization*, San Jose, 2004. IEEE Computer Society.

[LGP05] Xiaoming Li, Maria Jesus Garzaran, and David Padua. Optimizing sorting with genetic algorithms. In *CGO'05*, pages 99–110. IEEE Computer Society, 2005.

[LM01] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55. ACM Press, 2001.

[LPWH02] Junqiang Liu, Yunhe Pan, Ke Wang, and Jiawei Han. Mining Frequent Item Sets by Opportunistic Projection. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 229–238, New York, NY, USA, 2002. ACM Press.

[MA03] Kenneth Moreland and Edward Angel. The FFT on a gpu. In *ACM SIGGRAPH Graphics hardware*, pages 112–119. Eurographics Association, 2003.

[MBDM97] John Montrym, Daniel Baum, David Dignam, and Christopher Migdal. Infinitereality: a real-time graphics system. In *SIGGRAPH*, pages 293–302. ACM Press/Addison-Wesley Publishing Co., 1997.

[nVi] nVidia Corporation. NVIDIA Cg Toolkit. http://developer.nvidia.com/object/cg_toolkit.html.

[Owe05] John Owens. Streaming architectures and technology trends. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 9, New York, NY, USA, 2005. ACM Press.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *SIGGRAPH*, 21(3):703–712, July 2002.

[PHL+01] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. In *ICDM*, pages 441–448, 2001.

[PSX+04] Markus Püschel, Bryan Singer, Jianxin Xiong, José Moura, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, February 2004.

[Ric76] J.R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.

[SON95] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In *The VLDB Journal*, pages 432–444, 1995.

[Sta] Stanford Univ. Graphics Lab. GPU benchmark suite. http://graphics.stanford.edu/projects/gpubench.

[TTT+05] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 277–288, New York, NY, USA, 2005. ACM Press.

[UAUA03] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets. In *FIMI*, 2003.

[UKA04] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets. In *FIMI*, 2004.

[Vap95] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[YLR+03] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David A. Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A Comparison of Empirical and Model-driven Optimization. In *PLDI*, pages 63–76, 2003.

[ZG03] Mohammed J. Zaki and Karam Gouda. Fast Vertical Mining using diffsets. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335, New York, NY, USA, 2003. ACM Press.

[ZKM01] Zijian Zheng, Ron Kohavi, and Llew Mason. Real World Performance of Association Rule Algorithms. In Foster Provost and Ramakrishnan Srikant, editors, *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 401–406, 2001.

[ZPOL97] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New Algorithms for Fast Discovery of Association Rules. In David Heckerman, Heikki Mannila, Daryl Pregibon, Ramasamy Uthurusamy, and Menlo Park, editors, *In 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 283–296. AAAI Press, 12–15 1997.

Author’s Biography

Changhao Jiang was born in Jiang Xi province and grew up in Nanjing, China. He attended the Attached Middle School of Nanjing University of Aeronautics and Astronautics (a.k.a. Nanjing No. 7 middle school). After a short period of study at Tsinghua High School in Beijing, he entered the Department of Computer Science and Technology of Tsinghua University, China in 1995. He received both of his Bachelor and Master degrees in Computer Science from Tsinghua University in 1999 and 2001 respectively. While in Tsinghua, he made major contributions to the research project — “Smart Classroom” by designing and implementing the middleware software infrastructure for it. After receiving his Master degree, he joined the Computer Science Department of Carnegie Mellon University as a research staff and researched on location-aware application in the “Aura Project”. Two years later, he started his Ph.D. study at the Computer Science Department of University of Illinois at Urbana Champaign. His Ph.D. study mainly focused on automatic performance tuning and algorithm selections.